Preface

What this book is intended for

This book is intended for a first course in Java programming. Java is an object language. Programming in an object language is mostly about writing classes, so class design and implementation is presented from the beginning.

But, it takes three or four weeks for most people to internalize enough expertise to understand classes, so the first several chapters (after the introduction) are written in a quasi-tutorial style. The reader/student is directed, repeatedly, to put down the book and go to the screen to try out code. This is essential. Programming, like juggling, can't be learned without practice. It is a process skill and process skills must be practiced.

To the student:

Welcome! Computing is changing our world. It is not easy to learn at first, but once you start to understand it, it can be fun and remunerative. Just remember: practice, practice, practice! And, don't be afraid to experiment -- that is how you learn.

To the teacher:

The way to teach from this book is to do the example programs just before the students. That way, you will have them clearly in mind, and will just have encountered some of the problems they will run into. Carefully explaining bugs you had, how you diagnosed and solved them, is invaluable to the students. Debugging is not easy; they need a model.

There are some review problems and programming exercises at the end of each chapter, but it is intended that the instructor will come up with other programming problems that seem interesting.

If you have students who are not comfortable with files and directories, they will need some extra help with those, they are not covered here.

In closing:

This is a work in progress. There are necessarily numerous errors, omissions, and sections in need of improvement. Feedback is welcome at levenick@willamette.edu

Good luck!

jrl

Chapter 1: Programming is like juggling

A. Computing as a fact of life

This millennium is an exciting time to be involved in computing. The web, which did not exist 20 years ago (!), is an important new part of our society, our culture. Ordinary people, with no training in computing, can sit at a personal computer and access millions of other computers across the planet. Anyone with patience and the ability to follow simple instructions can make a web page and post information that is immediately accessible from around the world. This is a revolutionary development. Java is a programming language designed for programming on the web.

B. What is computing anyway?

i) Computing as information processing

Computing is information processing. Always. Only. Although many people experience computing as web browsing, or chatting, or game playing, the underlying programs are always processing information by executing instructions.

A program is a series of instructions that is executed by the processor. Each instruction does some small task like moving a little information from one place to another, or adding one number to another and storing the result, or comparing one number to another and taking different actions depending on which is bigger. Although each individual action is small, a modern processor executes billions of instructions a second and so can accomplish a fair amount in a short time.

Information is stored mainly in two places, temporarily in memory and permanently in files on disk. A file is a series of numbers. This is the same "file" you might store from a word processor or an email program. When you save a file in a word processor, the information in your document (i.e. the text you have typed, along with the formatting information) is converted to a series of numbers and written on the disk along with the name of the file that you select. Later, when you load that file again, the numbers are converted back into text to be displayed. The files you save (and the information in the files) remain on the disk indefinitely unless you delete them.

Information in memory, by contrast, lasts only so long as the machine is turned on and the program that is using that memory is running. When the program terminates that memory may be used for other processes. When the machine is turned off, all the information in memory is lost.

a) State and its representation

In an ordinary digital computer, memory is measured in bytes. It is common, in 2004, for machines to have a gigabyte of memory; that means 2^{30} bytes. Each byte is composed of 8 bits, and is just enough memory to hold a single letter. As you have likely heard, all information in a digital computer is ones and zeros; since there are 2 possible values it is called binary. A single **bi**nary dig**it** is called a bit. Memory is a long sequence of numbered locations, each of which can hold one byte. The good news is, programming in Java, you will almost never need to deal with bits, bytes or memory locations.

You will, though, need to understand that information in your program is stored in variables, which are associated with particular locations in memory, and that a particular variable may take on different values at different times. The current value of a variable is referred to as its "state". This is an ordinary usage of the word state, but perhaps not a common one. If it seems confusing, you might think of the "state of a light bulb" (i.e. either on or off), or the "State of the Union".

b) Definition of algorithm

A program is an algorithm written in a particular programming language. That’s fine, but what’s an algorithm?

Rough Definition: An algorithm is a step-by-step description of a process to solve a problem.

Thus a recipe is, roughly, an algorithm. It lists the various ingredients that should be added in what order, and the cooking or baking process. Most recipes are not quite algorithms because they require judgment to carry out correctly.

Better definition: An algorithm is a step-by-step description of a process, where each step is described explicitly and requires no judgment, to solve a problem, or a class of problems.

The reason many recipes do not fit this definition is that they include directions like, "Bake until done", or "Cook until just tender.". While any experienced cook understands these instructions, the cook must exercise judgment to follow them. People do this very well, but computers do not.

So, when presented with a problem to be solved by a program, a programmer's job is first to formulate an algorithm which will solve that problem, and then to convert it into a programming language, so that it can be executed (i.e. carried out) by a computer.

c) Action! The only three statements that do anything

The remainder of this book concerns techniques that allow you to solve problems and implement algorithms in Java. There will be several hundred pages explaining control structure, class structure, objects, expressions, methods and data structures. Oddly, in spite of all that, there are only three actions that actually accomplish anything. Input statements and assignment statements change the state (or value) of variables; and output statements send information out of the program (usually for a human to read). That's it, three things. Input, assignment, output.

1. Input statement - bring information into the program
2. Assignment statement - change the information in (state of) a variable
3. Output statement - send information out of the program

Everything else in a program, all the hours that a programmer spends designing and coding and debugging, only arranges for those three things to happen the right number of times and in the right order. This may seem strange, but it is true.

d) Structure: everything else

The parts of a program besides input, assignment and output may be divided into three categories:

1. Control structure - selects which statements are executed, in what order, and how many times
2. Data structures - organize data (information) so that it is more convenient to access
3. Class structure - organizes classes so that they are easier to understand, modify and work with.

These three will be the subject of much of the rest of this text.

ii) Computing as a revolution

Computing, viewed from the inside, is always about information; inputting information, processing information and outputting information. Viewed from the outside, it is an exciting and revolutionary development. It is transforming our world, our ways of communicating, learning, playing, and our understanding of ourselves. The changes computing will bring are mostly in the future; I am fond of saying, "The computing revolution has not properly started yet.".

a) The web as a new cultural phenomenon

Some people think of "culture" as meaning opera or art galleries. There is a broader meaning that includes language, education, technology and even dating. Many everyday activities are being transformed by the web. How we shop; how we communicate with our friends, family, and co-workers; how our cars and televisions work; how we plan and spend our time; these and more are done differently if we have an internet connection nearby.

b) People have never had a tool for processing information before

We are only just beginning to learn how to use computing. Nothing is settled yet. The leading hardware and software producers come out with updates monthly. But the way we deal with information has been changed forever. A good example is Google. It sends bots out to collect information in the dead of night (i.e. when the net is not so busy), then catalogs and indexes it so when people type in queries it can respond quickly and direct them to relevant web sites. If you’re used to Google this may seem like no big deal. But it is. Search engines and the web allow information to be disseminated orders of magnitude faster. Even research scientists use search engines instead of spending long hours in libraries searching for paper copies of research articles. What will this mean for our culture? Who can say? But, it will certainly change it.

c) Automated reasoning and process

Computing also allows us to study process. A program is a mechanization of information processing. Before computers, information processing was only done by people (and other natural systems). We have learned some little about process thus far, but who knows what the future will bring?

d) Artificial Intelligence?

Artificial intelligence is an exciting and alarming possibility. Might we all be replaced by intelligent machines? There have been many projects starting in the 1950’s to build programs to do language understanding, automatic translation, scene analysis, and more recently build autonomous vehicles. Thus far, successes have been extremely limited. So, although computers are everywhere, blindingly fast, and never forget; to call them stupid would be a compliment; they have *no* intelligence. Nevertheless, clever programmers can make them do some amazing things.

One reason artificial intelligence is so difficult is that while computers can add numbers millions of times a second, and store and access millions of facts or rules without ever forgetting even one, they do not learn as people do, they do not form and apply concepts flexibly. People are the product of billions of years of evolution and we have highly developed and highly specialized information processing capacities. Additionally, people are born with a set of unconscious patterns corresponding to important relationships and ways of thinking. Important how? Evolutionarily. Certain ways of processing information are more adaptive than others, and individuals must have survived long enough and done the right things to have and raise children, or they are not our ancestors. Our fascination with sex and violence is not accidental. Neither is our love for children. Our abilities to communicate with metaphors, to ascribe meaning, and to discern pattern in noise are beyond the reach of any computer yet programmed.

The enterprise of attempting to create intelligence in a digital computer has been likened to trying to climb a tree to the moon. Early, preliminary attempts seem to yield good progress ("Look! See how high I’ve climbed already?"), but sooner or later frustrating impasses always seem to loom ("Huh, the branches are getting pretty skinny up here."), and eventually the project is abandoned ("But... maybe if I found a tree on a very high hill?"). There’s no way to know how apt that metaphor is (and that’s always a question you should ask when encountering a metaphor, otherwise you risk being led badly astray!). Everyone knows you cannot climb a tree to the moon, whereas whether a digital computer could have intelligence remains to be seen.

e) Societal impacts

Our culture is changing in many ways because of computing, here are just three examples.

1) Around the middle of the second millennium kingdoms arose in Europe. In the days before computing, to raise an army to make war on your neighbor, or defend yourself from them, required first assembling a bureaucracy. A small army of clerks and functionaries was needed to coordinate the calling up, feeding, housing and supplying of any army. This meant that to wage war a ruler had to enlist or compel the cooperation of

many many civilians. Nowadays, one person with a thousand dollar machine plus access to appropriate databases and software could coordinate much of that without assistance. Will this change warfare? Has it already? Look around. Watch and see.

2) Many Americans put a premium on their privacy. They hate the idea of anyone compiling and/or selling data about them without their permission and cherish the notion of relative anonymity in transactions on the web. But the nature of computing and the web means that privacy is essentially an illusion. If you send email across the country there are copies of it on at least several mail servers; worse, there are logs. Admittedly, only system administrators can access them, but if there were a reason to, they could.

Uninformed people may imagine that you can do things anonymously on the web. It is true that your personal identity is hidden, but where you are sitting and your ISP are definitely not a secret. If they were, there would be no way for the web page you are receiving information from to send it to you. Even though you can’t see them, there are servers and routers relaying the packets across the planet to your screen; and they are all logging all the transactions, just in case someone needs to find out who was accessing what.

3) Some people predict a collapse of traditional brick and mortar commerce. As more and more goods are sold over the web, there is less and less need for physical stores. Of course there will always be stores, but perhaps soon there will be considerably less of them.

C. The past: A (very) brief history of computing

Although it is not strictly necessary to know the history of computing before learning to program, there are some interesting perspectives one may develop by doing so. Therefore, I offer this brief history.

i) Living in scaffolding - a cautionary tale

A cathedral took many years to build. Imagine building a gigantic stone structure without power tools. Huge blocks of stone, high walls of stone. To place a block of stone on a wall, you must first lift it up and then set it in place, making sure that it fits tightly. This requires solid, strong, more or less permanent scaffolding.

It was common for the workers to construct living quarters in the lower levels of the scaffolding (alongside the already completed walls). Likely because this was a convenient and easy place to build a shelter to cook for the workers and for the workers to eat during inclement weather. As years turned into decades and the walls (and the vacated scaffolding next to them) grew higher, it was simply easier for the families of the workers to take up residence there. As decades turned to centuries, and construction was interrupted by wars or plagues, people were born, grew, raised children and died, living in the scaffolding, knowing no other life. There were even cases where the original project was abandoned and the workers settled into the scaffolding as a permanent residence.

There is a danger, in any endeavor, that temporary measures, adopted as a means to an end, remain in effect for so long that practitioners no longer remember that they were not the goal of the project. This is not, generally, a good situation. Digital computers are a case in point.

ii) Why are we still using this prototype?

This text will emphasize generic problem solving principles, in addition to Java programming. They will be used in service of programming, but will typically be applicable to many other areas. I use the phrase "problem solving" in a special sense here. It is the activity that one engages in when one is trying to accomplish something and runs into a problem that stops them. These problem solving principles are for when you *don't* know how to cope with a problem. Here's the first (and perhaps the most general).

a) Problem Solving Principle #1 - Build a prototype

When attempting to solve a difficult problem, first build a prototype that solves a similar problem that is simpler or smaller. This is useful for several reasons. If you're stuck on a problem, sometimes a smaller or simpler version of the same problem will be easy. Plus, sometimes the reason a problem stops you is that you've never thought carefully about anything like it. The cognitive structure you generate by solving the simpler version sometimes allows you to see through the more complex problem.

Here's a problem that will perhaps stop you. Assume there are 32 people in a classroom. Each person is asked whether or not they have ever been snowboarding. They each come to the desk at the front of the room, and write their answer on a piece of paper on the desk; the first on the first line, the second on the second line, and etc. How many possible different sequences of yeses and nos are there?

Perhaps you've been studying combinatorics and know the answer right away. If not, read on. What's the smallest we could make the problem? How many people? Right, one. One person can write one of two things; yes or no -- so there are two possible sequences (if you can call one thing a sequence). The good thing about solving a problem with one thing instead of n things is that it is usually trivial. The bad thing is that it may not tell you much.

So, try two people. The first can write either yes or no. Then the second can write either yes or no. If the first person writes yes, the possible pairs are {yes, yes} and {yes, no}. If the first person writes no, they are {no, yes} and {no, no}. So four total.

The trick, now, is to generalize to n people, or in this case 32. For one person the answer was two for two people it was four; can you discern the pattern? Commit to a pattern. Then see if it is true for three people. Here's how I do the analysis for three.

As we just saw, for the first two people there were four possible states of the list:
{yes, yes}
{yes, no}
{no, yes}
{no, no}
The third person can either add yes or no, giving these possibilities (the third person's answer is in **bold**):
{yes, yes, **yes**}
{yes, yes, **no**}
{yes, no, **yes**}
{yes, no, **no**}
{no, yes, **yes**}
{no, yes, **no**}
{no, no, **yes**}
{no, no, **no**}
Twice as many possibilities as with 2 people. So the number of possible states is not twice the number of people, but rather, it starts at 2 and *doubles* with each additional yes or no added to the list. Thus, you can see the answer with 32 people is, 2*2*2*2...*2, 32 times, which is written 2^{32} (that's somewhat more than 4 billion). Remember this number, 2^{32}, and how it was derived; you will meet it again later.

b) Von Neumann's prototype

The second example of building a prototype of a system involves the inventor of the first digital computer, John Von Neumann. In the 1930s Von Neumann set out to build a thinking machine. Since the only things we know of that think are brains, he decided to model a brain (an eminently reasonable idea!). He was under the impression that brains were analog devices, made up of neurons that were more or less active depending on their inputs. Remember, in those days there were no transistors, so he was working with vacuum tubes, which are pretty good analog devices. He decided his computing machine would have a number of elements with values that ranged between 0 and 1, where 0 represents all the way off, 1 all the way on, 0.5 half way on, 0.75 three quarters on, and etc. He started trying to settle on the details of this thinking machine, and found himself stuck on how exactly to implement it. So, he decided to build a similar, but simpler machine; namely one with only two states, 0 and 1; a binary computer.

It turned out that even this simpler machine was not trivial to implement, but with time and persistence they got it to work. Von Neumann died before he ever built the real thinking machine, and here we are living in the scaffolding. Perhaps Von Neumann was right; perhaps mechanical intelligence will require analog computers. We shall see.

iii) Evolution of computing

The rate of evolution in computing is astounding. Digital computing is changing so rapidly for a number of reasons: a) it is new, b) hardware is improving rapidly, c) software is improving rapidly, d) more and more people are getting involved, both as users and programmers. Any human endeavor, at its inception evolves rapidly. Early, simple, clumsy, poorly conceived systems give way to more sophisticated, better debugged, easier to use systems.

a) Hardware

In the 60 years that computing has existed, computers have been utterly transformed. Early computers were made with vacuum tubes, like very old radios and televisions. The invention of the transistor allowed a truly digital device. The first transistors were hand-made and large. They were mounted on circuit boards and wired together. Soon hundreds of tiny transistors were being packaged on a chip; these were called integrated circuits (ICs) and were mounted on similar boards. For example, in a 1960’s computer the central processing unit (CPU), or processor, which does the arithmetic and logic, was a board perhaps 18" square, packed with chips on one side and festooned with more wires than you’d want to count on the other. Before long tens of thousands of transistors were being packed into a chip, a technique, then called very large scale integration (VLSI). One day, someone managed to pack all the functionality for a CPU onto one chip, and the age of the microprocessor began.

A microprocessor can run much faster than any processor spread out on a board for one simple reason; the information in the processor has a shorter distance to travel. Several facts will help illuminate this. Electrical signals travel at about 0.6c; where c is the speed of light. Modern processors commonly have clock speeds of 4gHz or more. A clock speed of 1gHZ means the clock ticks a billion times a second, the time from one tick to another is a billionth of a second, a nanosecond. Light travels about a foot in a nanosecond. So, if components of the CPU were a foot apart, there is no way a signal could get from one to the other in time for the next clock cycle. That would slow down the processor. Thus, every modern processor is a microprocessor and every modern computer a microcomputer.

b) Education, language and culture

In spite of that, there are books and people who still talk about "mainframes". Some out of date (but still in publication) books include typologies of computers including: mainframes, mid-sized computers, mini-computers and microcomputers, as if it is a spectrum from large and powerful to small and not, when the reverse is typically true. Why is this?

This is a characteristic of the slow evolution of culture and language. Language in a rapidly changing culture lags behind the phenomena it describes. An example is "floppy disks" as the name of 3 1/2 inch removable disks. They replaced the 5 1/4 inch removable disks, which were actually flexible (i.e. floppy). Another computing example is RAM. This is the acronym for random access memory, the main memory in computers. It was called "random access" because it supplanted tape memory which was strictly "serial access". If the information you needed was on the other end of the tape, you had to wait for it to rewind; this is why computers in old movies had so many whirling tape drives. In time, the language (and movies) will catch up; assuming computing stops changing so quickly!

c) Software: programming the hardware

As hardware has evolved, so has software. Software is the programs that control the hardware. It is pure information, the stuff of dreams, and seemingly as difficult to control.

The first computers were programmed by physically connecting and reconnecting wires to the components. The reason was that there was no memory to store the programs. Once memory was invented (tape and then RAM), programs were written in binary, in machine language that the processor could execute directly. Each tiny instruction for the processor was laboriously entered by setting toggle switches. Even the simplest programs were very long and tedious to debug; there were no screens or keyboards. Next, card punches and card readers made it possible to type programs, store them on cards, and feed them in.

Programming in machine code is a big headache and unbelievably slow. Before long, some enterprising systems programmer wrote a symbolic assembler that eliminated some of the mind-numbing tedium (computers are excellent for mindless, repetitive tasks!). Assume the add instruction, in machine code, were 2; then to add the number at location 143 you might write: 2143. An assembler allows the programmer to move a little away from machine code, and write "add 143" instead. (By the way, 2, 3, and 4 are not binary digits; the actual binary instruction would have looked more like 0010000101000011.)

Assembly coding is not much fun, and is ridiculously inefficient. Once programmers had good assemblers, they realized they could write compilers. Compilers input some higher level language (like Fortran, or C) and output assembly code. The assembly code then goes into the assembler which emits machine code.

Fortran and C were great advances over assembly code, but before long more powerful languages were invented. There are functional languages, database languages, logic languages and object languages (among others). C++ was an early object language based on C. Java is a later language based on C++; it was designed for programming on the web and eliminates some of the worst shortcomings of C++. There will be many other new languages as time goes on. Perhaps you will design one.

d) Why it's not quite that simple

Perhaps you noticed that the distinction between hardware and software was not quite as clean as it might be. Hardware is stuff you can hold in your hand. Software is information. Plugging and unplugging wires was the first example of software, and wires are definitely physical; their arrangement is logical, but still...

People like to create simple categories. Hardware/software. Us/them. Nature/nurture. But things are often not quite so simple. A hundred years ago a debate raged over how much of what animals (including people) do is determined by genetics and how much by experience. The blank slate faction said it was all experience; the instinct school claimed it was mostly built in; some cooler heads argued it was about 50-50. Now most everyone knows that a better answer is 100-100 -- what we become is determined by our genetic heritage *and* our experiences. To neglect either would be a mistake.

Another common and natural false dichotomy is to divide the world into us and them. There are people who blame "them" for whatever goes wrong. "Them" could be teachers or students, government or citizens, parents or children, our race or theirs, our country or theirs, our religion or theirs. If people are not paying attention they seem to do this automatically; there’s reason to think it might be an innate proclivity. But we can learn not to.

The hardware/software dichotomy is also murky. It is possible to bake programs into silicon; there are hardware Java chips. Processors have microcode inside that governs their function. It is even possible to build virtual machines; software simulations of hardware. Perhaps the most revolutionary aspect of Java is the Java Virtual Machine. This requires a bit of explanation.

e) The Java Virtual Machine

Different computers have different processors. For a program to run on a particular machine, it must first be translated into the machine code of the processor on that machine. Thus, if you have a program that must run on a dozen different CPUs, it must first be translated into a dozen different "binary" files. When a new CPU is invented, the program must be retranslated for that CPU. If you hope to distribute a program across the web, this is a major headache.

This problem is solved in Java by the introduction of an intermediate form, byte-code. Byte code is a machine independent form which runs on the java virtual machine. The Java virtual machine (VM) for a particular machine knows how to interpret byte code and execute instructions on that machine to accomplish what the programmer intended. A Java program is compiled into byte-code and distributed. It can then be run on any machine that has a Java VM installed. The VM must still be written for every type of processor, but it’s a huge improvement to only distribute one program instead of every program.

D. Juggling (!)

I juggle on the first day of my introductory programming courses. I explain that programming is like juggling, you can’t learn to do it by watching. It’s not quite the same, but you can watch me juggle if you go to:
www.willamette.edu/~levenick/juggle/juggle.avi.

I've been juggling for many years and can easily keep three balls in the air for long periods of time without dropping them. I already know how, so it's easy. If you're just trying juggling for the first time, you will drop the balls often. If, each time you miss one, you curse yourself, or the ball, or whoever distracted you, and think to yourself, "I'm just no good at juggling!", you are not helping yourself learn. The appropriate response is, "Oops! Missed!"; then pick up the balls and keep practicing.

Programming is similar in that almost every program has mistakes initially; these are called "bugs". Only novice programmers imagine that any nontrivial program will be right the first time. Experienced programmers make fewer mistakes than neophytes, but, like learning to juggle, when you run into bugs, if you get angry or imagine it reflects on you personally it doesn't help. It's normal to have bugs, especially initially.

i) Learning to program

Programming is also rather like writing in many respects. They both are creative, iterative processes without any one correct way. They both have syntax (grammar) and semantics (meaning), and grammatical errors can obscure the meaning in either. Of course, no one reads programs for pleasure; on the other hand, novels do not deliver email or control automated factories.

There are many different ways to teach Java programming. Some books still start by teaching the old C constructs, and only introduce classes after 6 or 10 weeks. Modern approaches introduce classes earlier. This text begins with classes and adds a cognitive component. A programming environment always includes a programmer; teachers and students of programming ignore the characteristics of programmers, and in particular, novice programmers, at their peril. Good software development methodology minimizes cognitive overhead, and thus leaves the programmer, at whatever level of expertise, with enough cognitive capacity to solve the problems that will inevitably arise -- this is especially important for beginners.

Learning to program is not easy. The first several weeks, before you can do even the simplest things, can be especially frustrating. Fortunately, after five or six weeks, when you have mastered the basics of input, output, classes, and calculation, the ability to make a machine do what you want it to balances out the initial difficulty of doing so.

One of the reasons programming is difficult initially is that there are so many details one must learn before one can construct even the simplest program. There are a number of facts and concepts (a few dozen) that one must grasp before programming begins to make sense -- unfortunately, they are all inter-related, and understanding (or explaining) one, requires understanding (or explaining) all the rest. So, at first, the whole enterprise can seem hopelessly confusing and even daunting. But, with patience, persistence and practice you can surmount this obstacle; and once you learn to program it can be very rewarding and remunerative. If, given these cautions, you wish to continue, read on!

ii) The Approach used here: Less is more; more is less

Computing is different from other fields. First, it is brand new. Mathematics, rhetoric, psychology, physics and philosophy are thousands of years old. The first digital computer (which was as big as a house and less powerful than a modern day low end calculator) was developed in the 1940's. Java, the language this book teaches, was released in 1995 (by Sun Microsystems). It extends the old C++ language which extended the older C language from the 1970's. Computing is evolving at an unheard of pace. Better hardware makes possible better software; better software allows programmers to build even better software; as programmers mature who have been educated in the new paradigms, they can invent even better paradigms. The synergy between faster, cheaper hardware, better programming tools, and better educated programmers will result in a transformation of our lives. And you can quote me.

Second, because computing is so new there is no consensus on how to teach it. In biology, or mathematics, or any of the established disciplines, introductory courses have been taught to millions of students over the past hundred years or so. As a result, these disciplines have well entrenched examples and excellent textbooks, tested on class after class after class. Some Java textbooks, by contrast, are rewritings of C++ textbooks. Others are a hodgepodge of hints and techniques. It's not that computer scientists don't know how to write, it's more that there hasn't been time, and it's not yet clear how to help people learn to program well in Java.

This text takes a different approach, the less is more and more is less approach. Less is more, in that there are many extraneous details of Java that are best avoided in the beginning, so that the student may focus on what really matters. On the other hand, it is imperative that students begin by learning to write classes, even though this is difficult until they grasp a certain set of the basics. So even though it is not easy to learn to write classes, that's where we will start; this seems like more, but in fact, after a few chapters it will result in there being much less to learn later. Let's get to it!

E. Conclusion

This brief (and somewhat opinionated) introduction to programming, computing, culture and human nature has introduced many of the themes that will run through this text. With any luck it has whetted your appetite for an introduction to Java programming which follows immediately.

Chapter 2: Programming: A quick dip in the pool

A. Introduction

This chapter introduces the structure of Java programs through several examples. It also presents an important problem solving technique, namely learning to define problems in such a way that you can come to grips with their essentials instead of becoming hopelessly mired with inessential issues. In this spirit the examples are presented without complete explanations of the intricacies of every component; a thorough presentation will be deferred until Chapter 5. This is done for two reasons. First, it is more interesting to write programs and see them work than to memorize long lists of details. Second, it is much easier to understand the details of a programming language after you have had some experience working with it.

i) Learning to program

Once, a travelling saleswoman was driving down a country road and spied a pair of coveralled legs protruding, unmoving from under a gigantic combine. Concerned for the well-being of this prostrate farmer, she stopped her car, hopped the fence, walked up to him, and asked, "Are you okay?". "Yep, just trying to fix this combine.", he replied. The saleswoman studied the bewildering complexity of gears, levers, hydraulics and wires for a moment, then asked, "How did you learn to fix such a complicated thing?". "Don't know a thing about fixing 'em.", he replied. "Then how on earth do you expect to be able to fix it?". "Well...", drawled the farmer, "...I figure a person made it.".

A programming environment is a little like that combine. It is composed of many interacting parts and at first it certainly looks like you might never figure it out. Surprisingly, programming is easy. The computer always does exactly what you tell it to. Programming is also fun, once you get the hang of it. Learning to program may seem a daunting task. To the uninitiated, the complexity may appear overwhelming, incomprehensible. Many interacting parts must be in the proper relationship before even the simplest program will work.

Like that combine, the Java language is large and complex, but, again like the combine, it is very useful once you know how to use it. The old C programming language evolved into the object based language C++ and then into Java. As a result, Java has some complicated syntax. If a novice programmer were required to understand everything in a first program, it might take weeks or months before that program could be written. But, this book uses a spiral approach. First the essentials of Java are introduced by way of examples. Those concepts will then be revisited in more detail in Chapter 5; this will provide the understanding to accomplish many programming projects.

Learning to program in Java, like fixing a combine, *is* possible. Learning to control the information processing machine we call a computer is initially a lot of work, but after surmounting that initial obstacle it can also be exciting and rewarding.

ii) Problem Solving

"There are only two kinds of proofs in mathematics - trivial proofs and impossible proofs. The difference is that the trivial ones you already understand." *Freshman Calculus Prof.*

The author of this text is an experienced programmer. He has written large, complex software in numerous programming languages. But, when he first uses a new programming language he writes a tiny program that displays the single word, "greetings", on the screen. Why is this? The answer is simple, but difficult for some to learn. Do you know (without looking ahead) what it is? If you do, you already know one of the most important lessons of ***problem solving***.

a) Definition of problem solving

Life is full of problems. You are already an expert problem solver, otherwise you would not be where you are now, and you would not be reading this book. Problem solving (as used in this context) is different from ordinary functioning or coping. Every day you find your way home to sleep. Sometimes your car breaks, or your bike is stolen, or you’ve sprained your ankle, or there is a traffic jam, or road construction, or a train blocks the way. You deal with those problems and get home anyway, but that’s mostly just coping. Problem solving is what you do when you are stuck; when you’re stopped and don’t know what to do next. That’s never a pleasant position to find yourself in, but if you develop your problem solving skills, it is a lot more comfortable (and less common!).

b) Details

Here’s a fact that a novice programmer might not know: When you first learn to program you get stuck frequently. Why? Because programming requires familiarity with a large number of details. Worse, when you are starting out, since you don’t really understand what you are doing, that mass of details doesn’t really mean anything, so there’s no way to remember them all. Later, as you become familiar with the process, and develop good programming habits, you’ll deal with many of those details automatically without having to think about them. But initially, even though the individual details are small and not especially interesting, every one of them must be correct for your program to work.

It’s a little like when you move to a new city. At first you only know where a few places are, and getting anywhere is a mystery and takes concentration. Then you learn a few places and routes, but anytime you get off them, you are lost. In time, it becomes familiar and you don’t even have to think about how to get places. But at first it’s all new and confusing. Programming too. But it *does* improve.

c) Bugs

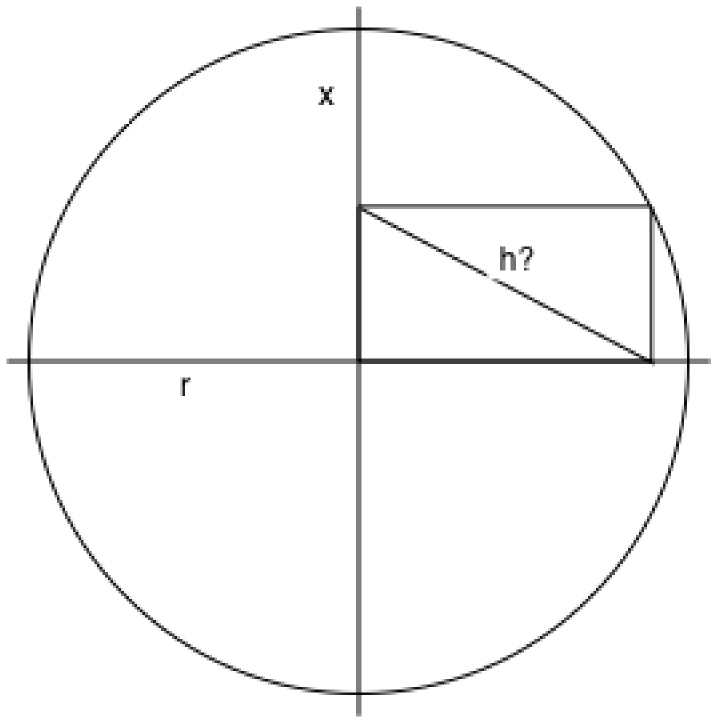
Errors that occur when a program runs are called ***bugs***. There’s a story that one early computer didn’t work for days until someone discovered a dead moth between a contact. The story may be apocryphal, but the term is here to stay. An accomplished programmer has much experience solving the problems that arise in the course of programming. In other words, part of learning to program is learning to deal with bugs; both finding and fixing them (and the former is sometimes harder than the latter!).

d) A tool kit for solving problems

This text will include a number of problem solving techniques. Some of these will be specific to programing in Java, but the majority will be more generic. Think of these techniques as tools. If you learn these techniques, then when you confront problems, both programming and otherwise, you may feel better prepared. As if you’ve acquired a toolbox of useful problem solving methods. Instead of discovering yourself stuck and feeling like you don’t know what to do, you may find yourself saying, "Huh! Stuck!", and then immediately start to consider which of the various techniques in your toolbox is most likely to be helpful.

e) A problem solving exercise

Almost everyone has been exposed to the Pythagorean Theorem: in a right triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides. Consider the following diagram as a circle centered at the origin, together with a rectangle inscribed in the upper right quadrant. The radius of the circle is r. The height of the rectangle is y. The question is, what is h? If you knew the width of the rectangle you could use the Pythagorean Theorem, but all you know is the radius, r. Try to figure out h; but don’t spend more than a few minutes on it. The answer is later on in the chapter.



iii) The two types of Java programs: Applications and Applets

Java programs are either Applets or Applications. Each type of program has unique capabilities.

a) Applets

`Applets` are programs that can run in web browsers. They are commonly used to add functionality to web pages. When you use the web, if you pay attention, you will sometimes see the message "Starting Java" in the lower left corner of the screen. It is very convenient to be able to execute Java Applets from web pages; indeed, Applets were designed for this purpose. But, there is a problem. When you go to someone's page and an Applet starts running, it is running on your machine. If there were no safeguards, this program, that you might not even know is running, could be damaging your computing system, or stealing all the information from your disk. To avoid this possibility (and thus to convince people to allow Applets to start up in browsers at all!) Java has a Security

Manager, that limits what Applets are allowed to do. The most important things that Applets cannot do are reading and writing files on your local computer, and opening ***sockets*** to communicate with other machines.

b) Applications

Applications run independently; they do not depend on a browser (although they do require a ***Java Virtual Machine*** to be installed). They cannot be embedded in a web page. But, they are not prohibited from file I/O, or opening Sockets; they are full-fledged programs.

c) Components of a Java program

i) A high level typology

What are Java programs made of? Abstractly, a program is a series of symbols that conforms to the grammatical rules (also called ***syntax***) of a particular programming language.

One of the big advantages of object oriented programming is that code is reusable. ***Packages*** allow you to easily import classes that other people have written into your program. These classes include input/output mechanisms (to move information into and out of programs), mathematical functions, screen control and many other things.

Class declarations include both memory declarations (which declare variables to store information) and method declarations. Method declarations explicate how members of a class respond to the messages of the same name. A useful metaphor is to think of the objects in a program as actors, and methods are the ***scripts*** the actors will read when they get cues (messages). The ***main method*** is the main script for an Application.

ii) A template for every Java main method

Every Java Application must have a method named "main". This is the method that will be executed when the program runs. The main method must have the following structure:

Code Example 2.1

```
public static void main (String [] args) {
    General Declarations
    Statements
}
```

Pattern for every main method.

When your program runs, it executes each line in the main method from the top down. The general declarations may create objects and the statements may send them messages. After the last statement is executed and the closing } is reached, the main method, and thus the Application, is terminated.

B. Examples:

i) The first example of a class - A Robot Greeter

Imagine you have decided to build a robot. Before going to all the work and expense of buying and assembling the motors and gears and whatnot you wisely choose to simulate a prototype first. You decide to write a Robot class which (recalling the problem solving advice above) initially does nothing but say hello.

Code Example 2.2

```
public class Greeter {
    public static void main (String [] args) {
        System.out.println("Greetings!");
    }
}
```

Greeter class to display "Greetings!".

This program has only one class, named Greeter, with only the main method whose heading reads: `public static void main(String [] args)`

That’s a mouthful. Think of it as an incantation for now; like "abracadabra!". Additionally, like any language, the Java programming language contains a number of ***idioms***, sequences of symbols whose meaning cannot be derived from the individual symbols, but which must be learned as a whole, by rote.

This program is not particularly useful. The only reason for it is to help you learn how to define classes in a simple setting where you can easily focus your attention on the essentials. (This is the reason, if you hadn’t guessed, why experienced programmers write a greetings program first!).

Problem Solving Technique

Create a prototype before attempting any difficult project.

If there are too many details, too many parts, too many things to keep in mind, sometimes making the problem smaller or simpler will allow you to solve it and then adapt that solution to the larger problem.

The next box explains the code in this first example.

From: Code Example 2.2 on page 22

```
1 public class Greeter {
2     public static void main (String [] args) {
3         System.out.println("Greetings!");
4     }
5 }
```

This is the Greeter class declaration.

Line 1: Every class declaration begins with the word class. After "class" comes an identifier, the name of the class being declared. ***`identifiers`*** must begin with a letter and be composed only of letters, numbers and underscores. The name of the class is Greeter.

Lines 1-5: Next comes a pair of curly brackets containing a list of the memory and methods of this class. Here there is no memory and only one method, named main. Every Application you write must have a main method, and that method must have:

Line 2: `public static void main (String args [])` is the heading of the main method. For now consider this an idiom or an incantation.

Line 3: The "body", or script, of the main method follows that heading. Like the script of every method, it is enclosed in {}s. The main script for a Greeter consists of the single line: `System.out.println("Greetings!");` This is an idiom meaning display the literal string of characters "Greetings!" on the screen.

a) Running your program

Run this program before proceeding. If you are using this book in the context of a class your instructor will likely have explained by now how to do that. If you are using it in another context, now is the time to consult with an expert, or take the time to do the tutorial, or read the documentation for the Java environment you are using. Assuming you are using NetBeans, consult NetbeansAppendix A on page 327 for instructions.

If you have never programmed, once you have compiled and run this tiny program, you will have taken a big first step. Before long it should take a minute to do. Unfortunately there are many little things that you must learn, so the first time may take much longer. It should not take more than an hour. If you spend a whole hour without success it is likely a wise use of your time and energy to seek out someone to assist you.

Speaking of solving puzzles, here’s a clue; the two diagonals of a rectangle are equal. Now do you know what h is (in "A problem solving exercise" on page 19)? Sorry about the Pythagorean misdirection; the point was that how you look at a problem may determine how difficult it is to solve. If you were determined to calculate the width of the rectangle, you were doomed from the start. This is an instance of the following:

Problem Solving Technique

Adopt a different perspective.

How difficult a problem is depends on how you look at it. A problem that seems impossible from one angle sometimes is trivial from another.

b) Mechanics: Typing and running your code

Whatever system you are using there are three steps to running a program:

1. creating source files - the Java code in the shaded boxes is called source code, when it is stored in a file that is called a source file. Netbeans writes the shell of your classes for you. Since the source code declares a ***`public`*** class called Greeter, Java insists that it be stored in a file called Greeter.java. Also, you must remember that capitalization matters! If you type `PUBLIC static void main`, it will object. If you type `(string [] args)` it will not know what `string` is; it only understands `String`.
2. invoking the compiler -- before your program can run, it must be converted into ***`byte code`***; that is the task of the compiler after it checks the syntax.
3. executing the program. -- if your code compiles without errors you can then execute it. When you choose Project/Execute, Netbeans compiles and executes your code.

If you have not done so yet, adjourn to the keyboard to create and run this robot greeter program. Then, mysteriously you will have crossed the divide and can be labelled a "programmer" -- writing (or even modifying) a class is something 99% of the people on this planet have never done!. If your program had/has many errors you might turn to the "What could go wrong" section in the back of the chapter for hints.

ii) The second example of a class - a personalized RobotGreeter

This section presents a class that is able to say "Greetings, Spike.", or "Greetings, Buffy.", or "Greetings, whomever.", depending on who you ask it to greet.

Most Java Applications have a class with a main method and little else. The main method creates one or more objects and sends them messages; the work of the program is done by these objects and classes. This example, and the rest in this text, will have a class which only exists to interface with the outside world.

a) A prototype

As you may recall, if a system is complex, build a ***prototype*** first; after that works, then (and only then!) elaborate it. First I will present a prototype with the desired structure that always says "Greetings" (just like the last example).

Code Example 2.3

```java
class PersonalGreeter {
    public void sayHi() {
        System.out.println("Greetings!");
    }
}

public class RobotGreeter {
   public static void main (String [] args) {
       PersonalGreeter myGreeter = new PersonalGreeter();
       myGreeter.sayHi();
   }
}
```

Prototype personalized RobotGreeter Class with main driver program.

Notice that the PersonalGreeter sayHi() method does exactly what the old main() method did, namely displays Greetings! on the screen by using the idiom:
`System.out.println("Greetings!");`

From: Code Example 2.3 on page 26

```
1 public class PersonalGreeter {
2     public void sayHi() {
3         System.out.println("Greetings!");
4     }
5 }
```

This is the PersonalGreeter class declaration.
Line 1: Notice that identifiers consisting of more than one word have the first letter of the second and subsequent words capitalized.
Lines 1-5: Next comes a ***block*** (a pair of curly brackets and what's between them). Notice how similar it is to the Greeter class.
Line 2: Begins the sayHi method. Unlike the main method, here there is nothing in the ()'s.
Line 3: The "body", or script, of the sayHi method follows this heading. Again, like the script of every method, it is enclosed in {}'s.

.

From: Code Example 2.3 on page 26

```
6
7  public class Greeter {
8      public static void main (String [] args) {
9          PersonalGreeter myGreeter = new PersonalGreeter();
10         myGreeter.sayHi();
11     }
12 }
```

This is the Greeter class declaration.
Line 9: creates an object which is an instance of the PersonalGreeter class (this is called instantiation) and stores it in the variable myGreeter. Once it is created it can be sent messages. Notice that the first character, 'm', is lowercase; the convention is that classes have uppercase first letters, objects, lowercase -- if you follow this convention, you can tell right away from the name of a thing whether it is an class or an object.
Lines 10: sends the `sayHi()` message to the object named myGreeter. Recall that information processing in Java is accomplished by sending messages to objects.You can tell that messageName() is a message by the ()s.

This prototype can only respond to one message. And its response is always to say "Greetings! ". This program is useful to convince yourself that you can write a class and make it do something (you did type in that program and test it didn't you? If not, it's time to go to the screen and modify your previous program, just change it so that it is like the above. Remember, programming is like juggling; can't really learn by watching, can't really learn by reading. Plus, aren't you bored with reading these details?).

b) Elaborating the prototype personalized greeter

Welcome back. Finally, let's elaborate our prototype to make it capable of greeting various people. The question, as always, is, "How best to accomplish that? ". Usually there is more than one way to approach a program. We could add a bunch of methods like sayHiToSpike() and sayHiToBuffy(), but with the use of ***parameters*** we could add a single method that could greet whoever we wanted it to. This example does just that. It is an extension of the previous. The changes are in **bold**. The rest is unchanged.

Code Example 2.4

```
1  public class PersonalGreeter {
2      public void sayHi(String who) {
3          System.out.println("Greetings " + who + "!");
4      }
5  }
6
7  class Greeter {
8     public static void main (String [] args) {
9          PersonalGreeter myGreeter = new PersonalGreeter();
10        myGreeter.sayHi("Spike ");
11        myGreeter.sayHi("Buffy ");
12     }
13 }
```

Personalized RobotGreeter class and driver

Line 7: There is no public in front of class. Java only allows one public class per file.

Free advice: although the ***access modifiers***, public, private and protected are valuble in complex programs, if you simply omit them, nothing will go wrong at this stage. Feel free to leave them out for now (assuming your instructor is okay with that!).

The two lines that are changed in the main method both send the PersonalGreeter object named myGreeter the SayHi message, first with the parameter `"Spike "` and then with the parameter `"Buffy"`. When the parameter is `"Spike"` this message causes myGreeter to display "Greetings Spike! " and when the parameter is `"Buffy"`, it displays "Greetings Buffy! ". The next section will explain Strings and parameter passing, for now it is enough to remember that things in parentheses after message names are parameters.

Quick question: How could you modify this program to greet other people?

c) The String type.

You will often work with literal strings of characters, like "Java", or "Hello world!". There is a class in Java called ***`String`***, which is designed for just that (Notice the capital S' in String; you remember what that means? Right, that String is the name of a class).

d) Details

Perhaps you are not in the mood to take on a bunch more detail, but would rather come back and grapple with it later. That would be fine. If you'd rather skip ahead to run this program first, that would be fine (it's at "Running the personalized robot greeter" on page 31).

From: Code Example 2.4 on page 28

```
1 public class PersonalGreeter {
2    public void sayHi(String who) {
3        System.out.println("Greetings " + who + "!");
4    }
```

The sayHi(String) method in the PersonalGreeter class.

The heading includes the access type (public), a return type (void), the name of the method (sayHi), and a parameter (String who) enclosed in parentheses

Line 2: Says that RobotGreeter objects will respond to a sayHi message that has a String parameter.

Line 3: The body of the sayHi method is a single message, `println(String)` which is sent to `System.out` (the screen). There is one parameter with three parts:

1. the `String "Greetings "`
2. the value of the parameter named who
3. the `String "!"`;

Java treats the plus sign after a String as the ***`concatenation`*** operator, so it pastes those three things together to make a single String, namely, `"Greetings` ????!", where the ???? is the value of the String parameter was passed with the sayHi(String) message.

e) Using parameters to pass information to a method.

Every time a RobotGreeter follows this script it will display "Greetings" and "!". But, the value of the parameter, who, may be different each time the script is followed. "Greetings" and "!" are inside double quotes and so are String literals, they will be displayed literally. In contrast, who, is not in double quotes, so it will not display the letters 'w'-'h'-'o', but rather the current value of the parameter named who. The value of who will be whatever string was in the parentheses where the `sayHi(String)` message was sent to a RobotGreeter object. There is no way to tell what that value might be without looking there.

From: Code Example 2.4 on page 28

```
5 public class Greeter {
6     public static void main (String [] args) {
7         PersonalGreeter myGreeter = new PersonalGreeter();
8          myGreeter.sayHi("Spike ");
9          myGreeter.sayHi("Buffy ");
10    }
11 }
```

The Greeter class that creates (instantiates) a PersonalGreeter and sends it sayHi(String) twice.

Line 7: Creates a new PersonalGreeter and stores it in the variable called myGreeter.
Line 8: Sends the `sayHi(String)` message to myGreeter with the String parameter "Spike". To repeat, this will cause the myGreeter object to follow (or, in technical terms, "execute") the `sayHi(String)` method (script), using "Spike" as the value of the String parameter named who. Thus, inside the `sayHi(String) method` in PersonalGreeter, "Greetings " + who + "!" will turn into "Greetings Spike!", and be sent to `System.out` along with the `println(String)` message, and thus will end up on the screen. Many small steps. The good news is, the computer carries them out tirelessly.
Line 9: Exactly the same, but with the parameter value "Buffy".
Line 8 was explained very slowly; ordinarily one might say instead: Line 9: Sends the sayHi message to myGreeter with the parameter "Spike". And, to an experienced Java programmer, `myGreeter.sayHi("Spike")` means exactly that, and requires no explanation at all!

f) A Digression

If you have never programmed before, there are a number of new concepts in the previous example. If you did not understand it, read it again. Type it in. Get it to work. Now, read it again. Hopefully it will make more sense. If it still doesn't,**DON'T PANIC**! There are many concepts in any discipline that cannot be grasped until you have had a certain amount of experience. Let me tell you a story. When I am trying to learn a new computer system or language I usually read the entire manual or language description; it normally only takes 4 or 5 hours. Since I know little or nothing about how to use the new system or language, I can't understand much of what I read; thus, whenever I lose the thread, can't understand what I'm reading, I just skip to the next section. Then I go and play with the system or language for a few days or a few weeks, inventing little problems to solve. This inevitably leads to numerous unsolved mysteries and frustrations. Then, I reread the manual. This second reading is often very illuminating as now I know something of the system and have a number of questions I want to answer.

Learning to program takes time. Plus, a complete explanation of parameters is in Chapter 5. For now, patient reader, please reserve your reservations and proceed on the assumption that everything will become clearer shortly.

g) Running the personalized robot greeter

Before reading on, go now to a computer. Input and run this example (or quicker, modify the Greeter class, but remember if your class is called Foo, then it must be stored in a file called Foo.java). In NetBeans, you can change the name of the class by right-clicking on it in the Filesystems pane (on the left) and selecting Rename.

After that works, insert your name and the names of several friends for the robot to greet, compile and run it again.

iii) Third example: A minimal Applet

a) The power of inheritance

The second kind of Java program is an Applet. Before writing a RobotGreeter Applet we will try out the simplest possible Applet (you remember, first write a prototype).

Code Example 2.5

```
1 public class RobotGreeter extends java.applet.Applet {}
```

A minimal Applet

Line 1: A class declaration. It makes RobotGreeter extend java.applet.Applet; this means a RobotGreeter can do anything a java.applet.Applet can. Since there is nothing between the {}s, the RobotGreeter class does not add anything to, or change anything about the java.applet.Applet class; it is essentially another name for that class.

You might well be asking yourself if this class that appears to do nothing can run; or what it might do if it runs. Try it and see!

b) Executing an Applet

To run an Applet (yes, run is a synonym for execute!) requires four steps: create the source code, compile the source code to make a .class file, create HTML code to start the Applet, open the HTML code from a Java enabled browser. Fortunately Netbeans handles these details for you. See NetbeansAppendix B on page 329.

c) A greetings Applet that uses a graphics context

Now for an Applet version of a robot greeter. There are a number of ways that one might make an Applet that printed "Greetings!". This particular method uses paint(Graphics), which is the standard method for drawing in a Component, and you'll be seeing it regularly later.

Code Example 2.6

```
1 public class FirstApplet extends java.applet.Applet {
2
3  /** Initialization method that will be called after the applet is  loaded
4      *  into the browser.
5     */
6    public void init() {
7    }
8
9    public void paint(java.awt.Graphics g) {
10      g.drawString("Greetings!", 0, 100);
11   }
12 }
```

An Applet that writes Greetings! on the screen.

Lines 9-11: The only change to the prototype is the method, public void paint(Graphics); this is automatically invoked when the Applet starts.

Line 9: The method header, think of this as "abracadabra".

Line 10: The only statement in paint, draws the String "Greetings!" on the screen at location 0, 100; that's column 0 row 100 (measured in pixels). Try some other numbers.

Proceed to the screen and try this one out.

C. Conclusion

The first program that experienced programmers learning a new programming language write is one that simply says "Greetings", or in the older C tradition, "Hello world". This is because expert programmers have learned from painful experience that if you try to do complicated things before simple things, you can waste more hours than you might initially believe.

By writing a program that merely writes a word or two on the screen you either encounter all the difficulties of a new programming environment in the simplest possible context, where you will know to focus your attention on the environment instead of the program; or, you will discover that the programming environment is easy to use and go on to more difficult tasks immediately.

Either way, this is an example of an important problem solving (or in this case problem reducing) technique -- build a prototype first!

This chapter presented the two types of Java greetings programs and explained roughly how they were constructed. Although numerous details were presented, many others were glossed over, or explicitly ignored. The next two chapters will address some of those skipped details as well as introduce more involved examples.

D. End of chapter material

i) Problem Solving Techniques

Problem Solving Technique

Create a prototype before attempting any difficult project.

Problem Solving Technique

Adopt a different perspective.

ii) What could go wrong?

Any number of things! When you first program, the details are legion and any one can trip you if you forget it. The good news is that ignorance can be cured.

Problem 2.1 -- I don’t have access to a computer.
Possible causes: Many.
Possible solutions: Gain access.

Problem 2.2 -- Compiler error messages. There are many ways to generate syntax errors. Missing a single keystroke will generate an error and can generate multiple errors. Capitalization errors can be very difficult to find. Here are a few examples from:

```
examples/Greeter.java [21:1] not a statement
        System.out.println"Greetings!");
                  ^
```

```
examples/Greeter.java [21:1] ';' expected
        System.out.println"Greetings!");
                          ^
2 errors
Errors compiling Greeter.
```

The errors here come from omitting the (after println.

```
examples/Greeter.java [11:1] 'class' or 'interface' expected
publc class Greeter {
^
1 error
Errors compiling Greeter.
```

The error here comes from omitting the i in public.

```
examples/Greeter.java [20:1] cannot resolve symbol
symbol  : class string
location: class Greeter
    public static void main (string args[]) {
                             ^
1 error
Errors compiling Greeter.
```

The error here comes from typing an s instead of an S in String.

iii) New terms in this chapter

Applet - a Java program that runs in the context of a web browser. Also a class in java.applet. 20

Application - A java program that runs independently. 21

bug - An error in a program. 19

byte code - The intermediate form that the Java compiler puts into the .class file to be interpreted by the Java Virtual Machine when the program executes. 25

Component - a generic class in java.awt that includes Button, TextField and other common GUI components. 33

concatenation - to attach together end to end. If you concatenate "psycho", "the", and "rapist", you get "psychotherapist". 29

identifiers - Java names. Must start with a letter and be composed of only letters, digits and underscores; case matters. 24

idiom - a sequence of symbols whose meaning cannot be derived from the individual symbols, but which must be learned as a whole, by rote 23

Java Virtual Machine - Software that creates the Java environment on a particular machine. 21

main method - where execution begins when your Application runs 21

Package - A group of methods arranged so you can easily import classes that other people have written into your program 21
parameter - information sent along with a message which invokes a method 28
problem solving -- The behavior one engages in when one is stuck and doesn't know what to do next. 18
prototype - a simplified, preliminary version of something, in this case, a program. 26
public - An access type; means anyone can see this. 25
Socket - A mechanism to connect one computer to another (virtually). Also a class in java.network. 21
String - the Java class whose instances are each a literal series of characters. 29
syntax - Grammar. Every programming construct has both syntax (grammar) and semantics (meaning). 21

iv) Review questions

2.1 What are the two types of Java programs?
2.2 What's the difference between them?
2.3 How (in the context of a Java program) do you print a message on the screen?
2.4 How is information processing accomplished in Java?
2.5 What are {}s used for?
2.6 When do experienced programmers write the greetings program?
2.7 What problem solving technique are they using when they do?
2.8 What are parameters used for?
2.9 Why is the S in String a capital letter?
2.10 What is `public static void main(String [] args)?`

v) Exercises

2.1 Add these lines to your program and see what they print.

```
System.out.println("2+2");
System.out.println(2+2);
System.out.println(2+2 == 4);
System.out.println(2 > 3);
System.out.println(2 < 3);
System.out.println("backslash-t, i.e. \t is a tab\ttab\ttab");
System.out.println("backslash-n, i.e. \n is a newline\nnewline\nnewline");
```

2.2 Now try these

```
int x=17;
System.out.println(x);
System.out.println("x=" + x);
System.out.println("Which of those outputs was easier to understand?");
System.out.println(x == 17);
x = x + 1;
System.out.println("after adding 1, x=" + x);
System.out.print("is x still 17?  No.  See? When" +
       " it compares them it says ");
System.out.println(x == 17);
```

2.3 Modify the paint method in your Applet by adding these lines.

```
    g.drawOval(200,200,100,100);
    g.fillOval(400, 300, 75, 288);
    g.setColor(java.awt.Color.RED);
    g.drawRect(27, 27, 300, 10);
    g.setColor(java.awt.Color.GREEN);
    g.drawLine(30, 300, 400, 20);
```

It is likely you will need to enlarge the Applet window before you will see them all.

2.4 Now modify paint to draw a simple picture, instead of random stuff like in the previous question. Draw a simple house. Then try your house. Then the front of the building your class meets in.

Chapter 3: Class design and implementation

A. Introduction

This chapter illustrates the process of class design, coding and testing. It starts with a description of a simple ATM simulator and concludes with a working Applet that implements it. The program is grown in stages, starting with a very simple prototype and adding features only after each prototype works. The resulting Account class will be incorporated into a larger bank database system in Chapter 11.

i) A description of the task

Imagine you are given this description for a programming assignment: Write a minimal ATM program that will manage 3 bank accounts. Each account will have a name and a balance. Allow users to display their current balances and withdraw as much (simulated) money as they want with a graphical user interface (***GUI***).

ii) Before beginning to program: Design!

There is a strong impulse to start to program too soon. When given a problem description, some beginning programmers start typing before they have a clear idea of what it is they are doing. In a way this is unavoidable, since a novice programmer knows practically nothing about the programming language. Nevertheless, it is possible to have a clear idea of *what* one is attempting to accomplish even if the *how* is a bit unclear.

It is human nature to experience confusion as pain. Experienced programmers have learned not to start typing before they have a clear understanding of what it is they are attempting. Painful, frustrating experiences have taught them to think through problems before committing to code. Therefore, between the time they read or formulate a description of a programming task, and when they sit down to type code, they do what is called ***design***. Design can take many forms, but it always features clear, simple thinking. A coherent design supports the programming process, and can make the difference between success and failure.

One design technique is to start with the user interface; decide what the user will see and what actions the user will be provided; then design classes that support those actions.

a) The user interface

Start by drawing a picture of what the user will see when the program runs. For simplicity, start with a single bank account. To design a good user interface you must consider what information will be presented to the user and what actions the user will take to interact with your program. What are those actions and information in this case?

If you reread the description (do that!) you will see that the user can do just two things: ask for the current balance, and withdraw funds. For a withdrawal the user must specify how much money they want and that information must be input to the program. To display their balance they only need indicate that they wish to see it, so a button press will be sufficient, and then information must be output to the user.

Knowing that, what sort of user interface does your ATM need? At minimum, it will need: 1) a button to request the current balance; and, 2) somewhere to type the amount to withdraw. The Java Components for these two screen objects are Button and TextField. You are probably familiar with both of these components, they appear on many web pages. When you fill out a form on the web, the boxes you type in are typically TextFields, and the buttons, Buttons. Grab a piece of paper and make a quick sketch of what your user will see when your simulated ATM runs. Note that you can do this without *any* knowledge of how the program will work.

b) What classes will we need? What will they do?

A principle of object design is to create classes that correspond to the things that the program is modelling, and to store information in the objects corresponding to where it resides in the world. A real ATM machine communicates with a bank; user information is kept in accounts. So, it is reasonable to think of having an Account class, and to store information about each person's account in a separate Account object. Note the use of capital letters at the beginning of class names; this usage is to both remind you that class names begin with capitals and also help make clear when words refer to classes (as opposed to objects in the world).

So, to simulate an ATM associated with a bank that has three customers we will need three Accounts (one to keep track of each customer's name and balance), a Bank (to contain the three Accounts), and an Applet (to handle the GUI). That would be a lot of code to write all at once, so those three tasks will be attempted one at a time and then combined. This is an example of the following:

Problem Solving Technique

Stepwise refinement

First, understand the problem, then break it into 5+/- 2 subproblems. For each, if it is trivial, solve it, if it is not trivial, solve it by using stepwise refinement

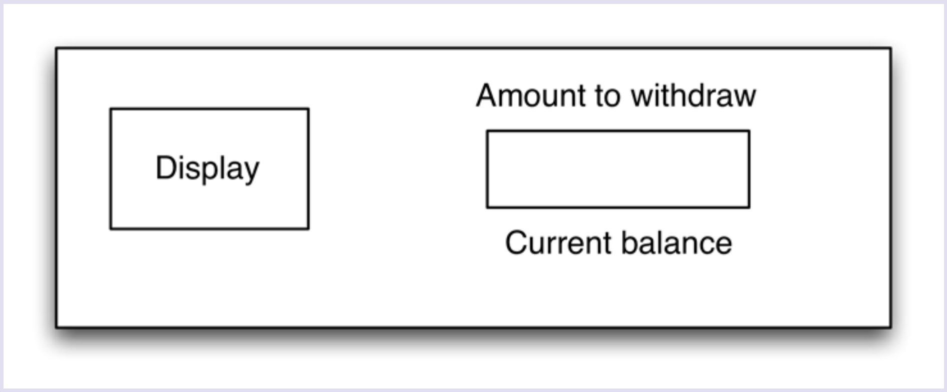
The first step is the most important - until you clearly understand a problem, any attempt to solve it is very unlikely to succeed. Sometimes people attempt to solve problems before understanding them; mostly they fail and are frustrated.

Notice that this is a recursive definition, the thing being defined is used as part of its own definition. Circular definitions are bad. Recursive definitions can be very good, so long as they do not recurse forever. Since the subproblems are smaller than the original, as this recurses, eventually they are all so small as to be trivial and the recursion stops.

B. Building and testing the prototype GUI

Figure 3.3 is a rough sketch of a possible GUI for an ATM that keeps track of the balance of a single bank account. There is a Button with the label "Display" and a TextField with the text "Amount to withdraw" above it, and "Current balance" below it. The TextField will be used to both enter the amount to withdraw and display the current balance.

Figure 3.1



Rough sketch of GUI for the ATM.

The next sections will explain how to build and test a GUI with those two components using Netbeans.

i) Getting Started

1. Start Netbeans and create a new project with a GUI Applet (see NetbeansAppendix C on page 331) for details.
2. Add a Button with an action and test it (NetbeansAppendix D on page 331).
3. Add a TextField to type the withdrawal amount in and display the balance (see NetbeansAppendix G on page 337).

ii) Using the Button to alter the TextField

When the user presses the Button, its ActionPerformed method will be executed. Netbeans writes the shell of the method; you, the programmer, must insert code to make it do what you want when the button is pushed. Add this line of code after line 27 in Code Example 3 on page 332

```
    textField1.setText("Greetings!");
```

Execute the modified program, it should make the TextField say Greetings when you press the button.

iii) Simulating one bank account by hand (without writing the Account class)

You know how to get the text from a TextField (`getText()`), how to set the text in a TextField (`setText(String)`), and how to get control when a Button is pushed. Before going on to writing classes, let’s experiment with a simple ATM with just one bank account balance stored in the Applet. You must do the following three things:

1. Create a variable to contain the current balance
2. When the user types an amount to withdraw and hits enter, get the withdrawal amount into the program as a number
3. Subtract it from the balance and display the new balance

These will be explained next; after some practice the explanations will make more sense.

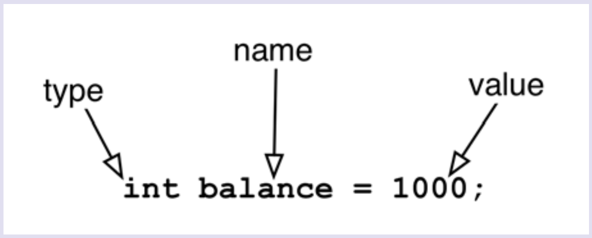
a) Create a variable to contain the current balance

Assume the bank account starts with 1000 dollars. The program must store the value 1000, and then decrease it when money is withdrawn. To store information (like 1000) a Java program uses what is called a ***`variable`***. A variable must be declared, with a name and a type and a value, like this:

```
        int balance = 1000;
```

This declares a variable whose type is int (i.e. like an integer, it stores one whole number) and sets its initial value to 1000 (see Figure 3.2). Add that line of code to your test

Figure 3.2



An assignment statement, illustrated.

Applet, outside of any method, but inside the class. You may be asking, "Where is inside the class?". The class definition is everything between the {}’s following the name of the class. Now, you might ask, "Where is outside any method?"; this will be explained soon, for now before line 3 in NetbeansAppendix 3 on page 332 would do.

b) When the user hits enter, get the withdrawal amount

Change the body of `textField1ActionPerformed` by adding these lines:

```
        int withdrawal = Integer.parseInt(textField1.getText());
        System.out.println("withdraw=" + withdrawal);
```

The first gets the text from the TextField (which has type String), converts it to a number (`Integer.parseInt()` does that) and stores it in a variable called withdrawal. The second prints that value, so you can see if it worked. Try it out now (don't forget to type a number in the TextField! What happens if you type anything else?).

c) Subtract it from the balance and display the new balance

Add two more lines (after the two you just added):

```
        balance = balance - withdrawal;
        System.out.println("new balance is: " + balance);
```

The first subtracts the withdrawal from the balance and store the remainder back in balance. The second, um, prints the new balance. Execute this code. Try hitting enter more than once.

Congratulations! There are three things that do all the work in computing: input, assignment and output; you have just used all three!   Now, on to creating classes.

C. A generic problem solving technique

Let us take a slight detour for a problem solving technique that will be useful in thinking about the Account class. Try to answer this word problem (even if you don't like algebra).

Mary is twice as old as John. Two years ago, she was 3 years older than he will be next year. How old is John?

Maybe you can read that, see through it, and just know the answer. Maybe you have learned a technique called "guess and check" where you try out ages and adjust them in the right direction until you stumble on the answer. Maybe you know how to write equations to compute the answer. The first technique is a great one, if it works. Guess and check finds the answer, but does not give you any insight into the problem. Writing equations and solving them is a more general and useful method. But not everyone can do this sort of algebra problem. Here's a generic problem solving technique that may allow you to solve such problems (and which has direct relevance to designing object programs).

Problem Solving Technique

What are the things? What are their relationships?

By answering these two questions you concretize your conception of the problem. Just doing that may make solving it easier because it will cause you to think carefully about it. In the context of object design, the things are potential classes, the relationships, potential messages.

In the problem at hand, at first glance there are several things, Mary, John, and their ages. But, if you think about it, you will soon realize that Mary and John don't matter, only their ages. For brevity, call their ages M and J. And not only are their current ages of interest, but also Mary's age two years ago (M-2) and John's next year (J+1). Those are the *things* of interest.

The two statements before the question state two relationships. They can be rewritten as: (*Mary is twice as old as John.)* M = 2*J; and, (*Two years ago, she was 3 years older than he will be next year.)* M-2 = (J+1)+3. The latter simplifies to M = J+6, and you can then substitute 2*J for M to find the answer.

D. Account class: Design, implementation and testing

Your program must manage three bank accounts. It would be natural to create three bank account objects, where each would correspond to and hold the information for one account. To be able to create objects (like bank account objects) you must first define a class. The class will then manufacture that kind of object when you tell it to.

i) Account class design

Let's apply the "things and relationships" technique to the ATM problem. The things in this problem are bank accounts, plus the names and balances of each account; the relationships between those things is that each account has a name and balance associated with it. Each account will need a way to store those two pieces of information.

Along with the information to be stored in each account, we must also consider what actions will be performed on, or using, that information. There are only two: display, and withdraw. In the former the balance (and perhaps the name) must be displayed, in the latter the balance must be reduced when money is withdrawn.

The details of how to store the information in each account and how to access and change it will be covered in the next section.

ii) Converting the design to Java code

A class definition includes declarations of variables (where information is stored) and methods (which operate on that information).

a) Variables (state)

Variables encode state. This usage of "state" is very similar to a common English usage. If someone says to you, "What is the state of your bank account?", they mean, how much money is in it. Here are some facts about variables that you will need to know later:

1. Variables are containers for information. The purpose of variables is to hold information. That information may be of various types; numbers, letters, words, or even objects. Java variables are similar to, but different from, variables in algebra. Like in mathematics, Java variables might contain any number of values. The particular values determine the state of the computation (see "State and its representation" on page 2).
2. There is only one way to change the information in a variable, which is to execute an ***`assignment statement`***. In algebra, you might encounter a formula like x = y * 2, and be asked, if y=2, then what is x? In Java, by contrast, `x = y * 2`; means take the value of y, multiply it by 2, and store that value in the variable x (or *assign* that value to x). So, if the value of y were 17, and that assignment statement were executed, afterwards the value of x would be 34.
3. Variables hold exactly one value at a time. Whatever value had been in x before the assignment statement was executed is irretrievably lost. If you wanted to keep the old value, you would need another variable to hold it.
4. Every variable has three attributes: name, type and value. Java is what is called a typed language. In addition to a name and a value, every variable in Java has a type.

For now we will only consider variables of two built-in types. The first you have already seen; `String` is used to store a series of literal characters. The second, `int`, is used to store whole numbers, like 12, 1,000,000, or -37.

b) Adding variables to the Account class

To start with, create an Account class in Netbeans that includes a main method. Follow the instructions in NetbeansAppendix M on page 340 -- Creating a class with a test driver. Add these variables after line 12 in Code Example 7 on page 340.

Code Example 3.1

```
    String name;            // a String variable called name
    int balance;            // an int variable called balance
```

Adding the variables to the Account class.

`int balance;` declares a variable of type int with the name balance.

`"// an int variable called balance"` is a comment meant for a person to read; Java ignores it.

So far, this class is not very useful because although each Account could store a name and a balance, there isn't any way to do anything with them. That's what methods are for.

c) Methods (action) - Accessors and Withdrawal

Methods allow classes to do things, to perform actions. You can write methods to do whatever you like. The body of a method has a list of instructions to follow to accomplish whatever that method is supposed to do.

- **Accessing/changing information in an object**

There are two things that our Accounts are supposed to do besides retaining the name and balance associated with the Account: they must allow some other method, outside of Account to discover the value of the balance variable, and also allow the balance to be changed when money is withdrawn. So the questions that must be answered are: 1) How to get the value of a variable that's inside an object? and 2) How to set the value of a variable that's inside an object?.

Code Example 3.2

```
1    public void setBalance(int nuBalance) {
2        balance = nuBalance; // set the balance
3    } // setBalance
4
5    public int getBalance() {
6        return balance;   // return the balance
7   } // getBalance
```

Accessors for balance.

Lines 1-3: The setBalance method sets the balance to the value that is sent along with the message.

Line 4: This blank line is inserted to set off the methods visually. It is a simple, but important, element of style.

Lines 5-7: The getBalance method returns (i.e. sends back as its value) the current value of the balance variable.

The answer is ***accessors***, methods that allow you to access the variables inside objects. These common, simple, useful methods are not easily understood at first.The problem is that even though accessors all follow the same pattern, there are many details involved in the mechanisms that implement them. Once you have programmed for a few weeks (or a half a dozen, depending on how often and how carefully you do it) accessors will seem easy and natural. For now, accept them as an idiom. Add these two methods right after the two variables you already added.

This class, though very small, can now be tested. To reiterate, it is important to test your code as you develop it; that way, when something goes wrong, it is easier to isolate the error -- there are simply fewer places to look.

- **Testing the accessors**

How can you test this class? You write what is called a ***driver program***, the sole purpose of which is to test your class. This may seem like a waste of time, but it is not! To convince yourself that a class works you must test all its methods. Fortunately, there are only two: getBalance() and setBalance(int). Go to the Account class. Modify the main method so that it looks like this:.

Code Example 3.3

```
1 public static void main(String[] args) {
2     Account myAccount = new Account();
3     System.out.println("Before balance=" + myAccount.getBalance());
4     myAccount.setBalance(1234);
5     System.out.println("After balance=" + myAccount.getBalance());
6 }
```

Accessors for balance.

Line 2: Create and store an Account, call it myAccount.
Line 3: Check the initial balance, it should be 0. This tests getBalance().
Line 4: Set the balance to 1234.
Line 5: See if the balance is now 1234. This tests setBalance() and getBalance()

Now, set the main class (Project/Set Project Main Class) to Account, and execute your program. Assuming it displays 0 and then 1234, it worked and you can move on to the withdraw(int) method. If you made any typing mistakes, fix them. Then, forward!

- **Withdrawals**

Along with accessing the balance variable, the other action our Account class must take is handling withdrawals. Therefore it will need a method, which might as well be named withdraw. It is important to give methods (as well as classes and variables) descriptive names. That makes less things to remember.

When a user types an amount to withdraw and then hits the `Enter` key we should send our Account object a withdraw() message. Will the withdraw() method need any information to carry out its task? Yes, it needs to know how much to withdraw! So it will need a parameter to pass that information to the method. The amount to withdraw is a number, in whole dollars, so the type of the parameter is `int`. The name of the parameter is up to us; it can be any legal identifier -- in the example, the name `amountToWithdraw` was chosen (see Code Example 3.4).

How should the withdraw method change the state of the Account object which receives it? This is the same as asking how the state of a bank account should change when a person takes money from it using an ATM. The answer is that the balance should be reduced by the amount of money withdrawn. So, first our method must calculate the new balance by `balance - amountToWithdraw`, and then store that amount back in the balance variable. To change the value of a variable you use an assignment statement; like this: `balance = balance - amountToWithdraw;`. This assignment statement is the only line in the body of the withdraw method .

Code Example 3.4

```
1    public void withdraw(int amountToWithdraw) {
2        balance = balance - amountToWithdraw;
3    }
```

The withdraw method for the Account class.

Line 1: The method heading. There is one parameter of type int whose name is amountToWithdraw.

Line 2: An assignment statement. This will subtract whatever value is in the parameter amountToWithdraw from the value in the variable balance (inside whatever Account the withdraw message was sent to), and then store the result of the subtraction back in that same variable (inside the Account object that got the withdraw message).

Add this method to your Account class and test it. How to test? Add two more lines to your main method, possibly those in Code Example 3.5.

Code Example 3.5

```
public static void main(String[] args) {
    Account myAccount = new Account();
    System.out.println("Before balance=" + myAccount.getBalance());
    myAccount.setBalance(1234);
    System.out.println("After set balance=" + myAccount.getBalance());
    myAccount.withdraw(235);
    System.out.println("Withdrew, balance=$" + myAccount.getBalance());
}
```

Testing the withdraw method in Account as well.

Assuming your program says the final balance is $999, this means all three methods work and you are ready to use your Account class along with your GUI to build the complete system. Well, almost ready. There's the small matter of keeping track of three accounts instead of one.

iii) Objects and classes

Novice programmers sometimes struggle with the distinction between classes and objects. It is rather like the relationship between the part of speech, noun, and individual nouns. Words like "dog", "house", and "money", are nouns. I.e., "dog", "house", and "money", are instances of the category noun. Similarly, an object is an instance of a class. Every object is an instance of some class.

If you are just learning to program, until very recently you had never heard the terms class and object used the way they are used in the context of programming. This unfamiliarity makes making sense of the difference between them rather difficult. It gets easier with practice.

To review: classes are patterns for creating objects of that type; they define the information those objects will contain (variables) and the actions they can take (methods). Once the class is defined, you can create (instantiate) as many objects as you want of that type. Once you have created an object, you can send it messages to accomplish whatever task you are working on.

a) Cookies and cookie cutters metaphor

A useful metaphor for classes and objects is cookie cutters and cookies. If you have a cookie cutter in the shape of a star, you can use it over and over to make many star-shaped cookies. After the various star cookies are cut, they can be decorated in different ways. Classes are patterns (like cookie cutters) for making objects (cookies). All objects (also called instances) of a particular class have the same form (like all the cookies from the same cutter). Every Account you create (by saying `new Account()`) will have its own balance variable and its own name variable. These variables may contain different values in different objects.

It’s important to remember the distinction between objects and classes. For while cookies can be delicious, if you bite a cookie cutter you could hurt your mouth; plus it wouldn’t taste good.

b) new Account(); Instantiation! Alakazam!

Before you can send a message to an Account you must create it, and store it in a variable. This is demonstrated in "Code Example 3.5 on page 50" on page 52. Instantiation may safely be considered as magic for the present. In Chapter 5 the details will be explained, and at that point it will be essential to understand, but for now you might just think of `new Account()` as the incantation to make a new Account object appear.

From: Code Example 3.5 on page 50

```
1
2              Account myAccount = new Account();
```

Instantiation. A line of code that creates an Account object.

Line 2: There are two parts here: 1) Account myAccount - declares a variable of type Account (that's the class name) whose name is myAccount. 2) new Account() -- instantiates an Account (i.e. creates an instance of an Account object). The equals sign between them makes these two parts into an assignment statement. The action of this assignment is to store the newly created Account in the myAccount variable.

Recall, the convention is to name classes beginning with a capital (Account is the name of the class), and objects (or instances of the class) with a small letter (myAccount is the name of the object).

c) Instances and instance variables

The Account class has two instance variables, name and balance. Therefore, every object of type Account (or instance of Account) has its own instance variables named name and balance; just as real every bank account has a name and a balance associated with it. You will see an example of what this means in the next section.

Exercise: Use your Account class in conjunction with your GUI to keep track of the balance of one account. You already have all the necessary code; all you need is to understand it well enough to rearrange it to do the job.

E. Creating and testing the finished GUI

i) GUI design

We have built and tested a prototype GUI and completed the Account class. We are now ready to solve the original problem. We will again follow the technique of starting with the user interface and then writing the classes to support it. Designing the GUI both puts our focus on the user and what actions they can take (which is a good policy) and helps us form an image of the task ahead of us.

We already have a display Button and a TextField to handle withdrawals; all we need is a means to choose between the three accounts. How should the user select the account to withdraw from or display the balance of? In a real ATM machine, the user inserts their card and then enters their PIN. But, we don’t have a card reader, and don’t want to keep track of PINs just yet. A simple solution is to add three buttons for the three accounts. When the user pushes the Account2 button, the program will act as if the owner of the Account2 has successfully logged in; the program will display the balance from Account2. Subsequent withdrawals will come from Account2, until another Button is pushed.

ii) GUI implementation

Add three more Buttons to your GUI (see NetbeansAppendix D on page 331 if you’ve forgotten how). The Buttons appear named Button2, Button3, and Button4 and they are labelled the same way. These names do not remind you what they mean, and the user will object if to select account 3, they must push the Button labelled Button4. Just as it is important to give classes, variables and methods descriptive names so that you can remember what they do without thinking, it is important to label Buttons well or the user will become confused. When there is only one, the name is not so important, but the more there are, the more it matters. Change the labels on the Buttons to "Account 1", "Account 2" and, "Account 3" (see NetbeansAppendix I on page 338). Change their names to "selectAccount1Button", "selectAccount2Button", and "selectAccount3Button" (see NetbeansAppendix K on page 339). Make sure the Button labelled "Account1" is named "selectAccount1"! It is very confusing and hard to figure out when the Account1 Button causes access to Account3.

Now, add actions for the three new Buttons (double-click them, remember?). They are supposed to select the current account, so have each one send the Bank a message: selectAccount1(), selectAccount2(), or selectAccount3(). Naturally, the selectAccount1Button should send the bank the selectAccount1() message.

Here's how the actionPerformed code Netbeans wrote for the account1Button looks:

Code Example 3.6

```java
private void account1ButtonActionPerformed(java.awt.event.ActionEvent e{)
    // Add your handling code here:
}
```

actionPerformed() for account1Button.
Notice that the name of the Button appears as part of the method name.

All you need to do is add code to tell the bank to selectAccount1(), as shown in Code

Code Example 3.7

```java
private void account1ButtonActionPerformed(java.awt.event.ActionEvent e{)
    theBank.selectAccount1();
}
```

The code to add

Example 3.7. Do the same for the other two Buttons. You will notice that Netbeans indicates that there is an error on each of those lines. This is because it does not know what theBank means. You will need to add the line

```java
    Bank theBank = new Bank();
```

outside of any method (details below) and create the Bank class. After you do these two things, the errors will go away and the program will be ready to run.

F. The Bank class: Design, implementation and testing

i) Bank class design

The task of the Bank class is to simulate a tiny bank with three accounts. Thus, it will need three Accounts. It must also keep track of which Account is currently in use, and make withdrawals from that Account.

There are two tasks we must accomplish; 1) create the three accounts, and,
2) conceptualize and implement a technique to remember which of the accounts the user is currently working with. It turns out that declaring one more Account variable and setting it equal to whichever Account is currently in use will be sufficient; see below.

ii) Converting the design to Java code

The first thing to do is to create a Bank class with a main method; see NetbeansAppendix M on page 340 for instructions. Netbeans will write the following class

Code Example 3.8

```
1 /*
2  * Bank.java
3  *
4  * Created on April 21, 2004, 2:40 PM
5  */
6 public class Bank {
7
8     /** Creates a new instance of Bank */
9     public Bank() {
10    }
11
12    /**
13     * @param args the command line arguments
14     */
15    public static void main(String[] args) {
16    }
17}
```

The initial Bank class

a) Variables

To create the three accounts, plus the current account variable, add these four lines after line 6 in Code Example 3.8. That's all it takes.

```
    Account account1 = new Account();
    Account account2 = new Account();
    Account account3 = new Account();
    Account currentAccount = account1;   // to start with
```

The first three lines declare variables called account1, account2 and account3, instantiating three Accounts and storing one in each. The fourth declares another Account variable called currentAccount, and sets it equal initially to account1. Thus if you do a withdrawal before pushing any of the three buttons it will come from account1.

Code Example 3.9

```
public class Bank {

    Account account1 = new Account();
    Account account2 = new Account();
    Account account3 = new Account();
    Account currentAccount = account1;   // to start with

    /** Creates a new instance of Bank */
    public Bank() {
```

The Account declarations inserted into the Bank class.

b) Methods

The bank needs to withdraw money from and display the balance of the current account; it also must be able to set the current account. Code Example 3.10 shows the methods that do these things. These may be inserted anywhere in the class block of the Bank class, as long as they are outside of other methods. Right after the variables would be fine.

Code Example 3.10

```
public void selectAccount1() {currentAccount = account1;}
public void selectAccount2() {currentAccount = account2;}
public void selectAccount3() {currentAccount = account3;}

public int getBalance() {
    return currentAccount.getBalance();
}

public void withdraw(int withdrawalAmt) {
    currentAccount.withdraw(withdrawalAmt);
}
```

The method declarations for the Bank class.

Both getBalance() and withdraw just pass the buck to the Account class by sending the same message to the currentAccount object. Notice that getBalance returns the value that comes back from Account’s getBalance.

c) Testing

Modify the main method in Bank as shown in Code Example 3.11. Then set the starting class to Bank (Project/Set Project Main Class...) and run the program again to make sure the Bank is working correctly. Sometimes it is helpful to draw a picture of the state of the

Code Example 3.11

```
1 public static void main(String[] args) {
2     Bank theBank = new Bank();
3     System.out.println("Initial acct1 balance=" + theBank.getBalance());
4     theBank.withdraw(100);
5     System.out.println("-100 acct1 balance=" + theBank.getBalance());
6     theBank.selectAccount3();
7     System.out.println(" acct3 balance=" + theBank.getBalance());
8     theBank.selectAccount1();
9     System.out.println(" acct1 balance=" + theBank.getBalance());
10    }
```

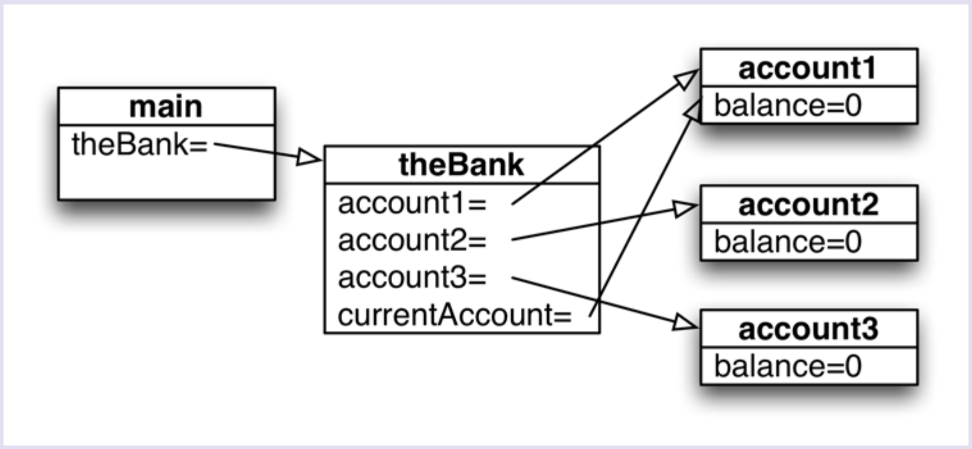
Test code for the Bank class.

Line 2: Create a new Bank object and store it in a Bank variable named theBank.
Line 3: Display the balance of account1 (remember currentAccount starts as account1).
Line 4: Withdraw 100 dollars from account1
Line 5: See if the balance is really -100.
Line 6: Tell theBank to change the current account to account3.
Line 7: See that getBalance now returns 0 (instead of -100)
Line 8: Go back to account1.
Line 9: Verify that it’s balance is still -100.

program at various points during its execution. For instance, after line 2 in Code Example

3.11 executes, the state of the program looks roughly like Figure 3.3. After line 6

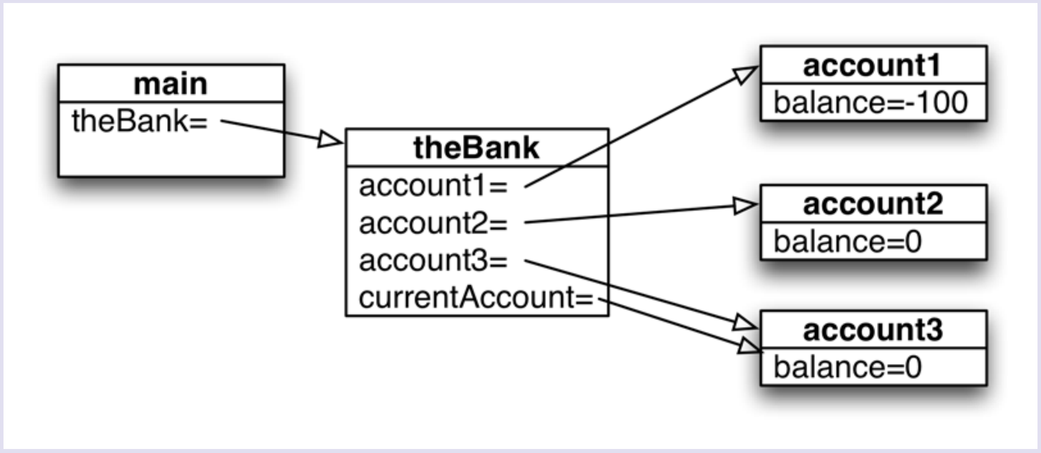
Figure 3.3



The state of the program after line 2 in Code Example 3.11 executes.
The main() method has a variable named theBank; it has four variables. The first three point to the three Accounts. The fourth, currently points to account1 as well.

executes the state of the program looks like Figure 3.4

Figure 3.4



The state of the program after line 6 in Code Example 3.11 execute
The only differences are the value of the balance in account1 and the value of currentAccount in theBank, which now points to account3 instead of account1.

G. Putting it all together - finally!

All you need to do to finish the project is to add one line to the Applet.

Code Example 3.12

```
public class ATM_Applet extends java.applet.Applet {
   Bank theBank = new Bank();

```

Alteration to Applet code.
To declare the Bank variable, theBank, add this line at the beginning of the ATM_Applet class.

That’s it! You are done! Now you can compile and run the Applet. Do that now (don’t forget to change the main class back to ATM_Applet -- Project/Set Project Main

Class...). Make sure it really does keep track of the three balances correctly. You will notice that everyone starts out with a balance of 0, but it still allows them to withdraw money.

Could you arrange for them to start with some other balance besides zero?

This turns out to be easy, but not at all obvious. The initial ***`default value`*** for every

Code Example 3.13

```
public class Account {
   String name;
   int balance=1000000; // Every Account starts with $1,000,000!

```

Alteration to Code Example 3.1.
To set the initial balance for every account to $1,000,000. Change is in **bold**.

instance variable is 0, but you can set it to something else as shown in Code Example 3.13. Do that now and rerun your program.

Assuming you've made that program run, good work! You have done some of the most difficult programming you will ever do; working in a new language in a new IDE is incredibly difficult. There are so many details and everything is unfamiliar so even the simplest problem can seem overwhelming. It only gets easier from here. With practice everything you did today will become easy and effortless, and in a few weeks you'll be amazed that this ever seemed difficult. That's how expertise works; you learn something and it seems simple. Then it's hard to understand why everyone else can't do it. Welcome to Java programming.

H. Conclusion

Programming is always an iterative process. No one writes finished programs from scratch in one go. Rather it is a process of successive approximation. There are two compelling reasons for starting with a prototype which does almost nothing and then adding functionality after each simpler program works. First, it allows the programmer to focus on one part of the program at a time, and thus reduces cognitive overhead during the design and implementation phase. This is also an advantage of writing and testing classes one at a time. Second, it makes the task of debugging much easier. Debugging is the hardest part of programming for all levels of programmers; it can be baffling, frustrating and exhausting. Stepwise implementation makes it much easier to find bugs when they occur, simply because a smaller part of the program is new at any time.

This chapter introduced classes; their design, implementation, testing, and incorporation into a larger program. It included various language constructs and components, including: variables, the assignment statement, declarations, instantiation, accessors, parameters, and return values along with the AWT Components, Button, and TextField. These various elements were presented rather telegraphically, with most of the detail omitted. A more complete and careful discussion of these topics may be found later on. Here the emphasis is on constructing a working program to experiment with to get a feel for both the Java language and the process of programming.

If you new to programming and/or Java you may be feeling a bit confused. You probably have many questions about what you've just done, and a number of concepts you are very unsure about. If so, good! There were too many concepts and details to describe or understand all at once. But, at least you have seen the process of constructing a working Applet with several classes and a GUI. That's a lot (And it's most of what programming is about.). It usually takes about 3 or 4 weeks for undergraduates in an introductory class to be able to do this sort of program. Oh, and then there was all the detail of using Netbeans to build the GUI and compile/execute the program. The next chapters will fill in the detail, and if it answers questions you have in the back of your mind, the details will be easier to understand and remember.

The next chapter will introduce Graphics, Color and software reuse techniques through the development of another example. After that comes a detailed explanation of many of the concepts and programming features glossed over here. Then some more elaborate (and interesting) examples will be undertaken.

I. End of chapter material

i) What could go wrong?

In "When the user hits enter, get the withdrawal amount" on page 44, if the contents of the TextField is not an int, an Exception will be thrown.

In "GUI implementation" on page 53, it is very easy to get confused over which Button is which, and which ActionPerformed() should send what message. If something goes wrong with the account selection that would be the first place to look.

i) New terms in this chapter

default value - the value you get if you don't do anything. Instance variables are assigned zero by default when they are created. If you want to initialize them to something else you may (e.g. int x=17;). 57
GUI - graphical user interface 39
recursive definition -- a definition that uses the thing being defined 41
variable - memory to store one value of a particular type 42

ii) Review questions

3.1 How do you create objects?
3.2 What is the difference between a class and an object?
3.3 When you declare a variable, you must specify its type and name (in that order). What are you allowed to call variables? I.e. what rules must variable name (indeed all Java identifiers) adhere to?
3.4 Here are two legal names: something, and Something. These are different, because Java is case sensitive. If you are following the convention for naming classes and objects which of these is the name of a class and which the name of an object?
3.5 Where is information stored in Java program?
3.6 How do you change the value of a variable?
3.7 What is the difference between the types int and String?
3.8 What are the three attributes of every variable?
3.9 What are parameters used for?
3.10 What are imports for?
3.11 Where do parameters go?
3.12 Is `doSomething()` a variable name or a method? How can you tell?
3.13 In `anything.everything(something)`, you can tell by the context what `anything`, `something` and `everything` are. What are these three things? The answers are message, object, and parameter, but which is which?
3.14 Recall that computing is always information processing; in the ATM problem description, what information is processed, input, or output, for each possible user action?

iii) Programming exercises

3.15 The instructions omitted the labels above and below the TextField. Add them.
3.16 Your finished Applet was not very user friendly. When a customer withdrew cash, and wanted to see the new balance, they had to push the display button again. It is simple (i.e. one line) to automatically display the new balance every time a withdrawal is made. Do so. Hint: write a public void display() method in your Applet

(that does exactly what the display button does) and use it in actionPerformed for the TextField right after you do the withdrawal (i.e. say display()).

3.17 After you do the previous exercise, you will discover a small inconvenience. If you want to withdraw the same amount repeatedly from the same account, you have to keep removing the balance and reentering the amount before you can withdraw again. Improve the usability of your interface as follows. After a withdrawal is made, move the cursor to the TextField and select the text so the user can simply copy the amount to be withdrawn (ctrl-c) and then paste and hit enter over and over. Add the code to actionPerformed for the TextField (right after the display() from the previous exercise). If the name of your TextField was theTF, the code would be:

```
theTF.requestFocus();
theTF.selectAll();
```

Chapter 4: Graphics and Inheritance

"One is not likely to achieve understanding from the explanation of another."
Takuan Soho

A. Introduction

This chapter will give you more exposure to and practice with writing classes in Java. It will also illustrate how to do simple graphics and introduce inheritance, a powerful feature of object oriented programming. Like the last chapter, it will not present all the details of the constructs used; that will be delayed until the next chapter. For now, try to become familiar with the process of thinking a problem through, coming up with an elegant design for a solution, then implementing and testing it -- those are the important lessons that will carry over into other programming languages and possibly even other areas. To learn to program, you must practice, reading about it is not good enough (as the Takuan quote implies).

i) A description of the task

Your task in this chapter will be to draw two eyes on the screen. For simplicity you need only draw the iris and pupil. Make the iris the exact same color as yours. The distance between the two eyes and the size of the pupil relative to the iris should be adjustable by the user.

ii) Creating a prototype

In the previous chapter you were introduced to the techniques of: a) Building a prototype then gradually adding functionality, and b) Sketching the GUI, then creating and testing it before writing any other code. Do that now. First, create a new project in Netbeans (consult with "Getting started with Netbeans and the greetings program" on page 327 if you've forgotten how). Add a GUI Applet called EyeApplet. Add and connect however many Buttons you will need. Compile and run your project, testing to make sure everything works so far (i.e. that all the Buttons invoke the correct `actionPerformed()` method, see "Adding, connecting and testing a Button" on page 331). Now you are in a position to try out the various Graphics commands as you work through the chapter.

iii) Object Oriented Design -- choosing classes to implement

A decision that you must make early in the design of a program to solve some problem is what classes you will use in the solution. Since your task is to draw two eyes on the screen, a natural candidate for a class would be Eye. Since an Eye consists of an iris and a pupil, two circles filled with different colors; Iris and Pupil are also candidate classes. How many classes make sense in a particular context is less than perfectly defined. For now, let's assume you will need an Eye class, and put off the decision on Iris and Pupil until you know a little more. Before designing the Eye class, there are several facts about Graphics and Color in Java that you need to know. Often in designing a class you must do some experiments, play with the elements involved, and learn about the related classes Java provides, before you know enough to make informed decisions about the details of the class you are writing. These next several sections will illustrate that process; then we will return to the Eye class; and once the Eye class is done, so is our task!

B. The Graphics class

i) The Graphics context

Java provides the java.awt.Graphics class to draw on the screen. The Sun API documentation says it well: "A Graphics object encapsulates state information needed for the basic rendering operations that Java supports.". In other words, to draw on the screen you need to have a Graphics context in which to do so. You have already seen how to draw on the screen in an Applet by including a public void paint(Graphics) method (on page 32). This section will provide a few more details.

ii) Inheritance, Components and public void paint(java.awt.Graphics)

When you write a `public void paint(Graphics)` method in your EyeApplet, it overrides the default `paint()` method in Applet. That sentence requires a bit of explanation. Look at the heading of the EyeApplet class definition (in the EyeApplet.java file). The first line of code is: `public class EyeApplet extends java.applet.Applet`; thus, the class is named EyeApplet, it is public and it is a subclass of java.applet.Applet (in other words, it extends java.applet.Applet). When you extend a class, instances of the subclass inherit all the functionality of the superclass. To add functionality, you simply add methods. To change the behavior of a method in the superclass, you write a subclass method with the same ***`signature`*** that does something different. Because EyeApplet extends Component (actually it extends Applet which extends Panel, which extends Container, which extends Component, but never mind right now), it automatically inherits a paint method (which does very little). If you want to paint your Applet differently (by drawing a circle or whatever on the screen) you write a `public void paint(java.awt.Graphics)` method in your subclass, and then that is executed instead. You will see examples of adding and modifying functionality in the FilledCircle class, below.

iii) Basics of graphics in Java

a) The coordinate system

From the perspective of a Java graphics context, the drawable area is a rectangle of dots numbered from left to right and top to bottom. The dots are called **pict**ure **el**ement**s** or ***pixels***. The pixel numbered (0,0) is in the upper left corner.

When you create a GUI Applet in Netbeans it automatically generates an ***HTML*** file for a browser to use in determining how to display it. Assuming you named your Applet EyeApplet, the HTML file is called EyeApplet.html. It is stored in the same directory as your source files. It contains the line:

```
<APPLET code="EyeApplet.class" width=350 height=200></APPLET>
```

This specifies the width and height of the graphics context (in pixels) within the frame of the Applet. This will be illustrated in the next section. If you want your Applet to be a different size, simply change the 350 and 200 (use the Netbeans editor!).

b) A few Graphics methods

The only method you need to draw an eye is `drawOval()`, but that will be easier to understand if first you know how `drawRect()` works. The `drawRect()` method has four parameters, all ints. The first two specify the upper left corner of the rectangle; the third and fourth, its width and height. In each pair, the first is horizontal, the second is vertical. Thus `drawRect(x,y,width,ht)` will draw a rectangle whose upper left corner is at (x,y), whose width is width, and whose height is ht.

The `drawOval()` method is similar. The four parameters are identical, specifying a rectangle, exactly as in `drawRect()`; the oval is inscribed in the specified rectangle.

The `drawLine()` method also has four parameters; the first two specify the coordinates of one end of the line, the second two, the other. See Code Example 4.1 for an illustration.

Code Example 4.1

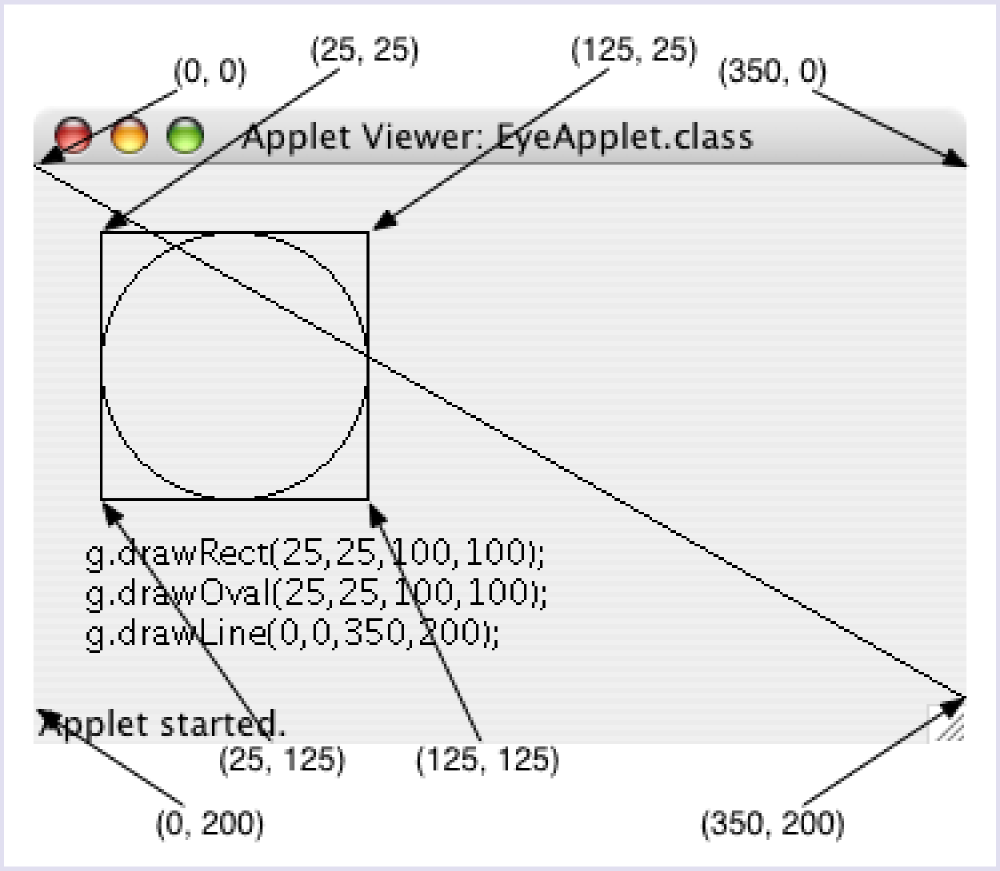
```
1    public void paint(java.awt.Graphics g) {
2        g.drawRect(25,25,100,100);
3        g.drawOval(25,25,100,100);
4        g.drawLine(0,0,350,200);
5        g.drawString("g.drawRect(25,25,100,100);", 20, 150);
6        g.drawString("g.drawOval(25,25,100,100);", 20, 165);
7        g.drawString("g.drawLine(0,0,350,200);", 20, 180);
8    }
```

A paint() method - see Figure 4.1 for its result.

Line 2: Draws a square with sides 100 pixels long, upper left corner at (25,25)
Line 3: Draws a circle centered in it, i.e. centered at (75,75), *not* (25,25).
Line 4: Draws a line from one corner of the graphics context to the other.
Lines 5-7: Draws Strings representing the previous 3 messages on the screen.

Figure 4.1

9



```
g.drawRect(25,25,100,100);
g.drawOval(25,25,100,100);
g.drawLine(0,0,350,200);
```

The result of the paint() method in Code Example 4.1
The coordinates of the corners of the graphics context and the square are indicated.

Notice that if we were hoping for a circle centered at (25,25) we did *not* get what we wanted. We will have to take this into account in writing the graphical display method for the Circle class.

C. The Circle class -- design and implementation

Circles are used to represent many things in GUIs. In a simulation circles might represent molecules; in a game, balls; on a map, populations in cities, or incidence of infectious disease.

As you have just seen, you can draw a circle in a graphics context by using `g.drawOval(int, int, int, int)`. But, if you were writing a program that displayed many circles, or if the circles moved around the screen (like the balls in a billiards game), or changed sizes (like graphics representing levels of infection), you wouldn't want to keep finding and changing the appropriate `drawOval()` code. If you tried, it would take a lot of careful attention to avoid changing the wrong one. A better solution is to hide the information about a particular circle, namely where and how big it is, inside an object. Then when you have multiple circles, you can just deal with them as Circles and let the details of where they are right now and how to draw them be handled by the Circle class. Additionally, if you need to change how they are drawn, there is only one piece of drawing code to change instead of numerous copies of it. Thus, you can avoid complexity and bugs at the same time. What a deal!

As with any class, in designing the Circle class, you must decide: 1) what information it will contain, and 2) what actions it will support, including how it will be displayed and tested.

i) Circle class design

a) State - What information completely describes the state of a circle?

To completely describe a circle you must specify its center and its radius; that's it. So we will need three variables; one for its radius and two for its position. For simplicity, and since the screen is made of discrete pixels, these can all be whole numbers, ints. For now. When we use Circle to display molecules later, it will turn out to be crucial for their positions to be able to be intermediate between pixels.

b) Action - What must a Circle do?

We will need accessors for all our variables, so that we can discover and/or change the position or size of a Circle. We will also need methods to display a Circle, both for debugging and graphically.

ii) Converting the design to Java code

a) Creating the Circle class

Create a Circle class (see "Creating a class with a test driver" on page 340 to refresh your memory, if needed).

b) Variables

Perhaps you already know how to declare the three variables? That would be:

```
int x;
int y;
int radius;
```

c) Accessors

These are just like the accessors for Account (See Code Example 3.2 on page 48) except the names of the variables are changed. Look back at that example and try to write the accessors for x, before looking at them on the next page.

d) toString()

Code Example 4.2

```
public int getX() {return x;}
public int getY() {return y;}
public int getRadius() {return radius;}

public void setX(int nuX) {x = nuX;}
public void setY(int nuY) {y = nuY;}
public void setRadius(int nuRadius) {radius = nuRadius;}
```

Accessors for the Circle class

Java has a special method that is used almost exclusively for debugging. It is called `toString()`, and its signature is: `public String toString().` As the name implies (once you are familiar with colloquial Java-speak) it converts an object to a String. While you are testing your classes (or debugging in general) you sometimes need to know what information is in an object, whether it contains the information you expect. If you have an object called anyObject, you can always find out what's inside by:
`System.out.println(anyObject);`

You don't need to type `.toString()` because in the context of a `System.out.println`, Java automatically adds it for you (although you may type it if you want).

You can write any `toString()` method that you choose, so long as the signature matches. Here's a `toString()` for the Circle class that is a bit verbose. Why it is written this particular way will become obvious in the next chapter:

Code Example 4.3

```
1    public String toString() {
2        String returnMe = "I am a Circle: ";
3        returnMe += "\tx=" + getX();
4        returnMe += "\ty=" + getY();
5        returnMe += "\tradius=" + getRadius();
6        return returnMe;
7    } // toString()
```

toString() for the Circle class

Line 2: Declare the String variable to return; set it to "I am a Circle: "
Line 3: Paste a tab (\t) onto it, followed by "x=" and the value of x.
Line 6: Return that whole String as the value of `toString()`

iii) Testing your code

That's enough code to test. Create a Circle and check that all the methods work. To do that, after you instantiate the Circle, display it, then change all the variables and display it again. Since toString uses `getX()`, `getY()` and `getRadius()`, by doing that you have used all the methods. See Code Example 4.4 for how the code might look. Type and run this

Code Example 4.4

```
1 public static void main(String[] args) {
2     Circle aCircle = new Circle();
3     System.out.println("before" + aCircle);
4     aCircle.setX(123);
5     aCircle.setY(17);
6     aCircle.setRadius(34);
7     System.out.println("after" + aCircle);
8}
```

Testing the first prototype Circle class.

Line 2: Instantiate a Circle called aCircle
Line 3: Display it.
Lines 4-6: Set all the variables.
Line 7: See if they changed.

test program (if you made mistakes, debug your typing). Then, since you know the Circle class can keep track of and change its variables correctly you are ready to add the graphical display.

D. Displaying a Circle graphically

To display a Circle graphically requires a method that draws the right sized circle in the correct location. It seems this should only take one line of code. Something like:
`g.drawOval(x,y,width,ht);`
But, what values should we use for x, y, width and ht? And where does the graphics context, g, come from?

i) public void paint(java.awt.Graphics)

A Circle knows its location and size (the location of the center is in its x and y variables and its size is in its radius variable). There is no way a Circle should know anything about graphics contexts, so that is best provided from the outside, by whatever method asks the Circle to display itself. This is what parameters are used for, to pass information to a method. The Applet was displayed graphically by a method called paint(); to keep down the cognitive overhead, we will use the same name for the method that displays a Circle graphically. So, perhaps all we need is: `g.drawOval(x,y,radius,radius);` as in Code Example 4.5).

Code Example 4.5

```
public void paint(java.awt.Graphics g) {
    g.drawOval(x,y,radius,radius);
}
```

A first try at a paint method for Circle.

This method has two logic errors; can you find them?

ii) Testing the paint method

Add the paint method from Code Example 4.5 to your Circle class and test it by modifying your EyeApplet to create a Circle, set its x, y, and radius to 100, and display it graphically in `paint()` (see Code Example 4.6). Unfortunately this draws the circle that

Code Example 4.6

```
1 public class EyeApplet extends java.applet.Applet {
2    Circle aCircle = new Circle();
3
4    /** Initializes the applet EyeApplet */
5    public void init() {
6        initComponents();
7
8        aCircle.setX(100);
9        aCircle.setY(100);
10       aCircle.setRadius(100);
11    }
12
13    public void paint(java.awt.Graphics g) {
14       aCircle.paint(g);
15    }
16
```

Creating and displaying a Circle graphically.

Lines 8-10: set the variables in aCircle to 100.

Line 14: to display the Applet, display aCircle; note that we are sending the graphics context (that came into `paint()` as the paramter, g) as a parameter.

would fit inside a square of size radius, whose upper left corner is at (x,y). That has two problems: 1) it is centered at (x+radius/2, y+radius/2), and, 2) its diameter is the radius of the Circle (To understand this, draw yourself a picture labelled with coordinates.

Problem Solving Technique

Draw a picture.

By drawing a picture, you can engage your visual-spatial processing system. Although people tend to take it for granted, the ordinary ability to walk through a crowd involves a feat of information processing. Your visual-spatial processor is more powerful than any computer on the planet, but so long as you are stuck in linguistic space it is idle. Drawing a picture can activate it. And when you look back at the picture you will remember what you were thinking.

Not interested in drawing right now? Then, it's time to put this down and do something else. You can't, *can't*, ***cannot***, program without understanding. It's hopeless. Seriously. So, either take the time, spend the effort to understand it, or, do something else! No sense wasting your time.). That second problem is very easy to fix, simply pass radius*2 instead of radius for the width and height to `drawOval()` (see Code Example 4.7). Now

Code Example 4.7

```java
public void paint(java.awt.Graphics g) {
    g.drawOval(x,y,radius*2,radius*2);
}
```

A second try at a paint method for Circle.
This method gets the size right, but the circle is still in the wrong place.

the size is correct, but the center is at (x+radius,y+radius), instead of (x,y). How could you fix this? The simplest way is just to subtract the radius from x and y in the parameters you send to drawOval, as seen in Code Example 4.8. Modify your code.

Code Example 4.8

```
public void paint(java.awt.Graphics g) {
    g.drawOval(x-radius,y-radius,radius*2,radius*2);
}
```

A correct paint method for Circle.

Draws a circle whose radius is radius, centered at (x,y). In the context of a particular Circle, x, y, and radius are variables specifying the state of that Circle.

Execute it to verify that the circle is displayed at the appropriate place.

iii) More than one Circle

Now that you have a working Circle class, you can create and display as many Circles as you want. Add another Circle as shown in Code Example 4.9. Execute it to make sure it is working properly; the circles should be concentric. Add another, intersecting Circle to make sure you understand the procedure (don't forget to add a line in `paint()` to display the third one!). Notice that you can add and display as many Circles as you want without ever looking back at the Circle class code. This is a huge advantage of programing with objects; once a class is written, you can forget the details inside it.

You are almost ready to design and implement the Eye class, as soon as you know a bit about color in Java.

Code Example 4.9

```
public class EyeApplet extends java.applet.Applet {
    Circle aCircle = new Circle();
    Circle bCircle = new Circle();

    /** Initializes the applet EyeApplet */
    public void init() {
        initComponents();

        aCircle.setX(100);
        aCircle.setY(100);
        aCircle.setRadius(100);
        bCircle.setX(100);
        bCircle.setY(100);
        bCircle.setRadius(50);
    }

    public void paint(java.awt.Graphics g) {
        aCircle.paint(g);
        bCircle.paint(g);
    }
```

Adding a second concentric Circle.

Changes to Code Example 4.6 are in bold. If you wanted a non-concentric Circle, change the parameters in `setX()` and `setY()` for bCircle.

E. The Color class

i) Setting the color of the Graphics context

A graphics context has a number of state variables, including the current color. The default color is black. You can change it with the accessor setColor(Color); i.e. you set the color of a Graphics object by sending it a `setColor()` message with a Color as the parameter. Just as you set the balance of an Account by sending it the `setBalance()` message with an int as the parameter; or the radius of a Circle using `setRadius()`.

ii) Built in Colors

The Color class has about a dozen colors predefined. To set the color to red, you would say: `g.setColor(java.awt.Color.RED);` Add that line between lines 18 and 19 in Code Example 4.9 and execute the Applet. Notice that only the second circle is red; if you move the `setColor()` before line 18, then both Circles will be red.

iii) Creating your own Colors

There are millions of colors possible in Java. You can create any of them by saying:
`java.awt.Color myColor = new java.awt.Color(red, green, blue);`
The three int parameters to the Color constructor set the intensities of red, green, and blue. All three must have values in the range, 0 to 255.

Exactly how many colors are available? You can calculate this from the fact that there are three parameters, each of which can take on 256 different values. It's very much like the analysis in "Problem Solving Principle #1 - Build a prototype" on page 7.

a) RGB color model

Java has two color models, but the simpler is the RGB model. RGB stands for red-green-blue. The color of each pixel is determined by the amount of illumination in those three colors. Any combination of values for red, green and blue is legal. To get pure red, you say `new java.awt.Color(255, 0, 0);` thus passing 255, 0, and 0, as the parameters to `new java.awt.Color()`; 255 for red (i.e. all the way on), 0 for green and blue (i.e. all the way off). Purple is a mixture of red and blue. So, for bright purple you would pass (255,0,255); for dark purple, perhaps (50,0,50).

b) The difference between pigment and light

The RGB values set the intensity of light emitted in each color. You are probably more used to mixing pigments than light. Light and pigment are not identical. If you mix blue paint with red paint, you get purple. If you then add yellow you get muddy brown (or possibly even black). When you mix red light with blue light you also get purple; but if you then add green, you get white! A combination of all colors of light yields white light. Think of a prism. It breaks white light into its constituents. If there is no pigment, white paper remains white; if there is no light, everything is black. So
`new java.awt.Color(255,255,255)` is white, `new java.awt.Color(0,0,0)` is black.

F. The Eye class: design and implementation

i) Designing an Eye class

For your purposes here, an Eye is two concentric Circles, the larger (the iris) filled with the color of your eyes (some shade of brown, blue, or green), the smaller (the pupil) filled with black. If you wanted two unfilled black circles, the Eye class could have two Circle variables and you'd be almost done already.

For the user interface, you must allow the user to move at least one eye horizontally, and adjust the size of the pupils. The simplest way to handle this is with two Buttons to move one eye right and left; and two more to grow and shrink the pupils (if your interface uses some other scheme, that's fine). Thus, you will need methods to change the size and location of an Eye. Fortunately these will be very easy. Assume an Eye had two Circle variables. When the user wanted to shrink the pupils, a `shrink()` message would be sent to the Eye. The `shrink()` method could then reduce the size of the pupil Circle using `getRadius()` and `setRadius()` (i.e. `iris.setRadius(iris.getRadius()-3)`). Similarly, the `moveRight()` method could adjust the locations of both Circles using `getX()` and `setX()`.

The Circle class does almost what you need already. Two things need to be modified. Instead of drawing a circle in black, you want it to fill the same circle in a particular color. One way to accomplish this would be to modify the Circle class. You could change `drawOval()` to `fillOval()` in `paint()`, add a Color variable and use it to set the color of the graphics context before you fill the circle. But, then if you wanted to be able to draw unfilled circles you'd need to remodify the Circle class. Instead, we will extend the Circle class. That way, you won't have to change the Circle class and code reuse can be illustrated by subclassing.

ii) class FilledCircle extends Circle

As mentioned above, when you extend a class, the subclass inherits the data and methods of the superclass. Thus FilledCircle can use x, y, and radius without redeclaring them. Plus, you can add additional methods to add functionality, and override existing methods to change functionality.

a) Design

- **Variables**

FilledCircle inherits x, y, and radius from Circle. It needs one additional variable, to keep track of its color.

- **Methods**

There must be some way to set the color of a FilledCircle, so it will need a `setColor()` accessor. The `paint()` method must be modified to fill the circle in that color instead of just drawing the outline of the circle.

b) Implementation

- **Create a FilledCircle class**

If you need to consult with "Creating a class" on page 339 to refresh your memory, but, pay attention this time! There's no sense being tethered to the appendix for longer than necessary.

- **Add a color variable**

Here's how to declare a variable of type Color called myColor:

```
 protected java.awt.Color myColor = new java.awt.Color(100,0,100);
```

This line autoinitializes myColor to a medium purple. This default color will help in debugging; anytime you see it, you will know that you forgot to set the color for this FilledCircle.

Some people don't like to type, or look at, "`java.awt.`" over and over. If you would prefer to just type:

```
protected Color myColor = new Color(100,0,100);
```

See Code Example 4.11 for a technique to allow this.

- **Add the accessor to set the color of the FilledCircle**

To be able to change the color of a FilledCircle, there must be an accessor. The standard name is `setColor()` and its form is identical with the other accessors you've seen; see Code Example 4.10. Before long accessors like this will be second nature. For now, realize that there is one parameter of type Color, named `c` (line 7, `java.awt.Color c`); and whatever value is passed through that parameter is stored in the instance variable named myColor (line 8, `myColor = c;`).

- **Override paint()**

The heading of `paint()` is `public void paint(java.awt.Graphics g)`. Thus, the parameter is of type Graphics and is named g locally (i.e. in the `paint()` method). The body of the method must first set the Graphics color to the color of this particular FilledCircle, then draw the filled circle. See Code Example 4.10.

Code Example 4.10

```
1 public class FilledCircle extends Circle {
2    protected java.awt.Color myColor = new java.awt.Color(100,0,100);
3
4    /** Creates a new instance of FilledCircle */
5    public FilledCircle() {}
6
7    public void setColor(java.awt.Color c) {
8        myColor = c;
9    }
10
11   public void paint(java.awt.Graphics g) {
12       g.setColor(myColor);
13       g.fillOval(x-radius, y-radius, radius*2, radius*2);
14   }
15}
```

The FilledCircle class.

Line 2: Declare a Color named myColor and initialize it to medium purple
Lines 7-9: Accessor to set the color of a FilledCircle
Lines 11-14: Paint the FilledCircle by setting the graphics color, then `fillOval()`

See Code Example 4.11 for a way to avoid typing java.awt. over and over.

Code Example 4.11

```
1 import java.awt.*;
2 public class FilledCircle extends Circle {
3   protected Color myColor = new Color(100,0,100);
4
5   /** Creates a new instance of FilledCircle */
6   public FilledCircle() {}
7
8   public void setColor(Color c) {
9       myColor = c;
10    }
11
12  public void paint(Graphics g) {
13      g.setColor(myColor);
14      g.fillOval(x-radius, y-radius, radius*2, radius*2);
15  }
16}
```

Simplified FilledCircle class using import.

Line 1: This import statement allows you to skip typing java.awt. before Color and Graphics. Compare to Code Example 4.10.

iii) Testing FilledCircle

Modify your existing Applet to test FilledCircle. It will be enough to simply change the Circles to FilledCircles and set the color of the smaller one to black. What will it display if it is working correctly? See Code Example 4.12 for the necessary changes.

Code Example 4.12

```
1 public class EyeApplet extends java.applet.Applet {
2    FilledCircle aCircle = new FilledCircle();
3    FilledCircle bCircle = new FilledCircle();
4
5    /** Initializes the applet EyeApplet */
6    public void init() {
7        initComponents();
8
9        aCircle.setX(100);
10       aCircle.setY(100);
11       aCircle.setRadius(100);
12       bCircle.setX(100);
13       bCircle.setY(100);
14       bCircle.setRadius(50);
15       bCircle.setColor(java.awt.Color.BLACK);
16   }
17
18   public void paint(java.awt.Graphics g) {
19       aCircle.paint(g);
20       bCircle.paint(g);
21   }
```

Test code for FilledCircle.

Changes from Code Example 4.9 are in **bold**.
Lines 2-3: Declare, instantiate, and store FilledCircles instead of Circles.
Line 15: Set the smaller's color to black so it won't be purple!

iv) The Eye class

Having built and tested a GUI Applet and a FilledCircle class, most of the work of building the Eye class is finished. Create an Eye class and add the following variables and methods.

a) Variables

An Eye has an iris and a pupil; these are both FilledCircles. Thus:

```
FilledCircle iris = new FilledCircle();
FilledCircle pupil = new FilledCircle();
```

b) Methods

Because an Eye is composed of two FilledCircles, most Eye methods will simply send the appropriate messages to those FilledCircles.

- **move left and move right**

To move an Eye left you must move both of its FilledCircles left, so the moveLeft method would simply set x in each to a slightly smaller number; see Code Example 4.13.

Code Example 4.13

```
1     public void moveLeft() {
2        iris.setX(iris.getX()-2);
3        pupil.setX(iris.getX());
4    }
```

moveLeft() for Eye.

Line 2: Set the x coordinate to 2 less than it was.
Line 3: Having reset iris.x, set pupil.x to the same thing.

The `moveRight()` method would be similar, except increasing x for each.

- **shrink pupil and grow pupil**

To shrink the pupil you can simply reduce the radius of the pupil FilledCircle; see Code Example 4.14. The `growPupil()` method is nearly identical. After you add these methods

Code Example 4.14

```
1    public void shrinkPupil() {
2        pupil.setRadius(pupil.getRadius() - 2);
3    }
```

shrinkPupil() for Eye.

Line 2: Set the radius to 2 less than it was.

to the Eye class, go back to the `actionPerfomed()` method for the shrink and grow

Buttons, and modify them to send those messages. There are two things you must make sure of in doing this:

1. There must be an Eye variable declared before you can send the message to it. Every message has the form `someObject.someMessage();` -- see "The message statement" on page 96.
2. To change what is displayed, you must invoke paint(Graphics) and to do that you must send the `repaint()` message. The details of this will be explained in a later chapter. For now, just use the code in Code Example 4.15

Code Example 4.15

```
1   private void growButtonActionPerformed(java.awt.event.ActionEvent evt) {
2       rightEye.growPupil();
3       repaint();
4   }
```

actionPerformed() for growButton

Line 2: Send the rightEye the growPupil message.
Line 3: Repaint the Applet so you can see the new pupil size -- don't forget this!!

- **public void paint()**

To display an Eye you must display both FilledCircles, first the iris, then the pupil (since if you do it in the other order, the pupil will be invisible) see Code Example 4.16. That's

Code Example 4.16

```
  public void paint(java.awt.Graphics g) {
     iris.paint(g);
     pupil.paint(g);
 }
```

paint() for Eye

See the simplicity of composition?

all there is to it.

That's all the methods we need (Or is it? Check the design to see if we did everything we planned to. Look back at Code Example 4.12, which tested the FilledCircle class; did it send any messages besides `paint()` to the FilledCircles?), so it's time to test.

c) Testing

Modify your Applet to create and display one Eye, as in Code Example 4.17. Run it.

Code Example 4.17

```
public class EyeApplet extends java.applet.Applet {
   Eye rightEye = new Eye();

   /** Initializes the applet EyeApplet */
   public void init() {
       initComponents();
   }

   public void paint(java.awt.Graphics g) {
       rightEye.paint(g);
   }
```

Test code for Eye.

Testing code for Eye. Note that unlike Code Example 4.12 there is only one line in `init()`. What did init contain in Code Example 4.12?

Once you find and eliminate all the typing errors, you should notice that there's no sign of the Eye. Why not?

d) Debugging

There are many possible reasons. Maybe it's never being sent `paint()`. Maybe it is painted in white. Maybe it's being drawn off the screen. Maybe it is so small you can't see it. Maybe something else is being drawn on top of it. The job of the programmer, at this juncture, is to determine the cause of the problem and fix it. Assuming it is one of the reasons listed above, how could you go about determining which it is? The answer is, use the scientific method. Design and carry out experiments to verify or eliminate each of those hypothetical bugs. Until you determine what is causing the problem, it will be difficult to fix.

You might start by making the Applet window bigger; maximize it and see if the Eye appears. Or, you might push the "grow pupil Button"; do it several times. This assumes that you have modified the event handling code for that Button so that it sends the `growPupil()` method to the Eye. If you haven't added that code yet, do so now.

In my Applet, after I pushed the grow Button several times, I was surprised to see a quarter of a purplish circle expanding from the upper left corner. Having seen this before, I immediately realized that the reason I didn't see anything at first was that the radius, x, and y, were all zero. Do you know why? The default initial value of instance variables is zero (see "Putting it all together - finally!" on page 59).

If you compare Code Example 4.12 (the Applet to test FilledCircle) and Code Example 4.17 (to test Eye), you will notice that `init()` in the former sets x, y, and radius for both FilledCircles and sets the color of the smaller to black; in the latter it does not. Somehow we must specify the location of the Eye and make its pupil black.

There are a number of ways we might set the initial size and location of an Eye. For now, simply add `setX()`, `setY()` and `setRadius()` methods to Eye, and send these messages to the Eyes in `init()`. To `setX()` for an Eye, all you need to do is send `setX()` to both the iris and the pupil. For `setRadius()`, send `setRadius()` to both, but make the radius of the pupil smaller.

A maxim of object programming is for classes to know the minimum. It makes sense for the Applet to control the location of the Eye, and possibly the size. Nevertheless, every Eye will have a black pupil, so the right place to set the color of the pupil is in Eye, not Applet.

1You may have noticed this code (written by Netbeans) in Eye.java.

```
    /** Creates a new instance of Eye */
    public Eye() {
    }
```

This looks like a method without a return type, with the same name as the class. It is called the default constructor, and is invoked when you say `new Eye()`. If there is any initialization code for instances of a class, it goes in the default constructor. So that is where the code to set the color of the pupil to black goes. See Code Example 4.17 for the

Code Example 4.18

```
1     /** Creates a new instance of Eye */
2     public Eye() {
3         pupil.setColor(java.awt.Color.BLACK);
4     }
5
6     public void setRadius(int r) {
7         iris.setRadius(r);
8         pupil.setRadius(r/2);
9     }
10
11    public void setX(int x) {
12        iris.setX(x);
13        pupil.setX(x);
14    }
```

Additional code for Eye.

Line 3: Sets the color of the pupil to black (so it won't be purple).

Lines 6-9: To set the radius of the Eye, set the radius of the iris to the parameter, set the radius of the pupil to half that.

Lines 11-14: To set x for the Eye, `setX()` for both its iris and pupil to the parameter.

code you should add. Do so, then run your program again. If you've made no mistakes, it will display an Eye with a black pupil. Chances are you have made one or more mistakes. If so, figure out what's gone wrong. Don't panic! Just pick up the balls and keep practicing. Try out the buttons. Do they work? Did you write code for each one?

G. Assembling a working Eyes program

Now that you have a working Eye class and a working Applet with buttons to adjust it, accomplishing the task of displaying two of Eyes is fairly trivial. Probably you already know what needs to be done. There are four things, all in the EyeApplet class.

1. Declare the second Eye (at the top)
2. Set the size and position of the second Eye (in `init()`)
3. Display the second Eye (in `paint()`)
4. Resize both pupils (in `actionPerformed()` for the shrink and grow Buttons).

These should all be simple since the code is already there for rightEye.

Make those changes, and test your code. The only thing remaining now is to make the Eye color match yours. You could experiment with changing the RGB parameters on line 3 of Code Example 4.11, but that means you'd have to recompile each time. A more efficient (and fun) technique is to use Netbean's Color Editor (see Appendix ?? Color Editor -- In the Form Editor, select a button, then in Properties click the ... to the right of background, click RGB and slide the sliders).

H. Conclusion

This chapter developed a program to display two eyes the color of the programmer's in an Applet. It did so by designing and implementing a Circle class, extending that to a FilledCircle, and finally building an Eye class that was composed of two FilledCircles. It thus illustrated both mechanisms for code reuse: inheritance and composition. It also illustrated the use of simple Java Graphics and Color plus walked through the process of developing a program incrementally.

A novice programmer would have spent roughly 3-6 hours to work through this chapter; there are so many details that needed to be correct. There is no substitute for spending the time to learn to program. Like juggling, you simply can not learn to do it by reading about it or watching someone else do it. Is the investment of time and energy to gain this skill a good one? Consider what you might do with that time otherwise. If the time would have been spent watching TV or playing video games... odds are you can finish that sentence.

The next chapters will review the material glossed over here in a more detailed fashion. If you choose to continue, see you in the next chapter!

I. End of chapter material

i) New terms in this chapter

HTML - hypertext mark-up language: an embedded command formatting language commonly used for web pages. 63
pixels - picture elements, the smallest drawable part of the output 63
signature - the type, name of a method along with the number of parameter and their types 62

ii) Review questions

4.1 Why are prototypes useful to build first?

4.2 Why is design important?
4.3 What are the first two things to do in design?
4.4 What is a graphics context?
4.5 What message do you send to an Applet to cause `paint()` to happen? Does it have parameters?
4.6 What are parameters for?
4.7 What are the two techniques of class reuse?
4.8 What does pixel mean?
4.9 How many colors (exactly) are possible in Java?
4.10 What are the parameters for `drawRect()`? `fillRect()`? `drawOval()`? `fillOval()`? `drawLine()`? `setColor()`?
4.11 How do you change the size of the Applet (so it stays changed!)?
4.12 What are accessors for?
4.13 How do you fix a bug you can't find?

iii) Programming exercises

4.14 Write the `paint()` method for Circle.
4.15 Write the accessors for Circle.
4.16 Write the Circle class.
4.17 Write the FilledCircle class.
4.18 Create a Target class that is displayed as alternating red and white bands of color. Hint; draw the biggest fillOval first and work in.

Chapter 5: Towards consistent classes

"When facing a tree, if you look at a single one of its red leaves, you will not see all the others. When the eye is not set on any one leaf, and you face the tree with nothing at all in mind, any number of leaves are visible to the eye ..."
Takuan

A. Introduction

Takuan was a Buddhist master who lived in the middle of the second millennium. You may wonder how a 500 year old quote from a Buddhist is relevant to computing; let me explain. Buddhists had been studying human nature for 2000 years at that point; and people haven't changed in 500 years. Our culture (language, technology, education) has changed radically since, but our DNA; our bodies, our brains, our minds are just the same. One of the things Buddhists have studied extensively is attention. They recommend paying complete attention to whatever you are doing, whether it is writing code or washing the dishes (see T. N. Hanh). The resulting focus can change your life.

How is object programming different from procedural programming (from which it grew)? In this context, the leaves on Takuan's imaginary tree are classes. If you are practicing object programming correctly, when you are writing a class, you are thinking of nothing but that class. This provides tremendous power, because you are not distracted and can focus on writing a bug-free class. Once it is written and tested, you can forget what's inside and devote your attention to other matters. This is one of the most important advantages of object programming.

A major constraint on a programmer's ability to think clearly (and thus solve the problems that inevitably arise when programming) is cognitive overhead. If every class is similar, once the pattern becomes familiar, the programmer's cognitive overhead is reduced. This chapter will present a method for writing classes that will be used in every class from now on. There are many styles of programming, and no one can say that one is best for everyone. The style recommended here is simple and consistent; that's enough for now.

This chapter includes more complete explanations of the components of classes. That requires a fair amount of detail and a bit of notation. So, the sections describing class components will be bracketed by sections presenting some of the associated details. Read those detail sections briefly, but return to them in a few days; they are dull, but learning them will save you many hours later. The first introduces the syntax and semantics of Java statements in a more rigorous way.

B. Details I - Statements in Java: syntax and semantics

So far, the code in the book has only used three kinds of Java statements: assignment statements, message statements and return statements. They can be easily distinguished. Every assignment statement has an equals sign, like, `balance = 17;`. Every return statement starts with the word return. Every message statement has a ".", like, `System.out.println("greetings");`, although sometimes the "."s are added automatically by the compiler and do not appear in the code. The good news is there are only about a dozen statements total in Java. You may be wondering what all the rest of the code so far has been composed of; mostly declarations (class, method, and variable). These will be covered in the next section.

i) Syntax and Semantics

Syntax is another word for grammar. It has nothing to do with meaning. The syntax of Java (or any programming language to date) is very simple compared to a natural language like English. The compiler uses what is called a context free grammar to check the syntax and, if it is correct, then converts the source code to byte code (See "The Java Virtual Machine" on page 12). A context free grammar is composed of a set of productions. The syntax of each statement is defined by a single production, as will be seen below. Once you have learned the syntax of each of the elements of Java, the mystery of syntax errors will be mostly dispelled.

Semantics is another word for meaning. In the context of programming, semantics means action (this is sometimes called operational semantics). The semantics of a statement is the action it performs when it is executed.

ii) BNF notation

The productions of a grammar are often represented in ***BNF*** notation, which stands for Backus Naur Form (Backus and Naur were the originators of this representation of grammars). This is a metalanguage, i.e. it is a language which is *about* a language. Since you are not familiar with the Java language, and you are familiar with English, here's an introduction to BNF for a subset of English.

a) BNF for a tiny fragment of English

Consider a fragment of English where every sentence is composed of a noun phrase followed by a verb phrase. This could be written as the BNF production:

<sentence> ::= <noun phrase> <verb phrase>

The "::=" means "is defined as". So, this is read, "A <sentence> is defined as a <noun phrase> followed by a <verb phrase>. Things between pointed brackets are called non-terminal symbols and must be defined somewhere in the grammar. So to complete this grammar we would need definitions of <noun phrase> and <verb phrase>. For now let's say that the only legal verb phrases are "runs" and "jumps". In BNF:

<verb phrase> ::= runs | jumps

The vertical bar (|) means "or". Symbols without brackets, like "runs" and "jumps" are called terminal symbols and must appear literally.

Assume that a noun phrase is either a proper name, or an article followed by a noun. In BNF:

<noun phrase> ::= <proper noun> | <article> <noun>

Let the proper nouns be "Jane", "Dick", or "Spot", the articles, "a" or "the", and the nouns be "cat", or "mouse". In BNF:

<proper noun> ::= Jane | Dick | Spot

<article> ::= A | The

<noun> ::= cat | mouse

All the non-terminals have been defined (i.e they have appeared on the left of a ::=) so the grammar is finished. What language does it generate? In other words, what are all the legal sentences in this language? Here they are, in left to right order:

Jane runs
Jane jumps
Dick runs
Dick jumps
Spot runs
Spot jumps
A cat runs
A cat jumps
A mouse runs
A mouse jumps

Summary of BNF Notation
::= -- is defined as
<x> -- one thing of type x
| -- or
[x] -- optional x
[x]* -- 0 or more x's

The cat runs
The cat jumps
The mouse runs
The mouse jumps

So, there are just 14 legal sentences in the language. Let’s add optional adjectives to make it a bit more interesting.

b) Adding adjectives

The notation for an optional symbol is [optional thing]. So, to allow the two sentences: "The big cat runs", and "The cat runs", the following productions will do:

<noun phrase> ::= <proper noun> | <article> [<adjective>] <noun>

<adjective> ::= big | small | black | white | ferocious

English allows multiple adjectives to modify a noun. If we wanted to allow a sentence like, "The big black ferocious cat jumps", the requisite production would be:

<noun phrase> ::= <proper noun> | <article> [<adjective>]* <noun>

Notice the "*" after the []’s. This means "0 or more repetitions of the symbol in the []’s. Here is the grammar all collected together:

<sentence> ::= <noun phrase> <verb phrase>

<noun phrase> ::= <proper noun> | <article> [<adjective>]* <noun>

<verb phrase> ::= runs | jumps

<adjective> ::= big | small | black | white | ferocious

<proper noun> ::= Jane | Dick | Spot

<article> ::= A | The

<noun> ::= cat | mouse

How many sentences does this grammar generate?

Review the BNF notation summary to be sure you know it, the next sections depend on it.

iii) BNF, Java and adaptive systems

The Java compiler, like any contemporary compiler, is very literal and rigid. It insists that source code match its grammar, symbol by symbol. Any deviation will result in compiler errors; and errors prevent the compiler from producing byte code, and without byte code you can't execute your program.

Syntax error are just details, but they are details that can stop you. If you omit, or misuse a comma in an English paper, you may be corrected by your teacher, or lose points, but your English teacher can still understand your meaning. The compiler, by contrast, will cut you no slack. If a program is missing a semicolon, or has a misspelled word, no matter how many times you compile it, it will still generate an error, and will still not run. In the person/compiler system, the person must make the adjustment, the compiler will not. Fortunately, once you know the BNF, you will at least know what the compiler is looking for.

iv) The assignment statement

The assignment statement, while simple and unprepossessing is the only one that changes the state of a program. Its grammar is shown in BNF 5.1. Every assignment statement

BNF 5.1 The assignment statement

<assignment stmt> ::= <variable> = <expression>;
Semantics 1: Evaluate the <expression> 2: Assign that value to the <variable>

matches this syntax. I.e. every assignment statement is a variable, followed by the ***assignment operator***, followed by an expression, and finally a semicolon. This has several implications: 1) The only thing that can appear on the left of the assignment operator is a variable and whenever you see an assignment operator, you know that whatever is to the left of it *is* a variable, otherwise it won't compile into working code. 2) Only an expression can appear to the right of the assignment operator and anything that appears there must be an expression, for the same reason. 3) Any other syntax is illegal.

What is not specified by the BNF, but is necessary for an error free program, is that the type of the expression must be compatible with the type of the variable. If the types are not compatible a compiler error will occur. The details of compatibility appear in "Expressions" on page 120.

Check which symbols in the following examples correspond to which BNF symbols. Examples:

```
balance = nuBalance;                  // see Code Example 3.2 on page 48
balance = balance - amountToWithdraw; // see Code Example 3.4 on page 50
```

```
currentAccount = account3;          // see Code Example 3.10 on page 56
myColor = c;                        // see Code Example 4.11 on page 82
```

v) The message statement

Computing is information processing. Almost all information processing in a Java program is accomplished by sending messages to objects. As BNF 5.2 shows, every

BNF 5.2 The message statement

<message stmt> ::= <object>.<message>([<actual parameters>]);
Semantics 1: Perform the parameter linkage (See "Parameters (actual, formal, linkage)" on page 104). 2: Execute the method body of the associated method, using the <object> as this.

message statement is an object followed by a period, followed by a message (which has parameters enclosed in parentheses).

The BNF does not specify that that method must be defined in the class the object belongs to (or one of its superclasses). Again, the compiler will catch the error if it is not (see "The mechanics of message sending" on page 138).

Examples:
```
myAccount.setBalance(1234);       // from Code Example 3.3 on page 49
System.out.println("Greetings");  // from Code Example 2.2 on page 22
rightEye.growPupil();             // from Code Example 4.15 on page 85.
```

But, here are some message statements that do not appear to match that syntax; they have nothing to match <object>.
```
initComponents();              // from Code Example 4.9 on page 77
repaint();                     // from Code Example 4.15 on page 85
```

This is because these are shorthand for `this.initCompontents()` and `this.repaint();` The compiler fills in "this" for you (see "Special to Java -- what is this?" on page 115).

vi) How to generate a Null Pointer Exception

One of the most common run-time errors is the Null Pointer Exception. That name is a bit worrisome, but actually descriptive. The way to generate one is to send a message to an object which has been autoinitialized to 0, which, for references (and all Objects in Java are references) is considered to be "null". So, if you send a message, any message, to an object variable before you set it to reference an object, it will always generate a Null Pointer Exception. Like this:

```
class Broke {
       Object theObject;

       Broke() {
           theObject.toString();
       }
```

Every time you have a Null Pointer Exception, it means your program has tried to send a message to an Object that hasn't been initialized (or has been set accidently to null). Every time.

See "Exceptions" on page 350 for more details.

vii) The return statement

BNF 5.3 The return statement

<return stmt> ::= return [<expression>];
Semantics If there is no <expression>, return immediately to where the method was invoked. If there is an <expression>: 1: Evaluate the <expression> 2: Leave the method immediately, returning that value as the value of the message that invoked the method.

Return statements are used to exit a method and can be used to pass information back to the point that the message was sent. If the return type of the method is not void, the compiler will insist that the <expression> exist and be of a type compatible with the type of the method.

Any of the accessors that get values have exactly one line in their bodies, a return statement. As you can see in Code Example 5.1 on page 99, the type int appears before `getBalance()` -- this is the type of the method.

Examples:

```
return balance;                        // from Code Example 3.2 on page 48
return currentAccount.getBalance();  // from Code Example 3.10 on page 56
```

Notice in the second example that the <expression> is a message; whatever its method returns is its value.

C. The basics of classes

As discussed previously, a class declaration defines a template for objects (or instances) of that type. It includes both variable and method declarations. Variables contain information. Methods perform actions; what an object can do depends on what methods are declared and how they are implemented. Now that you have some experience with a few classes, it is possible to gain a more detailed understanding of how they work. A class may have many methods, but this section will only cover the standard methods, accessors and toString() -- on the other hand, these will be described more or less completely.

i) Variables I (state)

Variables store information; the state of an object is determined by the value of its variables. Every variable has a name, a type, and a value. When a variable is declared, only the name and type are required. There are a variety of variables in Java, including:

BNF 5.4 Variable declaration 1

<variable declaration> ::= <type> <identifier>;
Semantics 1: Create a variable of the given <type> with the name <identifier>

instance, class, local, parameter and method variables. In this section only instance variables will be presented.

Examples:

```
int balance;  // see Code Example 5.1 on page 99
String name;
```

Recall the Account class (see Code Example 5.1). Each Account object must keep track

Code Example 5.1

```java
public class Account {
    protected String name = "nobody";
    protected int balance = 1000000;

    Account(){   //empty default constructor
    }

    public int getBalance() {return balance;}

    public void setBalance(int nuBalance) {balance = nuBalance;}

    public void withdraw(int amountToWithdraw) {
        balance = balance - amountToWithdraw;
    }
}
```

The Account class from Chapter 3.
To set the initial balance for every account to $1,000,000. Change is in **bold**.

of the name and balance in a particular bank account. So, the Account class has two variables, balance and name; they must be declared outside of any method. The type of balance is int, the type of name is String.

The alert reader will have noticed that the two variable declarations in Code Example 5.1 do *not* conform to the syntax in BNF 5.4. The `protected` keyword and the assignment are optional and do not appear in that BNF definition. In fact, the BNF for a variable declaration is:

BNF 5.5 Variable declaration 2

<variable decl> ::= [<access>] <type> <identifier> [=<expression>];

Semantics

1: Create a variable of type <type> with the name <identifier> with the <access> defined.

2: If the optional =<expression> is there, perform the assignment statement.

Examples:

```
Account account2 = new Account();    // see Code Example 3.9 on page 56
Bank theBank = new Bank();           // see Code Example 3.12 on page 59
String returnMe = "I am a Circle: "; // see Code Example 4.3 on page 72
protected Color myColor = new Color(100,0,100); // Code Example 4.11 on page 82
protected int balance = 1000000;     // see Code Example 5.1 on page 99
```

In the case of instance variables, the variable is created when the object is instantiated; i.e. if the object is an Account, when the new Account() is executed. If there is an =<expression>, the variable is then initialized to the value of the <expression>, otherwise to zero.

Each instance of a class has a copy of each ***instance variable***; that is why they are called *instance* variables. The balance variable in Account is a good example; every account needs to keep track of a different balance, and thus needs an instance variable do to it.

ii) Methods (control)

The standard methods include two accessors for each variable and a toString() method for debugging. Additionally, most classes have various constructors to make initialization simple, plus other methods which expand its capabilities. This section will present the syntax of method declarations, using examples from accessors you have already seen. Then a tool that writes these methods automatically will be introduced before the rest of the description of methods.

The syntax of every method declaration is a method heading followed by a method body.

BNF 5.6 Method declaration

<method decl> ::= <method heading> <method body>
Semantics A method declaration is not ever executed. So it does not have semantics in the sense of statements. Nevertheless, it does have a meaning, namely: Create a method for the current class with the signature declared in the method heading. The body of the method is executed when the corresponding message is sent.

The method heading can take several forms since the access modifier, the return type, and the parameters are optional; the name, and parentheses are not optional.

BNF 5.7 Method heading

<method heading> ::= [<access>] [<return type>] <identifier> ([<formal parameters>])
Semantics A method heading is never executed. It defines the signature of the method. Constructors do not have a return type. For ordinary methods, if the return type is not void, the method body must end with a return statement whose <expression> has a type compatible with the <return type>

Examples (again, match the code symbols to the BNF):
`void setBalance(int nuBalance)` -- [<access>] omitted, no return, one int parameter
`int getBalance()` -- [<access>] omitted, returns an int value, no parameters
`public String toString()` -- access is public, returns a String, no parameters
`public void paint(java.awt.Graphics g)` -- access public, no return, one Graphics parameter.

The BNF for <identifier> and < formal parameters> are omitted (recall that an identifier is any series of letters, numbers and underscores beginning with a letter). Parameters are addressed below ("Formal and actual parameters" on page 104).

A method body is simply a block statement and a block statement is zero or more

BNF 5.8 Method body

<method body> ::= <block statement>
Semantics Same as the block statement.

statements enclosed in {}s.

BNF 5.9 Block statement

<block statement> ::= { [<statement>]* }
Semantics Execute each statement in the block in order.

a) return types

Every method that is not a constructor must have a return type. Any legal type may be a return type. if nothing is returned, the return type must be declared as void.

b) Accessors

Classes typically store information; other classes need to access that information, both to discover what it is and to update it. The methods that give other objects access to variables inside an object are called accessors. These are also called "getters" and "setters", as they get and set the values of the variables they access.

- **getters - get values**

Every getter is the same, except for three symbols. Compare the two methods in Code Example 5.2; review Code Example 3.2 on page 48 if these seem unfamiliar. The only

Code Example 5.2

```
public int getBalance() {
     return balance;   // return the balance
}

public String getName() {
     return name;   // return the name
}
```

Two getters; for balance and name.

differences are the return type (int or String), the name of the method (getBalance() or getName()), and the variable whose value is returned (balance or name). Both have a return statement as the only statement in the body of the method.

The int return type in getBalance() means that the getBalance() message is an expression of type int and so can appear anywhere an int expression is legal (See "Expressions" on page 120).

- **setters - set values**

Setters too are all the same shape. Compare the two in Code Example 5.3. Both are void

Code Example 5.3

```
 public void setBalance(int nuBalance) {
     balance = nuBalance; // set the balance
 }

 public void setName(String nuName) {
     name = nuName; // set the name
 }
```

setBalance() and setName().

(meaning they return nothing), with one parameter, which is used to set the variable involved. Both have a single assignment statement as the body of the method. They are identical in form, the differences stem from the names and types of the variables being set.

c) Parameters (actual, formal, linkage)

Parameters carry information into methods. The first step in executing a message statement is to perform the parameter linkage. To describe the parameter linkage requires a few new terms.

- **Formal and actual parameters**

There are two varieties of parameters, ***actual parameters*** and ***formal parameters***. Consider Code Example 5.4, which is a copy of Code Example 3.3 along

Code Example 5.4

```
1 class Account {
2 ...
3    public void setBalance(int nuBalance) {
4        balance = nuBalance; // set the balance
5    }
6 } // Account class
7 ...
8
9 public static void main(String[] args) {
10    Account myAccount = new Account();
11    System.out.println("Before balance=" + myAccount.getBalance());
12    myAccount.setBalance(1234);
13    System.out.println("After balance=" + myAccount.getBalance());
14}
```

Actual and formal parameter for setBalance().
Line 3: The formal parameter in the setBalance() definition is int nuBalance.
Line 12: The actual parameter in the setBalance() message is 1234.

with the setBalance code from the Account class. Line 12 sends the setBalance() message to the myAccount object with the parameter 1234. Line 3 is the method heading for setBalance() and defines a parameter of type int. The former (`1234`) is the actual parameter, the latter (`int nuBalance`) is the formal parameter.

One mnemonic for which is which is that the formal parameter is part of the definition of the method and definitions are formal things. Plus, the actual parameter is the value that is *actually* sent along with the particular message that invokes the method.

BNF 5.10 Formal parameters

<formal parameters> ::= <formal parameter> [, <formal parameter>]*
<formal parameter> ::= <type> <identifier>
Semantics Declares one or more local variables which are created when the method is invoked and destroyed when it returns.

In BNF 5.10, the first production means, "<formal parameters> is defined as a <formal parameter> followed by any number of additional <formal parameter>s separated by commas". It may take a bit of thinking to realize why this BNF production generates that. The form is identical for actual parameters.

BNF 5.11 Actual parameters

<actual parameters> ::= <actual parameter> [, <actual parameter>]*
<actual parameter> ::= <expression>
Semantics Each expression is evaluated before the message is sent as part of the parameter linkage.

When there is more than one parameter, the formal and actual parameters are matched up in the order that they appear. The first actual parameter is said to correspond with the first formal parameter; the second, with the second; and so on.

Each formal parameter is a variable declaration, i.e. a type and a name; each actual parameter is an expression of a type compatible (See "Expressions" on page 120) with the corresponding actual parameter. Parameters of both types always appear between parentheses and are separated by commas.

- **Parameter linkage**

With the phrases actual parameter, formal parameter and corresponding parameter understood, the parameter linkage operation (the first step of the semantics of a message

statement) may be expressed fairly succinctly. To perform a ***parameter linkage***, for each parameter:
1. evaluate the actual parameter
2. assign that value to the corresponding formal parameter

What is the difference between this and the semantics of the assignment statement?

Example: Code Example 4.14 on page 84 is the shrinkPupil method for the Eye class. Its body is a single statement:

```
        pupil.setRadius(pupil.getRadius() - 2);
```

Answer these questions before reading the answers. It's okay to look back to discover the answers if you know you know, but can't remember.

What type of statement is this?

What message is sent?

What object is it sent to?

What type is that object (i.e. what class is it an instance of)?

What is the parameter?

Describe the semantics of executing this statement.

If you have no idea what most of the answers are, don't feel bad, there is a tremendous amount of detail piling up here, and it's all interrelated. Maybe it's time to take a break and then, when your head is clear and you can focus again, reread the beginning of the chapter. You do want to learn this stuff, right?

```
        pupil.setRadius(pupil.getRadius() - 2);
```

Assuming you could answer most of them, here are the answers. This is a message statement. It sends the setRadius message to the pupil object (which is a FilledCircle, See "The Eye class" on page 83), with the actual parameter `pupil.getRadius() - 2`.

To execute that statement, Java first performs the parameter linkage, then executes the message body. The parameter linkage has two parts: First, evaluate the expression, second, assign the value thus derived to the formal parameter. To evaluate a message statement, Java executes the message and uses what is returned as the value, so, the first thing that happens is `pupil.getRadius()` is executed.

The getRadius() method has no parameters, so the action is simply to execute the method body, which is one statement (`return radius;`), which sends back the value of the radius variable from inside the FilledCircle named pupil. Assume that its value is 50.

To complete the evaluation of `pupil.getRadius() - 2` Java subtracts the 2 from the 50 that came back from getRadius, yielding 48. That is then assigned to the nuRadius parameter in the setRadius() method. The body of that method is the assignment statement `radius = nuRadius;` whose execution stores the value of nuRadius (namely, 48) back in the radius variable inside the pupil object.

That's a lot of detail! The good news is that you don't actually have to think about that when you're programming, once you understand the basics; it all disappears into expertise. You just think something like, "Hmmm, shrinkPupil needs to make the pupil a little smaller. How about 2 pixels? Okay. `pupil.setRadius(2 less than it is);` Hmmm, what is the radius now? Oh yeah, getRadius(). So, `pupil.setRadius(pupil.getRadius-2);."` -- and type that.

d) toString()

This is the standard debugging output routine for every class. Its signature, `public String toString()`, conveys several pieces of information to a Java savvy reader: 1) the name of the method is toString, 2) it has no parameters, and 3) it returns a value of type String. When writing a class, you can make toString() return any information you choose, so long as it is in a String. The most obvious information to include is the type of the object (i.e. what class it instantiates) and the current value of its variables.

In Code Example 4.3 on page 72, the body of the method has 5 statements. The first is a

Code Example 5.5

```
1    public String toString() {
2        String returnMe = "I am a Circle: ";
3        returnMe += "\tx=" + getX();
4        returnMe += "\ty=" + getY();
5        returnMe += "\tradius=" + getRadius();
6        return returnMe;
7    } // toString()
```

toString() for the Circle class

Line 2: Declare the String variable to return; set it to "I am a Circle: "
Line 3: Paste a tab (\t) onto it, followed by "x=" and the value of x.
Line 6: Return that whole String as the value of toString()

variable declaration statement with initialization to the String "I am a Circle: ". This variable, returnMe, is not an instance variable, because it is declared in the body of a method it is a method variable (See "local variables: parameters, method variables, and for loop variables" on page 126) The next three each concatenate: 1) a tab, 2) an instance variable name, 3) an equals sign, and, 4) the value of that variable, obtained through its accessor. The last statement is a return statement which sends back the value of returnMe as the value of the toString() message wherever it was sent.

D. The ClassMaker tool

i) Motivation

The process of programming is not simple, nor is it easy. Even after you understand the concepts, there is a host of details that must be attended to before a program can be completed. Many of the details a programmer must deal with stem from the rigidity of the software, which stems from the mindlessness of current computers.

On the other hand, computers are incredibly useful owing to their blinding speed and disregard for endless repetition. In this regard, they are a pretty good complement to human skills/proclivities with the appropriate software; tasks that always involve the exact same actions can be mechanized. Robot factories good!

The large majority of programming time is spent selecting, designing, and implementing classes. Although the very simplest programs may have only one or a few classes, any substantial project has a number of classes. Once you have mastered accessors, constructors and toString(), writing them is less than exciting; and, as you have seen above, they have the same format in every class. That means software could be written to generate them automatically.

The author has written a ClassMaker class that inputs the shell of a class, with just the name and a list of variables, and produces the constructors, accessors, and toString() from a single button press. It is publicly available for your benefit, but you are only allowed to use it after you can write those methods correctly from memory. The reason is that the mechanics of accessors, constructors and toString() are those of the majority of all methods and until you understand how to write them, everything else will be hopeless. Hopeless. Get it? Don't use the ClassMaker until you can write these methods yourself without peeking. The easy way, is not easy in the long run.

ii) ClassMaker input and output

Once you are able to write constructors, accessors, and toString() from memory (in my classes, usually everyone can do it by the fourth or fifth week; it just takes practice), you are ready to use the ClassMaker. It takes some of the drudgery out of creating a new class, and any repetitive task that can be automated should be.

You can find the ClassMaker at:
http://www.willamette.edu/~levenick/classMaker/makeClass.html
Code Example 5.6 is the input to the ClassMaker to produce the Account class in Code Example 5.7. Note that it produces the default constructor, accessors for both variables,

Code Example 5.6

```
class Account {
    int balance;
    String name;
}
```

Input to the ClassMaker

and a toString() method that displays the values of both variables by using their constructors. You can copy and paste from the webpage into the Netbeans editor window.

Code Example 5.7

```java
public class Account {

    protected int balance;
    protected String name;

    public Account(){}   //empty default constructor

    public Account(int balance, String name) {  //initializing constructor
        this();
        this.balance = balance;
        this.name = name;
    }

    public int getBalance() {return balance;}
    public String getName() {return name;}

    public void setBalance(int balance) { this.balance = balance;}
    public void setName(String name) { this.name = name;}

    public String toString() {
        String returnMe = "I am a Account: ";
        returnMe += "\tbalance=" + getBalance();
        returnMe += "\tname=" + getName();
        return returnMe;
    } // toString()
}  // Account
```

The Account class produced by the ClassMaker

Pretty groovy, eh?

Read each line of code in it Code Example 5.7. Become familiar with it; that means be aware of: 1) the name of the class, 2) each variable (its name and type), and 3) each method (its heading and body). Do that now. Notice anything strange? The setters are different than those earlier in the text. They do exactly the same thing as the previous setters in a different way; the details are in "Special to Java -- what is this?" on page 115.

As an exercise, recreate the ATM Applet from scratch using the classmaker to create the frameworks for Account, and Bank. Notice how much of the code is written for you. Do the same for the Eyes Applet. Expect this to take several hours. Sorry. It’s good to practice; when you are trying to learn a new programming environment or language, you must repeat very simple tasks over and over until you can do them effortlessly without running into problems/perplexities each time -- then you will know you have mastered the process.

You should expect to discover that it's much easier to do things the second time, plus this time what you're doing will make more sense. Once the mechanics of using Netbeans becomes more of less automatic you will have more cognitive capacity for the problems that will inevitably arise while programming.

E. Constructors

A peculiarity of OOP is that often much of the functionality of a class is subsumed by the constructors (or constructor chains). This statement will make more sense once you have had some experience with constuctors.

Syntactically, a constructor declaration is like any other method declaration, with two differences. First, there is no return type (and no value can be returned). Second, its name must be the same as the class it is in.

Executing a constructor is just like executing any other method, but it happens automatically when you create an object of that type with matching parameters.

i) Default constructors

A default constructor has no parameters. When the following line is executed:

```
        Account myAccount = new Account();
```

these three things happen.

1. The new Account object is created, i.e. space is allocated for all its variables.
2. The default constructor is executed.
3. The newly constructed Account is returned and stored in the myAccount variable.

There are thus two ways to automatically initialize the value of an instance variable; either use an assignment with the declaration `int balance=1000000;`, or insert an assignment statement in the default constructor:

```
  public Account() { balance = 17; }
```

Assuming you did both, what would the initial value of balance be?

ii) Account class including a constructor with parameters

Usually, when you create objects, you wish to give them particular values. With only a default constructor, you are forced to first create the object and then set its values with accessors. These three lines can be compressed into one if there is a constructor that is passed initial values for the instance variables, as shown in Code Example 5.8. For a

Code Example 5.8

```
      Account myAccount = new Account();
      myAccount.setName("Frodo");
      myAccount.setBalance(1000000000);

   or

      Account myAccount = new Account("Frodo", 1000000000);
```

Creating and initializing an Account using a default constructor

Circle, one line replaces four as Code Example 5.9 shows. Obviously, it is less keystrokes

Code Example 5.9

```
      Circle myCircle = new Circle();
      myCircle.setX(200);
      myCircle.setY(100);
      myCircle.setRadius(77);

   or

      Circle myCircle = new Circle(200, 100, 77);
```

Creating and initializing an Account using a default constructor

and clearer to use the initializing constructor, and the ClassMaker writes it for you.

iii) Eye/FilledCircle/Circle classes including a constructor with parameters -- and simplifications appertaining thereunto

If we rewrite the Eye Applet using initializing constructors, there are a number of savings. Most obvious is where we create and initialize the Eyes . After using the default

Code Example 5.10

```
Eye rightEye = new Eye();

...

rightEye.setX(600);
rightEye.setY(100);
rightEye.setRadius(100);

 vs...

 Eye rightEye = new Eye(600,100,100);
```

Creating and initializing an Eye using an initializing constructor

constructor, the position and size must then be initialized; three lines of code in initComponents(). With the initializing constructor those three lines disappear; see Code Example 5.10. Better, by adding the color to the initializing constructor the color can be initialized when the Eye is created as well; Code Example 5.11. In this example, the

Code Example 5.11

```
 Eye rightEye = new Eye(600,100,100,new Color(200, 177, 200));
```

Creating and initializing an Eye with color using an initializing constructor

initializing constructors for both Eye and the built-in class Color are used. The fourth parameter to new Eye() is the constructor for Color. You will recall that the parameter linkage mechanism first evaluates each actual parameter and then copies each value to the corresponding formal parameter. To evaluate the Color constructor, a new Color object is instantiated and then passed to the Eye constructor (which passes it on to the

FilledCircle object that is created in it, see Code Example 5.12). This is an example of

Code Example 5.12

```
1  import java.awt.*;
2
3  public class Eye {
4
5     protected FilledCircle iris;
6     protected FilledCircle pupil;
7
8     public Eye(){}   //empty default constructor
9
10    public Eye(int x, int y, int radius, Color myColor) {
11        this();   // invoke the default constructor
12        iris = new FilledCircle(x,y,radius, myColor);
13        pupil = new FilledCircle(x,y,radius/2, Color.black);
14    }
15
16    public FilledCircle getIris() {return iris;}
17    public FilledCircle getPupil() {return pupil;}
18
19    public void moveRight() {
20        iris.moveRight();
21        pupil.moveRight();
22    }
23
24    public void paint(java.awt.Graphics g) {
25        iris.paint(g);
26        pupil.paint(g);
27    }
28
29    public String toString() {
30        String returnMe = "I am a Eye: ";
31        returnMe += "\tiris=" + iris.toString();
32        returnMe += "\tpupil=" + pupil.toString();
33        return returnMe;
34    } // toString()
35}  // Eye
```

Eye class, modified

Line 12: Pass along the Color, created in the actual parameter of the Eye constructor, as a parameter to the FilledCircle constructor.

what this section mentioned at the start, namely the phenomenon that in object programming, much of the work can be migrated to the constructors.

iv) Special to Java -- what is this?

a) this

The reserved word "this" has a special meaning in the context of an instance method, it is the object which was sent the message that caused this method to be executed; or, shorter, the current object. In the context of a constructor, "this" is the object that is being constructed.

The code written by the ClassMaker uses this to access instance variables. Compare the two setters in Code Example 5.13, which are copied from Code Example 5.4 and Code

Code Example 5.13

```
1    public void setBalance(int nuBalance) {
2        balance = nuBalance;
3    }
4
5    public void setBalance(int balance) {
6        this.balance = balance;
7    }
```

Two versions of the getBalance()

Both set the instance variable balance to whatever value is passed through the parameter. The second uses this. so the value of the parameter is stored in the instance variable instead of just being copied into itself.

Example 5.7. They both do exactly the same thing, namely assign the value of their parameter to the instance variable named balance. In the first version, the name of the parameter is nuBalance, so line 2 assigns the value of that parameter to the instance variable balance as desired. In the second version, the parameter is named balance, just like the instance variable. Thus, in the body of the second setBalance(), there are two different variables, both with the name "balance" If the programmer, without thinking, attempted to set the instance variable named balance to the value of the parameter named balance by typing `balance = balance;` on line 6, it could cause a hard to find bug. When there are two variables with the same name defined in the same place, Java uses the one that is defined the closest (actually the one defined in the nearest enclosing scope, see "Variables II (varieties and scope)" on page 126). In this case the parameter balance is defined closer (looking up the code from line 6), so it is used both times (it is said to ***shadow*** the instance variable). So, the value of the parameter balance is retrieved and stored back in the parameter balance, leaving the instance variable balance unchanged. To specify the instance variable balance, use `this.balance`.

The code the ClassMaker made for Circle included an initializing constructor, shown in Code Example 5.14. The parameters x, y, and radius, are the same as the instance

Code Example 5.14

```
1  public class Circle {
2
3     protected int x;
4     protected int y;
5     protected int radius;
6
7     public Circle(){}   //empty default constructor
8     public Circle(int x, int y, int radius) {   //initializing constructor
9         this();   // invoke the default constructor
10        this.x = x;
11        this.y = y;
12        this.radius = radius;
13    }
```

Circle class

Note the use of "this" in the initializing constructor.
Line 9: invoke the default constructor; this() is the default constuctor
Lines 10=12: set the three variables. this.x is the instance variable x

variables, so it uses this to store the values in the instance variables.

b) this()

If there are initialization tasks that are being performed for every new instance of an object, they should be done in the default constructor. That way, later, when you (or other people) add additional initializing constructors, as long as they invoke the default constructor, the functionality of all the constructors can be preserved even when someone subsequently alters the default constructor. To invoke the default constructor, in another constructor, you say, this(). But it must be the first line of the constructor body; otherwise it will not compile.

Wait! What was all that in the previous paragraph? It seemed to be about avoiding possible future problems if someone added initializing constructors and then after that someone else changed the default constructor. If all you're doing is trying to learn to program and writing tiny little programs that are just going to be discarded, *who cares*?! Okay, right, of course not. Like much of Java, this only makes any appreciable difference when you are doing something big and complicated. This is what makes learning Java a bit difficult at first. If you have the feeling it's more complicated than it needs to be to accomplish simple tasks, you are exactly right.

It's a bit like if you want to build a little, simple bird feeder and your friend who is a machinist says, "I've got just what you need.", and opens the door to a machine shop as big as a basketball arena, packed full with numeric control machines, whirring and spinning ominously. And all you really need is a hand saw and a hammer. Overkill. On the other hand, if you were one day planning to build something complicated, like perhaps a Mars rover, or a better cell phone, or... you name it; then you couldn't possibly do it with the hammer and saw.

So, don't worry about those details right now, just be aware of them, so if sometime you run into this() you won't be completely flummoxed. By the way, the three programmers, the first who wrote the class originally, the second who modified the initializing constructor, and the third who subsequently modified the default constructor, might all be the same person at different times. They might all be you.

F. Details II

i) Types

In Java, a type is either a primitive type, a built-in class or a user-defined class. Some types you have worked with include int, String, Applet, Account, Circle and Eye. The first of those is a primitive, the next two are built-in and the last three user-defined. There are several other primitive types, many built-in classes, and potentially infinitely many user-defined classes.

a) Primitive types

The primitive types are either numeric or non-numeric. There are only two non-numeric primitive types: char and boolean; these represent character and logical values, respectively. The numeric types are either, whole numbers, roughly like the integers (long, int, short, byte); or decimals, roughly like real numbers (double, float). But there are infinitely many integers and uncountably infinite reals, whereas every primitive Java type is represented in a limited amount of space and so can only take on a finite number of values. The primitive types, along with their possible values and operators therefor appear in Table 5.1.

Table 5.1: types, values, operators

type	range of values	operators
long int short byte	$-2^{63} <= x <= 2^{63}-1$ $-2^{31} <= x <= 2^{31}-1$ $-2^{15} <= x <= 2^{15}-1$ $-2^{7} <= x <= 2^{7}-1$	+, -, *, /, %
double float	$-1.8*10^{308}<x<1.8*10^{308}$	+, -, *, /
boolean	{false, true}	!, &&, ||
char	any keyboard character	none

For most applications ints will work fine for whole numbers and doubles for fractional numbers. There is no reason to use float, short, or byte, unless you discover you are out of memory; and that never happens (What never? No, never! What never? Well... hardly ever!). If you tried to count all the people on the planet, or keep track of the national debt, ints are too small, as 2^{31} is only a little more an 2 billion. Fortunately longs would work just fine for those. For unlimitedly large numbers there is the BigNumber class.

b) Numeric types, representation: bits, bytes and powers of two

Some people have the idea that computing is all about bits and bytes, zeros and ones; and it is, underneath (just as life is all about chemistry, molecular machinery, underneath). Modern computing deals very little with bits and bytes, but there are times when you need to understand them a bit. One of those times is if you want to understand how numbers are represented in Java and why they act the way they do.

The range of values for the type int is shown in Table 5.1 as $-2^{31} <= x <= 2^{31}-1$-- this has several implications and an informative cause. First, it means that if you need to store numbers larger or smaller than that, you must use another type, in this case, long. Second, if you add one to $2^{31}-1$, instead of getting 2^{31} as you might expect, you get negative 2^{31}instead! Try it for yourself. Set a variable to 2 billion (2000000000) and then add it to itself and print the result. How to do this? Type these lines into a main() method, or if your Applet is still on the screen, into `init`().

```
        int big = 2000000000;
        int bigger = big + big;
        System.out.println("2 billion + 2 billion=" + bigger);
```

On my machine this code it prints:

```
2 billion + 2 billion=-294967296
```

Which is certainly *not* 4 billion! Why does this happen? The explanation stems from the representation of ints.

An int is represented in Java as four bytes. A byte is 8 bits. A bit is the smallest possible unit of information; it has only two values, 0 or 1. So, a bit is just enough information to distinguish "yes" from "no". Four bytes have 8+8+8+8 bits, that’s 32, so an int can take on $2^{\wedge 32}$ different values (if you don’t know why, reread "Problem Solving Principle #1 - Build a prototype" on page 7). Half those values are used for positive numbers (zero on up), the other half for negative numbers; half of $2^{\wedge 32}$ is $2^{\wedge 31}$. Since there is no negative zero (i.e. the least negative number is -1) the smallest number is $-2^{\wedge 31}$, whereas zero is the smallest non-negative number, so that only leaves $2^{\wedge 31}$-1 other positive numbers.

The types, long, short and byte, have eight, two and one byte, respectively (that’s 64, 16 and 8 bits); if you look at Table 5.1, the ranges correspond perfectly to that.

c) Arithmetic operators

Table 5.1 shows four operators for doubles, the ordinary arithmetic operators, +, -, *, and /. They work just the way you would expect; when applied to two double operands, they yield a double value. Int operators when applied to two ints yield an int value. There are two int division operators, / and %. Int division is the division you might have learned in third grade, where 7/3=2 with a remainder of 1. The / operator, applied to two ints yields the number of times the second goes into the first evenly, 7/3=2. The % operator yields the remainder, 7%3=1.

d) Mixed expressions

When the two operands of an arithmetic operator are an int and a double, the int is converted to a double and double operator is used. As you would guess, a double value results. Thus, 2*2 evaluates to int 4, but 2*2.0 evaluates to double 4.0.

The reason the int is automatically converted to a double and not vice versa is that the range of doubles is so much greater than that of ints. There are many doubles which simply cannot be represented as ints, but every int can be represented exactly as a double.

ii) Expressions

All information in a Java program has a type. Information exists in both variables and expressions. Like a variable, an expression has a type and a value; unlike a variable, it does not have a name. Expressions are not declared and sometimes it is not obvious what their type is.

In the BNF thus far, expressions have only appeared in two places: on the right of the assignment operator and as actual parameters. That means that anything appearing either on the right of an assignment operator or as an actual parameter must be an expression syntactically.

Examples:
Here are some int expressions.
1. 17
2. 1 + 1
3. 1 + 2 * 3
4. 1 + 2 - 3 / 4 * 5
5. (1 + 2 -3/4) * 5

The values are: 17, 2, 7, 3, and 15. The first three should be obvious. The value of 17 is 17, 1+1=2, and everyone knows that you do multiplication before addition. The values of the fourth hinges on how integer division works. This is because the / operator does integer division, discarding any remainder. This can be the cause of very subtle bugs, as will be seen below. So 3/4 = 0, 0*5=0, 1+2=3, and 3-0=0.

The fifth expression uses parentheses to cause the multiplication to happen last, so (3+0)*5=15.

a) Precedence of operators

In an expression with multiple operators, the question arises, "Which operator is applied first?". In other words, which operation precedes which others? You are already familiar with precedence, since you know that 1 + 2 * 3 is 7. In ordinary arithmetic multiplication precedes addition. In computing, one says, "* has higher ***precedence*** than +"; it means just what you expect, that without parentheses, multiplication happens before addition.

Table 5.2: Precedence of operators

Higher precedence operators are higher in the table
.   -- the message dot
(cast)
!, unary -
*, /, %, &&
+, -, ||
<, >, <=, >=, ==

The safest rule is, if you're worried that precedence is a problem, use parentheses.

b) BNF for expressions

You may have noticed that the BNF for expression was missing; or not, whatever. Here it is. Notice that this is a recursive definition (if you've forgotten the term recursive, see

BNF 5.12 expression

```
<expression> ::= <constant>
               | <variable>
               | <message expression>
               | <expression> <binary operator> <expression>
               | (<expression> )
               | <unary operator> <expression>
```

A recursive BNF production.

"What classes will we need? What will they do?" on page 40); it can generate arbitrarily complex expressions. Binary and unary operators take two and one expressions as

BNF 5.13 binary operators

<binary operator> ::= + | - | * | / | % | && | || | < | > | <= | >= | == | !=
&& is and || is or; these are boolean operators <, >, <=, >=, ==, != are relational operators, they compare their operands and yield a boolean expression. The operator == is equals, != is not equals

operands, respectively . Notice that expressions may be arbitrarily complex.

BNF 5.14 unary operators

<unary operator> ::= - | !
- is minus, as in -17, ! is not, as in !(x>100)

c) Expressions compatible with a type

In an assignment statement, the expression on the right of the assignment operator must be compatible with the variable on the left. The same is true of actual and corresponding formal parameters.

The simplest form of compatibility is identity. I.e. an int expression is compatible with an int variable -- you can always assign the value of an expression to a variable of the same type.

For now, all you need to know is that expressions of type int are compatible with double variables, so it is legal to assign an int value to a double variable (or to use an int expression as an actual parameter corresponding to a double formal parameter), but the reverse is not true (See "Mixed expressions" on page 119). Expressions of type completely unrelated to a variable's type can never be compatible; you can never assign a char or a String to an int, or an int or a double to a String. But, sometimes you can convert them by hand.

d) Converting one type to another

There are a number of different techniques to convert one type to and another when they are incompatible.

- **String to int**

When input comes from a TextField it is always a String (the signature of getText() is `public String getText().` The method that converts a String to an int is Integer.parseInt(String). It was illustrated in "When the user hits enter, get the withdrawal amount" on page 44. There is a similar method for doubles.

- **Object to String**

Any Object can be converted to a String using toString() -- but, you already knew that!

- **int to String**

An int (or any primitive type) can be converted to a String by concatenating it to the empty String, "". Like this ""+17 becomes "17".

- **Casting**

It is possible, in some situations, to force an expression to be a particular type. When you cast an expression, you are essentially saying to the compiler, "I know more than you; do what I say here!". The next section illustrates casting.

e) An example - random Circles at random locations

Let’s say you decide to create a bunch of Circles at random locations and sizes and display them. There is a random number generator that you can access by saying ***`Math.random()`*** -- it returns a random number in the interval [0,1) whose type is double. To place a Circle at random requires random x and y values. To make it a random size requires a random radius value. Assume you want Circles with centers in a square 500 pixels on a side with radii up to 200 pixels; then you would need two random ints between 0 and 499, and one between 1 and 200. Since random() provides doubles in [0,1) you need to map from that small interval to the larger ones. The best way to do that is with a method (so as not to have to write the conversion code over and over - and if it turns out to have a bug, you will only have to fix it one place... plus, it might be useful later).

It is easy to perform this mapping, simply multiply. 500*0 = 0 and any number < 1 *500 is less than 500. So the method might look like Code Example 5.15.The only problem is,

Code Example 5.15

```
int rand(int max) {
    return Math.random() * max;
}
```

A method to return an int between the parameter, max, and 0 -- a type error.
This method will not compile because the expression has type double, but the method heading declares the return type as int

that when you multiply a double by an int you get a double. Casting a double as an int simply truncates anything after the decimal point, so the obvious solution is to cast that expression as an int, see Code Example 5.16. This compiles, but always returns zero. The

Code Example 5.16

```
int rand(int max) {
    return  (int) Math.random() * max;
}
```

int rand() with a cast to int -- a precedence error.
Now this method compiles, but always returns 0, because the precedence of a cast is higher than anything except the message dot.

way to understand why is to realize that *, (int), and the dot after Math, are all operators and will be applied in the order of precedence. So, first the random() message will be sent, returning a value in [0,1), then the cast, (int) will be applied, converting the value to an int 0 (through truncation), finally the multiplication will result in 0.

As always, the way to fix precedence problems is with parentheses, as shown in Code Example 5.17. Insert this method into your revised EyeApplet class, and replace the body

Code Example 5.17

```
int rand(int max) {
    return (int) (Math.random() * max);
}
```

Parentheses override precedence for a correct int rand()

The parentheses cause the multiplication to happen before the cast, so everything is copacetic.

of paint() with the line:

```
        (new Circle(rand(500), rand(500), rand(200)).paint(g);
```

This will draw a different Circle each time you resize, or drag the window, or push a button in the Applet. Try it. When it works, make 5 copies of that line as the body of paint() and execute that.

f) Random FilledCircles

If you feel like playing a bit more before continuing on with this endless progression of detail, try this. Create and display FilledCircles instead of Circles. The FilledCircle constructor requires a fourth parameter, of type Color. To generate a random Color, simply give it three random values in the range [0,255] as parameters, as shown in Code Example 5.18. Add this method to your EyeApplet.

Code Example 5.18

```
Color randColor() {
    return new Color(rand(256), rand(256), rand(256));
}
```

randColor() returns a random Color

Creates and returns a color with random RGB values between 0 and 255.

Then, modify paint() so it contains:

```
(new FilledCircle(rand(500), rand(500), rand(50), randColor())).paint(g);
```

as many times as you want random FilledCircles displayed.

If you wanted 20 FilledCircles, you could copy that line 20 times; or, you could write this for loop:

Code Example 5.19

```
for (int i=0; i<20; i++)
 (new FilledCircle(rand(500), rand(500), rand(50), randColor())).paint(g);
```

A loop to create and display 20 random FilledCircles

The details of this loop will be explained in Chapter 8, but for now just think of it as "abracadabra". Try it with 100 FilledCircles (yes, change the 20 to 100). Or 1000. Does it slow down much?

iii) Variables II (varieties and scope)

There are five different varieties of variables. These include instance variables, class variables, parameters, method variables and loop variables. These will be addressed in order of how often they have appeared in this text so far. They differ in where they are stored, how they are initialized, how long they exist, and where they are visible. This last is referred to as their ***scope***.

- **instance variables**

By far the most common variables you have seen so far, and will encounter in object programming are instance variables. See "Variables I (state)" on page 98 for details. They are declared outside of any method and are visible everywhere in the class. They may include an = <expression>, to initialize them, but if not, they are initialed to zero when the instance is created (before the constructor is executed).

- **local variables: parameters, method variables, and for loop variables**

There are three kinds of local variables. Unlike instance variables, their scope is the method or loop in which they are declared, and only exist while it is being executed.

Formal parameters have the form of variable declarations, namely <type> <identifier>. They are visible in the method they exist in. Their values are provided as part of parameter linkage when the method is invoked, and cannot generally be determined at ***compile-time***. There are many examples in the text so far and details are in "Parameters (actual, formal, linkage)" on page 104.

Method variables are declared within the body of a method. They have appeared in Code Example 3.11 on page 57 (Bank theBank), Code Example 5.4 on page 104 (Account myAccount), and Code Example 5.5 on page 108 (String returnMe). They exist only in the body of the method and must be initialized when they are declared.

The only for loop variable was in Code Example 5.19 on page 126, (int i). Loop variables only exist within the loop they are declared in, and must be initialized when declared.

- **class variables**

So far, no class variables have been used. They are not used very much in elementary programming; some people program for years and never use them. They are useful in certain situations though and you might run into one somewhere. Syntactically, class variables are exactly like instance variables, except they have the keyword ***`static`*** in front. Class variables do not belong to any instance, but instead to the class -- hence the name. They are used when there is information that must be accessible from every instance, but which does not belong to the instances. A class variable exists as long as the program is running and is visible from every instance.

Table 5.3: Variables - types, scopes and initialization

Variable type	Scope	Initialization
instance variable	entire class	auto, to 0 or by assign-ment
formal parameter	body of method	by parameter linkage
method variable	body of method	must assign initial value
loop variable	body of loop	must assign initial value
class variable	all instances	like instance

- **Example - Serial numbers**

Some people, when they first start object programming feel somewhat uncomfortable about having many nearly identical objects of the same class, and would like to be able to keep track of which is which. One easy way to distinguish between nearly identical objects in the world, is to attach a serial number to each one. The first one is number 1, the second number 2, and etc. The same may be done with Java objects. Here’s how.

Each object must keep track of its serial number, so there must be an instance variable; e.g. `int serialNumber;` -- this will be set to 1 for the first object instantiated, 2 for the second, etc. There must also be a class variable to keep track of the next serial number to be assigned; like this: `static int nextSerialNumber=1;`

Then, in the default constructor, there must be one line added.

```
     serialNumber = nextSerialNumber++;
```

This is shorthand for

```
     serialNumber = nextSerialNumber;
     nextSerialNumber = nextSerialNumber + 1;
```

The first line assigns the current value of the class variable nextSerialNumber (1 the first time) to the instance variable serialNumber. The second line increments the class variable nextSerialNumber.

That's all it takes. To see that it works add this line as the second in toString():

```
  returnMe += " my serial number is: " + serialNumber;
```

Modify the Circle or Account class (use the one with the main() method to make your task simple), and test to see that this works.

iv) Conventions

There are a number of conventions which are not required by the compiler, but that make it much easier to program. These are entirely arbitrary, but are pretty much standard in the industry.

a) Naming conventions

Identifiers should convey information; they should tell what they are or what they do. This includes class names, method names and variable names. Typically, variables are nouns, methods are verbs. Accessor names start with get or set and then the variable being accessed, e.g. int balance; getBalance() and setBalance().

b) Case conventions

Class names begin with upper-case letters. Instance names begin with lower-case letters. The second and any subsequent English word in an identifier starts with an upper-case letter. Constants are all upper-case.

c) The importance of good names

Descriptive names can make the difference between being able to debug your program and not. The reason is simple, cognitive capacity. People can only keep in mind around 5 things at once. If you name your variables that mean balance and radius, Frank and Ernestine, then you are squandering 2 of your precious 5 on remembering which means which. When the program is extremely simple, this is not a big problem, but if it is just at the limit of the programmer’s capacity, this could lead to disaster.

G. Recapitulation

As mentioned in the first chapter ("Learning to program" on page 13) there are a few dozen concepts that must be understood, at least vaguely, before a person can program in Java. Most of them have been covered in this chapter (if you’re feeling a bit overwhelmed, please be patient, it gets easier). Here’s a rough list; this would be suitable to read every night after programming until it is all very obvious. It appears, first as a list, then with explanations. When you can remember the explanations by looking at the list, then you can stop looking at it.

i) Information

a) types
b) values
c) variables
d) expressions

ii) Language Elements

a) classes
- variables
- methods
- constructors

b) objects
c) statements
d) identifiers
e) methods/messages
- signatures
- parameters (formal/actual)
- parameter linkages

f) syntax
g) semantics

iii) Process

a) editing
b) compilation
c) execution
d) debugging
e) prototyping

Here’s that same list with some explanatory text.

iv) Information

a) types

All information in a Java program has a type. There are primitive types, built-in classes and user-defined classes. It is possible to change types by casting.

b) values

Expressions have values. To compute the value of an expression it is evaluated. Every type has a range of legal values.

c) variables

Variables hold information. Every variable has a name, a type and a value. A variable only holds one value at a time. There are five different kinds of variables: instance, parameter, method, loop, and class.

d) expressions

Variables and constants may be combined in arbitrarily complex fashion to form an expression. Syntactically, expressions appear to the right of assignment operators and as actual parameters (Thus far. Later on you will see them other places.).

v) Language Elements

a) classes

A class is a template for objects of that type. It includes both variables and methods. Every object of that type has all the instance variables and can use all the methods declared in it.

- **variables**

Every instance of a class has its own copy of each instance variable. All instances share access to class variables which are stored in the class itself (interestingly, classes are also objects, they are instances of the class Class).

- **methods**

Methods have a heading and a body. The heading specifies the type, name and parameters of the method. The method body is a single block statement, which is a pair of {}s around a series of statements. To execute the method, Java executes each of those statements in order.

- **constructors**

Constructors are typeless methods with the same name as the class that are executed when a new object of that class is constructed.

b) objects

Objects (also called instances) of a particular class are created by saying

```
new ClassName();
```

This is referred to as instantiation. When an object is instantiated, the constructor corresponding to the signature of the new message is executed.

c) statements

Statements are what accomplish most of the action when a program executes. So far these statements have been presented: assignment statement, return statement, block statement, and message statement.

d) identifiers

Identifiers are Java names. They start with a letter, and are composed only of letters, digits and underscores.

e) methods/messages

When you send a message to an object, it invokes the method in that class with the same signature. First it performs the parameter linkage, then executes the method body.

- **signatures**

The signature of a method is it’s access type, return type, name and parameter types.

- **parameters (formal/actual)**

The parameter in the method declaration is the formal parameter, the one in the message is the actual parameter.

- **parameter linkages**

Each actual parameter is evaluated and its value copied to the corresponding formal parameter

f) syntax

The syntax of Java is defined by a set of BNF productions. Any source code not matching this grammar symbol for symbol is deemed to have compiler errors.

g) semantics

The semantics of a statement is the action it performs when it is executed. An experienced programmer has internalized the semantics of enough of the constructs of the language that solving routine problems is easy.

vi) Process

a) editing

Editing is when the programmer is inputting or changing source code (classes).

b) compilation

Compilation is when the compiler is checking the syntax of a class. Errors at this stage are compile-time errors and are either lexical or grammatical. Lexical errors happen when the compiler does not know what an identifier means; the most common causes are forgetting to declare variables, or typos. Grammatical errors occur when the syntax of the source code does not match the BNF description of the language precisely.

After verifying the syntax of a class, the compiler converts the source code to byte code. Assuming the source code is in a file called Foo.java, the byte code will be put in a file called Foo.class in that same directory.

c) execution

To execute a program, the byte code is interpreted by the Java Virtual Machine. This is when the work of the programmer comes to fruition. The semantics of the various methods are carried out to achieve some desired result. Errors here are run-time errors, and appear as Exceptions.

d) debugging

Debugging is the process of removing errors from a program. It is the most time consuming and frustrating aspect of programming. Any nontrivial program has multiple errors. Only novice programmer imagine that one can program without bugs. Like dropping the balls when juggling; it happens. One important skill in programming is learning to write code that is easy to debug.

e) prototyping

Building simple prototypes and adding functionality as the previous prototype works is perhaps the most important way to make debugging simple; there are simply less places to look for the bugs.

H. Conclusion

Programming in Java is accomplished by writing classes. Classes define the information objects of that type can store (by declaring variables) and the actions those objects can carry out (by declaring methods). Methods have a heading and a body; the former defines the signature of the method, the latter defines its action.

Although there are many programming constructs and statements, the only three that alter the state of the computation, the only three that really do anything, are input, output, and assignment; all the others are organizational, organizing both the structure of the program and which of the big three are executed in what order and how many times.

The standard set of methods that all classes should have, including constructors, accessors and toString(); are written automatically by the ClassMaker. Any others you will have to write by hand.

This chapter presented many of the details of Java programming. If it succeeded, you, the reader, are beginning to understand the interplay of the two dozen odd concepts that make up most of programming. If not, my apologies; with luck, perhaps after a bit more practice this stuff will fall together for you.

I. End of chapter material

i) New terms in this chapter

actual parameters - parameters in the parentheses of a message (may be any expression of the appropriate type) 99

assignment operator - a single equals sign 91

BNF - Backus Naur Form. A metalanguage for describing context free grammars. Commonly used to describe the syntax of a programming language. 88

compile-time - during compilation, compare with execute-time 121

formal parameters - parameters defined in a method heading (each must have a type and a name) 99

instance variable - a variable declared outside of any method, a copy is created for each instance 95

Math.random() - returns a random double in the range [0,1) 118

parameter linkage - when a message is sent with parameter, the values of the actual parameters are copied to the corresponding formal parameters 101

precedence - in an expression with multiple operators, which operator precedes which (* precedes +) 116

scope - the portion of a program where a construct is visible (or defined) 120

Semantics - meaning, or action. 88

shadow - to hide a variable, by being named the same thing in a more local scope; most often happens with parameters 111

static - modifier that creates class variables or methods instead of instance variables or methods 121

Syntax - grammar, or form. 88

ii) Review questions

5.1 Why are initializing constructors useful?

5.2 What is this?

5.3 What is this()?
5.4 Why are good names important?
5.5 What will this output?

```
byte x=127;
x++;
System.out.println("x=" + x);
```

5.6 Name 4 varieties of variables. What are their scopes?
5.7 What does it mean for one variable to shadow another?
5.8 What is the type and value of:

```
17
3.141
0.1*30
(int) 0.1*33
(int) (0.1*33)
2+2
"2+2"
"2"+"2"
Integer.parseInt("2"+"2");
13/4
13%4
""
""+13%4
"(int) 1.414"
```

5.9 What language does this BNF generate?

```
<S> ::= <A> <B>
<A> ::= a | a <A> <B>
<B> ::= b
```

5.10 What do the symbols: ::=, |, [], <>, [x]* mean in BNF

iii) Programming exercises

5.11 You can convert an int to a String by pasting it onto "". Try out this by

```
System.out.println("" + 17);
```

You should see 17 print. Now try

```
System.out.println("" + 17 + 17);
```

What goes wrong? Hint: you can fix it with parentheses.

Chapter 6: Software reuse

A. Introduction

Software is a new invention. It is nearly pure information, like a story, or DNA. Fortunes have been made, and will be made writing and selling software. Creating and distributing software as a commodity is very different from growing and selling soybeans, or building and shipping refrigerators. Once it is written, software can be distributed at low cost on CDs or at almost no cost over the web; this makes enormous profits possible.

A peculiarity of software is that new releases, revisions and updates are common. The extent to which the old software can be reused determines how much work these revisions entail. If even minor changes necessitate reworking large bodies of code, then they are difficult and expensive. On the other hand, if a new version of a product can be produced without extensive rewriting, it is simpler and cheaper. Thus, the possibility of reusing software without reworking it or even looking at it would be a tremendous advantage.

Object oriented programming makes possible software reuse by inheritance and composition. These techniques were introduced in Chapter 5 and will be revisited in more detail here.

B. Inheritance

i) The power of inheritance

Inheritance gives the programmer tremendous power. Once a class is written and debugged, another class can modify or enhance it without the programmer having to worry about destroying its functionality and often without even knowing what its code looks like.

For example, every user Applet class extends java.applet.Applet. The first Applet you wrote (Code Example 2.5 on page 32) had just an empty block following the heading,

Code Example 6.1

```
public class RobotGreeter extends java.applet.Applet {}
```

Code Example 2.5 (revisited)

and it worked just fine (although it didn't do much). This is because it inherited all the functionality of java.applet.Applet. Here, RobotGreeter is said to be a subclass of java.applet.Applet, and java.applet.Applet is the superclass of RobotGreeter.

ii) The Object class

At the top of the Java class hierarchy sits the Object class. Every Java class is a descendant of Object. All of the methods and variables in Object are defined in every object of any type. There are not very many methods defined in Object (you can check the documentation if you are curious -- http://java.sun.com/j2se/1.4.2/docs/api/); the only one we will address here is `toString()`.

Every user-defined class extends some class. If there is no `extends` keyword after the class name, the compiler inserts `extends Object` automatically.

iii) The mechanics of message sending

By now you are familiar with sending messages to objects. Perhaps it has begun to seem straightforward and natural; if a class has a method defined, you can send the associated message to objects of that type. If you try to send a message to an object whose class does not have the associated method defined, then a compiler error is generated and your code will not run. But, consider the FilledCircle class in Code Example 6.2 and the code

Code Example 6.2

```
1 import java.awt.*;
2 public class FilledCircle extends Circle {
3    protected Color myColor = new Color(100,0,100);
4
5    /** Creates a new instance of FilledCircle */
6    public FilledCircle() {}
7
8    public void setColor(Color c) {
9        myColor = c;
10    }
11
12   public void paint(Graphics g) {
13       g.setColor(myColor);
14       g.fillOval(x-radius, y-radius, radius*2, radius*2);
15   }
16}
```

From Code Example 4.11 on page 82

The complete FilledCircle class.

in Code Example 6.3 where iris, a FilledCircle, is being sent `setX()`. There is no `setX()`

Code Example 6.3

```
    public void moveLeft() {
       iris.setX(iris.getX()-2);
       pupil.setX(iris.getX());
   }
```

From Code Example 4.13 on page 84

`moveLeft()` moves goth the iris and pupil (two FilledCircles) left two pixels.

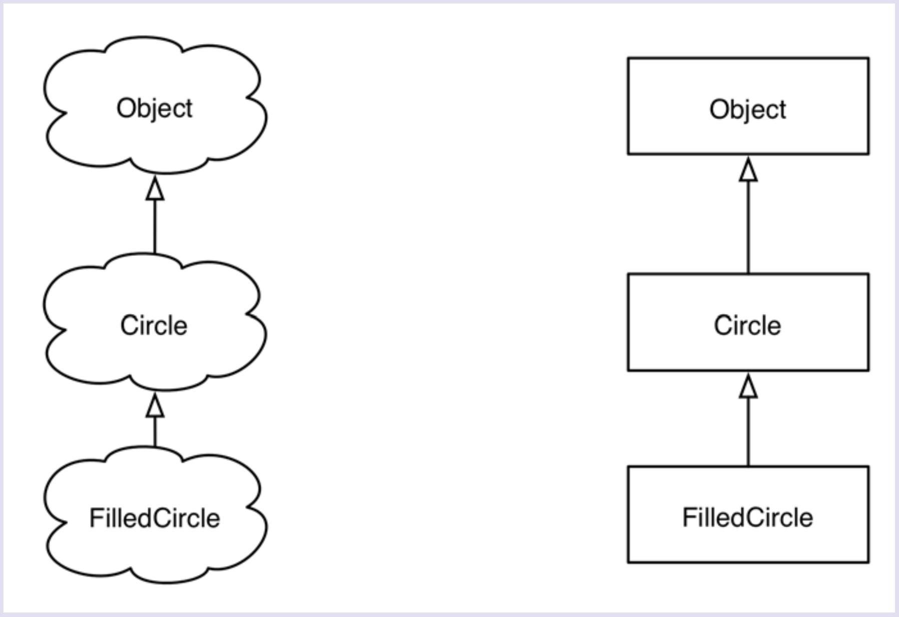
method declared in the FilledCircle class, so why does this code compile? For that matter, do you see an int x variable in FilledCircle? How can you set what doesn't exist? You probably already know the answer, if not, look at line 2 in Code Example 6.2.

At runtime, when an object is sent a message, if its class declares the associated method, the Java Virtual Machine (VM) executes that method. If its class does not declare that method, the VM checks its superclass; if the VM finds it there, it executes it, if not, the search continues up the class hierarchy. When it reaches the Object class; if the method is still not found there, a NoSuchMethodException is generated (you can read about Exceptions in "Exceptions" on page 350).

The FilledCircle class extends the user-defined Circle class, therefore FilledCircle inherits the x, y, and radius variables plus their accessors from the Circle class. The compiler thus allows `setX()` to be sent to iris because it was defined in iris's superclass.

It is often useful to have a picture of the class hierarchy. It aids memory and facilitates

Figure 6.1



Booch and UML diagrams

Two styles of diagram (Booch on the left, UML on the right). Both show that FilledCircle extends Circle which extends object.

communication. Figure 6.1 shows two common styles of class diagram for FilledCircle.

You can send any object toString() and be confident it will work, why?

iv) This is super!

Sometimes, in a constructor, you want to invoke the superclass's constructor. The most common time is in an initializing constructor with a number of parameters, some of which are for variables in the superclass. Just as `this()` invokes the default constructor for this class (see "this()" on page 116), `super()` invokes the superclass constructor.

For example, if you wanted to write an initializing constructor for the four variables in FilledCircle and there was already one for x, y, and radius in Circle, you could write the constructor in Code Example 6.4. The superclass initializes its three variables, then the

Code Example 6.4

```
1 public FilledCircle(int x, int y, int r, Color c) {
2     super(x,y,r);
3     setColor(c);
4 }
```

Initializing constructor for FilledCircle using super()

Line 2: Invoke Circle's initializing constructor to set x, y, and radius.
Line 3: Set the color (!).

color is set locally. Invoking `super()` must be the first thing in the constructor.

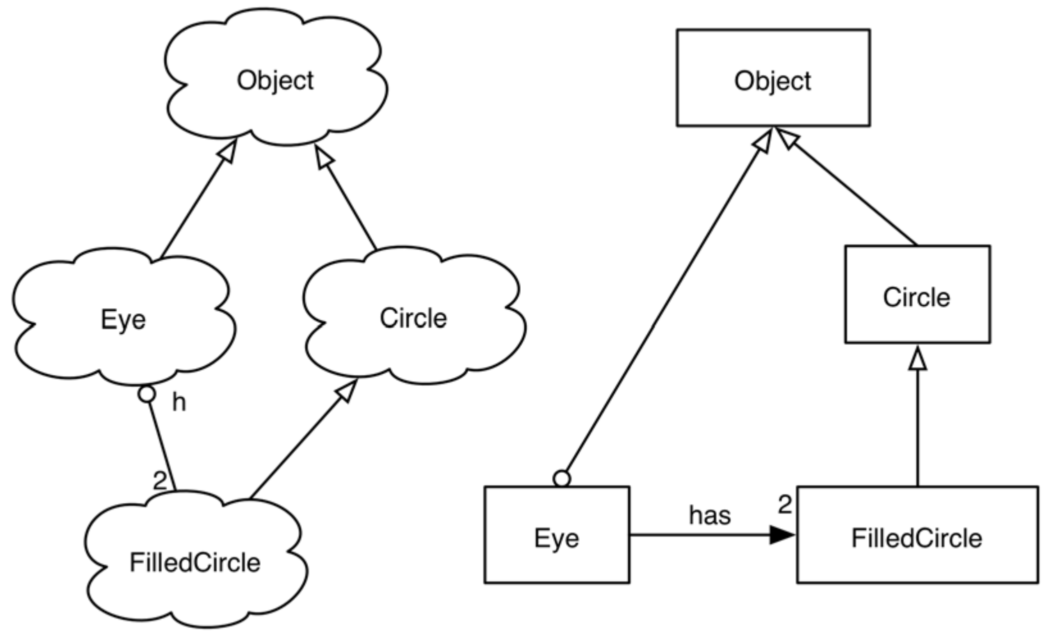
C. Composition

Having one thing composed of several others is common and familiar in everyday life. A student's daily schedule is composed of classes, meetings and meals. A face is composed of various features. Generally, we experience objects in the world as sets of features that travel around together. Containment is also familiar in the world. Dressers contain clothes of various kinds. Toolboxes are used to hold various tools. A backpack may contain books, pens, a water-bottle and a computer.

Composition in object programming shares features from both composite objects and containers in the world. It refers to a class being made up of several objects of other classes. So a composite object may be thought of as a container for other objects or it may be thought of as being the combination of those objects -- either way can be useful in different contexts.

The Eye class was composed of an iris and a pupil, both FilledCircles. Figure 6.2 shows the class diagram for Eye, again in two different styles. In the Booch diagram, each cloud

Figure 6.2



Booch and UML diagrams for Eye

Both show that an Eye extends Object and is composed of two FilledCircles which extend Circle which extends object.

represents a class. There are two kinds of arcs; arrows, representing inheritance, and lines with a dot and an h representing composition. The class at the dotted end **h**as or contains instances of the other. The number at the other end indicates how many instances, by default it is one. In the UML diagram, inheritance is also represented by an arrow, but composition is represented by a horizontal arrow with a "has" label.

To write a Face class, you might include two Eyes, a Nose, and a Mouth (once you wrote Nose and Mouth). To display it you would display each component in a particular spatial relationship to each other. You might write a Person class that included a Face, as well as various other body parts. Then you could create a Scene involving several Persons.

Composition allows you to assemble multiple objects of various types into a container. This is efficient and useful, as it allows you to group things and treat the ensemble as a single thing, thus simplifying your thinking.

D. Composition Programming Example: Snowpeople

i) A description of the task

Here is the programming task for this chapter. Write a Java Applet that will display several snowpeople of various sizes at various locations around the screen. To simulate warm weather, make the snowfolk melt some each time the user pushes a button labelled "warm sunny day". When a snowperson melts, increase the size of a gray puddle of melted snow under it.

ii) Overall Design

Perhaps, given your experience with the Eye class, you feel that you know enough to immediately start writing code. It would be fine to write a prototype SnowPerson class immediately, that’s a reasonable approach. But, before writing very much of the detailed code, it is important to think through the entire problem. What classes will you need? How will they be related? Which will do what? Carefully designing code before writing any can save many hours of frustrating debugging and redesign.

As always, the first questions in GUI object design are, "What will the GUI look like?", and "What classes will be needed?". There is never one right answer to either question, and after working on the implementation some you may realize the class structure or the GUI you’ve chosen needs revision. Making considered choices could save a tremendous amount of time. Aim first, then shoot.

a) The GUI

The interface for this problem is very simple. There is a single button that the user can push to simulate one day of melting. The only output is to display the snowpeople. It might be nice to draw some background for the snowpeople as well.

b) Classes

In addition to the Applet class (which, as usual, will handle the GUI), we will definitely need a SnowPerson class. We might also need a SnowBall and Puddle class, each of which would reuse FilledCircle. Or perhaps the SnowPerson could just contain four FilledCircles, named head, middle, base, and puddle.

c) How many classes should you have?

The right number of classes for a particular problem is somewhat ambiguous. It is perhaps a matter of taste, and as the expression goes, "there's no accounting for taste!". For a large problem, having only one class would be too few; for a small problem, having more than a handful would likely be too many. Let's keep open the choice of whether to have Puddle and Snowball classes for the present, until after considering some of the details.

iii) SnowPerson Design

The description of the task left a number of details unspecified, including: How many snowballs is a snowperson made of? What are their relative sizes? What color are they? How much does a snowperson melt in one day? When it melts, how quickly does the puddle grow? This lack of specificity forces the programmer to either make these decisions arbitrarily, or request additional information. If someone had hired you to produce a snowperson Applet, you would ask them if they preferred to specify those things, or if you should make your own decisions. Imagine how upset everyone would be if you made the decisions yourself, wrote all the code, delivered it, and your customer had totally different expectations. Here, these decisions have been made arbitrarily. Feel free to implement them differently.

a) How many? How big?

Let's assume that a SnowPerson is composed of three white snowballs getting smaller as they go up (as usual). Call them the base, the middle and the head. The exact ratio of sizes is not important; let's say the radius of the middle is 2/3 the radius of the bottom and the head radius, 2/3 the middle. If you don't like how this looks you can adjust it later.

b) Where do the three snowballs go?

Perhaps the most difficult decision is where to locate each of the snowballs. Recall that FilledCircles keep track of the position of their centers. Assume the Applet will specify where each SnowPerson goes using x and y-coordinates. Should that location be used for the top of the head? Or the bottom of the base? Remember that the snowfolk are going to melt. If the location of a SnowPerson were the top of the head, as it melted, the base would rise up into the air! That could be amusing, but is hardly how real snowpeople behave. Thus, the location of a SnowPerson will be where the base touches the ground.

Given that decision, the positions of all three snowballs are fixed. The base is centered its radius above its location. The middle is centered above that by the radius of the base plus its own radius (see Figure 6.4 on page 151). The head, similarly. The details of computing these positions are properly part of implementation.

c) Displaying the snowperson

To display a composite object, simply display each component (remember, the order of display can be important). Assuming the three snowballs and the puddle each store their color and position, this should be trivial.

d) Melting

This is another totally arbitrary decision. Let's say, for simplicity, that the radius of the base decreases by 10% each day and the ratios of the radii remains 2/3. For now, let's say the size of the puddle increases by 10 pixels each day.

iv) Implementation

a) Keeping things simple

Perhaps the most important skill a programmer learns is to keep every method simple. It is possible to make a program work with huge, sprawling methods, just as it is possible to build a vehicle out of spare parts, tape and baling wire -- but, it is almost never a good idea, especially if you have far to go.

b) Strike out on your own?

It won't be long before you start programming independently; coming up with your own tasks to program, or, at least writing programs to complete programming assignments without being provided with the answers. The sooner you can start implementing yourself the better. If you are ready to jump into implementation by yourself already, do it! Try it out and then come back here for hints. On the other hand, if it is still feeling a bit new and strange and you're not sure how to proceed, follow along here; but, try to do the things the text recommends doing before looking at the answers. Dependence is good, but not for too long.

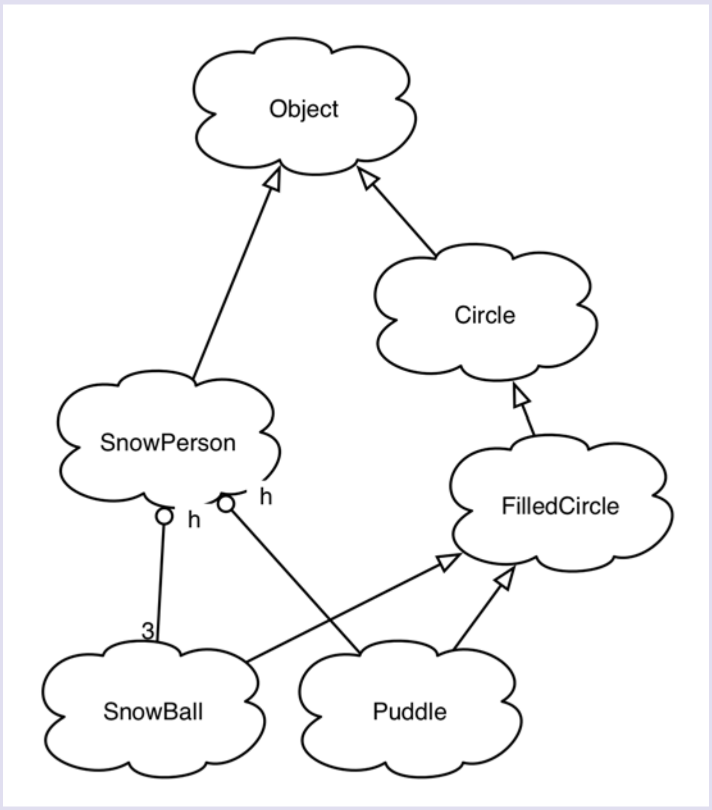
c) SnowBall? Puddle?

The decision about whether to create SnowBall and Puddle classes, or just go with four FilledCircles must now be faced. If you decide to write SnowBall and Puddle, should they extend FilledCircle, or contain a FilledCircle (those are the two techniques to reuse software)? The answers to both those questions hinge on two things: what actions those classes must implement in addition to FilledCircle, how complicated those actions are, and how many modifications you anticipate making in the future. If this code will never be used again, and there is only one additional method, and it is simple, then the answer is use FilledCircle and get it done with. On the other hand, if there are many complex methods needed and extensive modifications may be required, then the answer is write subclasses.

The actions of Snowball and Puddle are shrink, grow, and `paint()`. If they extend FilledCircle they will automatically inherit `paint()` from FilledCircle, as well as the accessors for y and radius (which are needed for computing their new sizes and positions during melting). If they wrap up a FilledCircle (i.e. if composition is used) they would have to implement those accessors all over again. So, that choice is simple, inheritance is more appropriate here.

Although it would be possible to implement a SnowPerson using four FilledCircles, composing it of three SnowBalls and a Puddle will be more elegant and allow inheritance to be illustrated. Given that decision, the class structure for this program is illustrated in

Figure 6.3



Booch diagram of classes

SnowPerson and Circle extend Object. A SnowPerson has 3 Snowballs and 1 Puddle. SnowBall and Puddle extend FilledCircle which extends Circle.

Figure 6.3. Notice that a SnowPerson extends Object and has three SnowBalls and one Puddle.

d) Implementation plan

Here's a series of tasks that will lead to a working program. The rest of this section will detail how to implement them.

1. Create a new directory to store the code for this project
2. Create a new project; mount that directory, and add it to the project.
3. Create a GUI Applet with a Button (for melting); hook up the Button
4. Create (or better, copy!) the Circle, and FilledCircle classes.
5. Create SnowBall, Puddle, and SnowPerson classes.
6. Add the chain of `paint()` methods so that repainting the Applet will repaint all the FilledCircles of all the SnowPersons
7. Add the code to calculate the locations of the snowballs.
8. Add additional SnowPersons
9. Add the melting code.

Naturally, after every step, test your code if you can.

You already know how to do the first three steps; come back here after you've done that. Feel free to look in the Appendices for instructions.

e) Creating the classes

- **Circle and FilledCircle**

Use the Classmaker for Circle, that way you will get an initializing constructor. The FilledCircle class from your Eye project does almost everything it should, except it does not have an initializing constructor, add the one in Code Example 6.4 on page 141.

- **SnowBall**

The SnowBall class extends FilledCircle. Since all SnowBalls are the same color, the constructor does not need a color parameter. The only method the Snowball class needs is `melt()`, which reduces the size by 10%; so the body of that method would be one line:

```
    setRadius(getRadius()*9/10);
```

It also needs an initializing constructor, with a single line:

```
    super(x,y,r,java.awt.Color.WHITE);
```

- **Puddle**

Like SnowBall, Puddle extends FilledCircle. It needs the same initializing constructor (but that sets the color to `java.awt.Color.GRAY`) and a `grow()` method to increase its size by 10 pixels (`setRadius(getRadius()+10);`).

- **SnowPerson**

The SnowPerson class has three SnowBalls called base, middle and head. It will need an initializing constructor that is passed the location and size. That constructor will then calculate where and how big the three snowballs should be. For a first prototype let all three be the same size, as in Code Example 6.5. This code centers the base at (x,y)

Code Example 6.5

```
1    public SnowPerson(int x, int y, int size) {
2        base = new SnowBall(x, y, size);
3        middle = new SnowBall(x, y-size, size);
4        head = new SnowBall(x, y-size*2, size);
5    }
```

Prototype initializing constuctor for SnowPerson

Line 2: Create the base snowball.
Line 3: Middle snowball is the same size, but higher.
Line 4: Head snowball is the same size, and higher still.

instead of putting the bottom of the base at (x,y). But, that’s good enough for now, you can adjust the locations and sizes later (after the SnowPerson shows up on the screen). First build the structure of the program, then refine it.

f) Adding the paint() chain

You will recall from Chapter 4, that to update the way an Applet looks we send it `repaint()`, which causes `paint(Graphics)` to be sent to it. When the Applet gets the `paint()` message, it should paint() all the things it displays. Add a `public void paint(Graphics)` method to your Applet that simply sends `paint(g)` to the Snowperson (see Code Example 4.6 on page 74). Before that will compile, you must have a Snowperson to send the message to; so, declare and initialize a SnowPerson instance variable in your Applet. That last instruction is completely explicit; if you don’t know how to do it you might look back at "Variables I (state)" on page 98. A big part of introductory programming expertise is familiarity with terms and concepts; the sooner you become familiar with them, the sooner programming will be easy. Perhaps you would consider reading "Recapitulation" on page 129?

Test that code (note that this is step 6 of "Implementation plan" on page 148), then on to making it look like a snowperson. If there are any problems, and the solutions don't jump right out at you, you might check "What could go wrong?" on page 154.

g) Letting the computer do the arithmetic

The SnowPerson constructor, when passed its size and position, must calculate how big the middle and base are and where the three SnowBalls will go. Calculating the sizes is very easy, see Code Example 6.6. Calculating the locations is more complicated.

Code Example 6.6

```
1   private void adjustSnowBallSizes() {
2       middle.setRadius(base.getRadius()*2/3);
3       head.setRadius(middle.getRadius()*2/3);
4   }
```

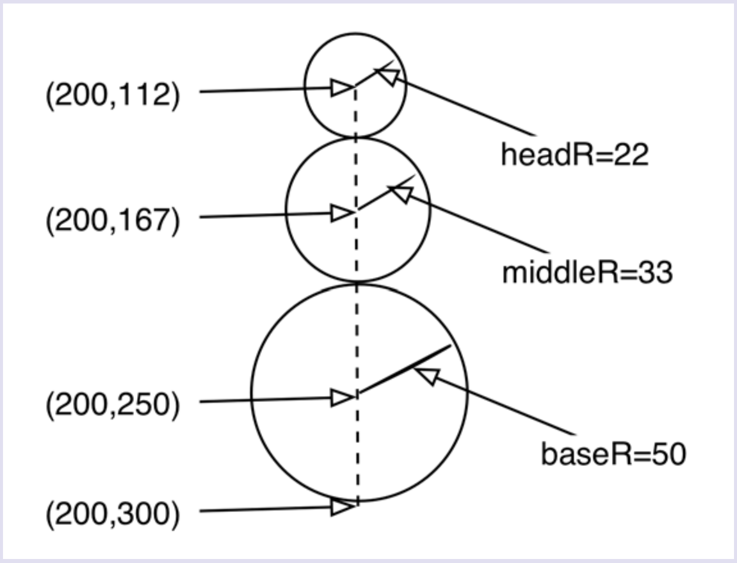
Setting the sizes of the middle and head from the base.

Line 2: Set the radius of middle to 2/3 the radius of the base.

Line 3: Set the radius of head to 2/3 the radius of the middle.

Consider, by way of example, a SnowPerson of size 50, located at (200,300) (see Figure 6.4). All three SnowBalls have the same x-coordinate, but the y-coordinates must be calculated. The center of the base will be 50 pixels above the bottom (since the radius is 50) at (200, 250), the top will be 50 pixels above that. The center of the middle SnowBall will be 33 pixels above that (since 33 is 2/3 of 50) at (200,167). Similar reasoning puts the center of the head at (200,112). If, more abstractly, the SnowPerson is at (x,y), and we

Figure 6.4



Locations of the three SnowBalls

Assuming the base has a radius of 50 (baseR=50), and touches the ground at (200,300), the coordinates of the centers of the three SnowBalls are indicated.

call the radii of the three SnowBalls baseR, middleR and headR, then we could write the y-coordinate of the base as, *y - baseR*.
The y-coordinate of the middle is *y - baseR*2 - middleR,* or *baseY - baseR - middleR.* The y-coordinate of the head is *y-baseR*2-middleR*2-headR,* or *middleY-middleR-headR.*

In Java, this looks like:
```
    base.setY(y-base.getRadius());
    middle.setY(y-base.getRadius()*2-middle.getRadius());
    head.setY(y-base.getRadius()*2-middle.getRadius()*2-head.getRadius());
```
Notice that the y variables in the various SnowBalls store the y-coordinates as they are computed. An alternative computation is:
```
    base.setY(y-base.getRadius());
    middle.setY(base.gety()-base.getRadius()-middle.getRadius());
```

```
head.setY(middle.getY()-middle.getRadius()-head.getRadius());
```

Add whichever version you like better to your constructor, and check that it works. Note that the y-coordinate should be stored in the SnowPerson (since when the base melts its center must be moved down to its radius above where it touches the ground). You could either type the three lines directly in the constructor, or put them in a method, called something like `adjustSnowBallLocations`() and invoke that method from the constructor. Which is better? It depends on two things. Grouping code together in a method with a name helps to make obvious what it does. Also, if it turns out you will need to use that code somewhere else in your program, then you can reuse a method instead of copying and pasting the code. My completed constructor appears in Code Example 6.7.

Code Example 6.7

```
public SnowPerson(int x, int y, int size) {
    this.y = y;
    base = new SnowBall(x, y-size, size);
    middle = new SnowBall(x, y-size, size);
    head = new SnowBall(x, y-size*2, size);
    thePuddle = new Puddle(x,y,0);
    adjustSnowBallSizes();
    adjustSnowBallLocations();
 }
```

Initializing constructor for SnowPerson.

Note that the computation of SnowBall sizes and locations is done in methods; these will be reused in `melt()`.

h) Adding additional snowmen

Add at least two more SnowPersons to your scene. Make them different sizes. Test to make sure they display correctly.

If you know how to that, do it. If not, read your Applet code (so it is in your mind). Look carefully at each line that you have written. Think about what each does. Now decide how to add another SnowPerson.

Stop. Don't read on until you have puzzled over what to do. Okay... There are only two changes needed. First, declare and instantiate another SnowPerson. Second, modify `paint()` so that it also draws the new one. That's it.

i) Making the snowman melt

When the user pushes the melt button it must send the `melt()` message to however many SnowPersons there are and then send `repaint()` (so you can see the changes). When a SnowPerson gets a `melt()` message, it must both decrease the size of its SnowBalls (and lower them) and increase the size of its Puddle. You can make these changes either starting with the Applet and working down, or starting with SnowPerson and working up. The text will take the latter approach.

There are a number of details that must be attended to to accomplish melting, but the `melt()` method can be written without paying any attention to them as can be seen in Code Example 6.8. Because the code to calculate the sizes of the middle and head, and

Code Example 6.8

```
public void melt() {
    base.melt();
    adjustSnowBallSizes();
    adjustSnowBallLocations();
    thePuddle.grow();
}
```

melt() method for SnowPerson

There are four action that must take place: reduce the size of the base, calculate the new sizes of the middle and head, calculate the new locations of there centers, and finally growing the puddle. The four lines of this method do that, but the details are in the methods.

the code to calculate the locations of all three SnowBalls were written as methods, this code is simple and easy to write.

Now add the code to send the `melt()` message to all the SnowPersons in the `ActionPerformed()` method of the melt Button and test the melt method.

j) Displaying the Puddle

If you are simply following these instructions, then the Puddle is not being displayed yet. Add the code to do that. All you need is to add `puddle.paint(g)` in `z`) in `paint()` in SnowPerson. Does it matter where in that method it goes? Test the completed (!) code.

Congratulations! You have just completed a programming assignment more complex and sophisticated than any introductory programming text could have imagined presenting in a procedural language. You have used composition and inheritance to leverage already written classes. You have implemented a `paint()` chain with a GUI interface and started down the road to understanding object programming. Not everyone makes it this far.

E. Conclusion

In Java, all classes are organized into a hierarchy, called a tree, with the class Object at the root. Every class has Object as an ancestor, i.e. Object is a superclass (although possible at several removes) to every other class.

Inheritance and composition allow software to be reused. This is a tremendous advantage over languages where software cannot be reused without extensive reworking. Given a working FilledCircle class, writing SnowBall and Puddle only took a few lines of code. Once you have working SnowBall and Puddle classes, the SnowPerson class becomes simple.

In object languages, algorithmic complexity can be reduced by building an appropriate class hierarchy. The more complex the code, the harder it is to understand. Complex code is difficult to write correctly and more difficult to debug when there are errors. Simple code is easy to write correctly and easier to debug when there are errors. And there are always errors. Only neophyte programmers imagine their code will not have bugs. Experienced programmers know better. Good programmers learn techniques to make it easier to find the bugs that inevitably creep in. They develop good habits; these habits allow them to succeed, even in difficult situations. Well thought out, coherent class hierarchy and incremental implementation are techniques that allow programmers to succeed.

F. End of chapter material

i) What could go wrong?

Problem 6.1 -- the SnowPerson doesn't appear on the screen.
Possible causes: 1) `paint()` is never sent, 2) white FilledCircles on a white background are invisible,
Possible solutions: 1) send the paint message, 2) change the color of the SnowBalls or the background

---

Problem 6.2 -- the SnowPerson never changes size
Possible causes: 1) the melt Button `actionPerformed()` method is not written, 2) it does not tell the SnowPerson to melt, 3) `repaint()` was not sent to the Applet
Possible solutions: 1) Hook up the Button, 2) send the `melt()` message from the body of `ActionPerformed()`, 3) send repaint() after `melt()`.

Problem 6.3 -- The SnowBalls overlap in the initial SnowPerson.
Possible causes: Bad arithmetic in the SnowPerson constructor.
Possible solutions: Fix the arithmetic.

Problem 6.4 -- The SnowPerson is upsidedown!
Possible causes: The programmer forgot that 0 in the y direction is the top of the screen. The author made this mistake.
Possible solutions: Redo the code for calculating where the middle and head go, remembering which way is up!

Problem 6.5 -- After melting, the middle and/or head are floating.
Possible causes: The programmer forgot to add the code to adjust the position of the middle and/or head after melting, the programmer remembered to adjust the position, but forgot to send the message.
Possible solutions: Add that code, invoke it.

Problem 6.6 -- The original SnowPerson appears on the screen, but the second doesn't.
Possible causes: 1) It's on top of the first SnowPerson. 2)`paint()` does not send it paint()
Possible solutions: 1) give it different coordinates. 2) If you called it person2, add `person2.paint(g)` to `paint()` in the Applet.

Problem 6.7 -- The original SnowPerson melts, but the second doesn't.
Possible causes: the melt `actionPerformed()` method doesn't send`melt()` to the second one.
Possible solutions: add the `melt()` message.

ii) Review questions

6.1 What are the two techniques to reuse classes?
6.2 Write a SnowBall class that uses each technique to reuse FilledCircle. Which seems better to you? Why?
6.3 Does SnowBall use composition or inheritance?
6.4 How do you know if you need more classes?
6.5 How do you know if you have too many classes?
6.6 Write down, one per line, every method invoked (including the values of the parameters) when `new SnowPerson(50, 200, 300)` is executed.
6.7 Write the `melt()` method for a SnowBall. Make it reduce the size of the SnowBall by 17%. Do you have to worry about which order the operators are applied?
6.8 Describe in detail what happens when the SnowPerson sends `melt()` to the middle. What object is this? What methods are invoked in what classes in what order?
6.9 Describe in detail what happens when a user pushes the Melt for a day Button. Include every method invoked and what class in resides in, in the correct order. Pretty scary, eh?

iii) Programming exercises

6.10 Send `toString()` to your Applet and System.out.println() what it returns. What does it print?
6.11 Put this line in initComponents in your Applet: `System.out.println(this);` How do you explain why it works?
6.12 Send `toString()` to an object that you know does not have it defined. What does that print?
6.13 Modify your applet to display 50 SnowPersons of random sizes at random locations. Look back at the previous chapter for how to do this.
6.14 Modify your code so that all of the SnowPersons move a few pixels toward the center of the screen each time you push the button (in addition to melting).
6.15 You may notice that the puddle of one SnowPerson covers up others. Modify the display methods so that the puddles are all in the background. Hint: draw all the puddles first -- i.e. add a `paintPuddle()` method and in the Applet first paint all the Puddles, then the Snowballs. Hint2: you can calculate the change you should make to the x-coordinate to move right or left (depending on if it is left or right of center) arithmetically. There is a method, `Math.abs()`, that will return the absolute value of an int.

Chapter 7: Conditional statements

A. Introduction

i) Procedural programming and control structure

In the old procedural paradigm, writing a program was essentially the construction of an algorithm (see "Definition of algorithm" on page 2). To design a program, the entire task was broken down into a series of subtasks. To implement that design, subprograms (also called subroutines) were written for each subtask. Typically there was a main loop which repeatedly called various of the subroutines depending on conditions. Looping and conditional statements were central to understanding and building programs, so they always appeared early in programming texts; very little could be done without them.

ii) Object programming allows you to substitute class structure

By contrast, in the object paradigm presented here, writing a program involves designing and implementing a class structure and a GUI. The finished program still implements an algorithm, but its complexity is distributed across the various classes. In a properly implemented object program, every class and method is simple. Once classes are written they may be reused with a minimum of labor.

Thus far, the programs in this text have accomplished conditional and repeated action by relying on the user and the event loop. The event loop is the mechanism built into the Java VM to handle user events. If the user wanted to make several withdrawals, they pushed the withdraw Button several times. To move the Eye left, they pressed the moveLeft Button. Nonetheless, object programs do need looping and conditional statements. This chapter and the next will introduce those two elements of control structure.

B. Different actions depending on conditions - Conditional execution

i) The if statement -- do something or don't

Every program thus far has run the same way every time you ran it. But there are times when a program needs to choose between actions to perform on the basis of the current conditions. For instance, ATM machines usually won't let you withdraw more money than you have in your account. The one in Chapter 3 would pretend to give out money even if the resulting balance was negative. The code for `withdraw()` is shown in Code Example 7.1. An if statement can be used to prevent the Account from being overdrawn,

Code Example 7.1

```
public void withdraw(int amountToWithdraw) {
    balance = balance - amountToWithdraw;
}
```

The withdraw method for the Account class (from Code Example 3.4).

as in Code Example 7.2. An if statement starts with the word if, then a boolean

Code Example 7.2

```
1    public void withdraw(int amountToWithdraw) {
2        if (balance >= amountToWithdraw)
3            balance = balance - amountToWithdraw;
4    }
```

A withdraw method that prevents overdraughts.

Lines 2-3: This if statement causes the assignment on line 3 to only occur if the balance is at least as big as the amount to withdraw.

expression in parentheses, then a statement to execute if that expression evaluates to true. This syntax is shown in BNF 7.1 . Remember that every legal if statement must match

BNF 7.1 The if statement

<if stmt> ::= if (<boolean expression>) <stmt> [else <stmt>]

Semantics

1: Evaluate the <boolean expression>

2: If the value of the expression is true, execute the <stmt> after <expression>

3: If the value is false and there is an else part, execute the <stmt> in the else part.

this syntax *exactly*. The expression must be of type boolean (since only the values true and false make sense here) and must be enclosed in ()s. Then there must be exactly one statement (if there are several things you want to do in the if part you must enclose them in {}s to transform them into a single block statement).

ii) if-else -- do one thing or another

An if statement (without an else) is used when you want an action performed only under certain conditions; it either executes the statement following the expression or does nothing, depending on the value of the expression. By contrast, an if-else statement is used to choose between two actions.

a) Example: preventing overdrafts while alerting the customer to the problem

If a person tries to withdraw more money than they have, the code in Code Example 7.2 would simply ignore them; which could be a bit unsettling. It might be better to let them know something had gone wrong. I.e. either make the withdrawal, or print an error message. That's a choice between two things, so the construct to use is the if-else statement, as in Code Example 7.3.

Code Example 7.3

```
1    public void withdraw(int amountToWithdraw) {
2        if (balance >= amountToWithdraw)
3            balance = balance - amountToWithdraw;
4        else System.out.println("Oops! You don't have that much!");
5    }
```

A withdraw method with an if-else statement.

Lines 4: The else part executes if amountToWithdraw > balance. Thus, if the user is trying to withdraw more than they have, instead of ignoring them, it will print an error message.

b) More complex boolean expressions

The boolean expression in an if statement may, like any expression, be arbitrarily complex. Numbers may be compared by any of the relational operators (see BNF 5.13 "binary operators" on page 122). Boolean expressions may be conjoined with || (or), and && (and), and negated with ! (not).

Examples:

```
x>0      // true if x is >0
x>=0 && x<=100 // true if 0<=x<=100, i.e. a legal exam score
x==7 || x==11// true if x is 7 or 11, i.e. a winner in craps
x!=7 && x!=11 && x!=2 && x!=12//true if x is not 7, 11, 2, 3, or 12
```

c) Truth tables

It is not always obvious exactly how to write boolean expressions, especially in complicated situations. When in doubt, a heavy-handed, but inevitably correct technique is to make a truth table. A truth table lists (or, as computer scientists like to say, enumerates) all the possible combinations of values for the boolean clauses in a boolean expression, along with the values of the expression under those conditions. For instance, assume there are two boolean variables, p and q. Each may take on either the value true (T), or false (F). Thus, for the pair, there are four possible values; TT, TF, FT, and FF. These are shown in the leftmost two columns of Table 7.1. The rightmost column has the

Table 7.1: Truth table I

p	q	p && q	p || q	!p
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

values for not-p (!p); notice that they are just the opposite of the values for p. The values for p&&q and p||q are shown in between; notice that || is inclusive-or, it includes the case when both p and q are true. There is another or operator, exclusive-or, which is true just if p or q is true, but not both. Java’s or is inclusive-or.

d) DeMorgan's Law

The ! operator seems straightforward, it turns true to false and false to true. But, there is a peculiarity of applying ! to expressions including operators. The ! operator distributes across parentheses, but it changes || to && and && to ||. See Table 7.2 for an example.

Table 7.2: Truth table 2

p	q	p && q	!(p && q)	!p && !q	!p || !q
T	T	T	F	F	F
T	F	F	T	F	T
F	T	F	T	F	T
F	F	F	T	T	T

The fact that !(p&&q)=!p||!q and that !(p||q)=!p&&!q is called DeMorgan’s Law; forget it and you will run into some nasty bugs.

e) Problem Solving technique - Analysis By Cases

It is very common in writing a program, and in problem solving in general, that one must do different things in different cases. For instance, if you are running under a frisbee, if it was thrown forehand, you expect it to tail off one way, if it was thrown backhand, the other. If it was thrown as a hammer (up-side-down) you expect it to slow down rapidly and tail off abruptly. In each case you do different things to catch it.

Analysis By Cases (ABC for short) is a problem solving technique designed especially for problems with multiple cases.

Problem Solving Technique

Analysis By Cases (ABC)

Identify the various cases. For each, answer the following questions (making a table if it is complicated): 1) How can you distinguish this case? 2) What action do you wish to take in this case?

Once you have identified each case, decided how to distinguish each case from the others, and what action to perform in each case, you are ready to write code. The examples will illustrate the use of this technique.

f) Example - a robot bouncer

Imagine going to a club and encountering a robot bouncer. The job of the robot bouncer is to only let in people who are at least 18 and to charge them each the cover charge. Write a method that is passed a Person as a parameter, and that outputs as a message to System.out what the robot would say to that person. Assume that the Person is passed as a parameter and that a Person object has an age and a balance variable with standard accessors.

To start with perhaps the best thing to do is write a method that only checks their age. There are just two cases here.If the person’s age is greater than or equal to 18, it should say "Welcome"; otherwise say "Sorry". That was the ABC method in a very simple context; so simple it is, well, trivial. The two cases were under 18 and not. They are distinguished by the age of the person. Etc. To write code for this requires an if-else statement, as in Code Example 7.4. This would work fine if there were no cover charge to

Code Example 7.4

```
1 void checkAge(Person aPerson)
2     if (aPerson.getAge() >= 18)
3         System.out.println("Right this way!");
4     else System.out.println("I'm sorry, you must be 18 to enter.");
5 }
```

Robot Bouncer that only checks age

collect. Next, one might reason that if the person is 18 or older, then the bouncer, instead of waving them in, should check if they also have the cover charge. This is shown in Code Example 7.5. There, line 3 in the previous example has been replaced by an if-else.

Code Example 7.5

```
1 void checkAgeAndBalance(Person aPerson)
2     if (aPerson.getAge() >= 18)
3         if (aPerson.getBalance() > 5)
4             System.out.println("Right this way!");
5         else System.out.println("Sorry, you don't have the cash");
6     else System.out.println("I'm sorry, you must be 18 to enter.");
7 }
```

Robot Bouncer that checks age and balance

This example replaces line 3 in the previous with an if-else to check if they have the money to pay the cover charge.

This is legal syntactically, since an if-else statement *is* a statement (check the BNF if you have any doubts about this). This would work; but it is cleaner to use a compound boolean expression to check both conditions at once. The person is allowed in if their age *and* their balance meet certain conditions. This is illustrated in Code Example 7.6.

Code Example 7.6

```
void checkAgeAndBalance2(Person aPerson)
    if (aPerson.getAge() >= 18 && aPerson.getBalance > 5)
        System.out.println("Right this way!");
    else System.out.println("You must be 18 and have $5 to enter.");
}
```

Robot Bouncer that checks age and balance using &&

iii) cascaded if-elses -- do one of a number of things

Sometimes a program must do exactly one of a number of things. In that case you can build a structure called a cascaded if-else by repeatedly using an if-else statement as the statement following the else. The code in Code Example 7.5 could be rewritten into a cascading if-else as shown in Code Example 7.7. The order that conditions are checked

Code Example 7.7

```
void checkAgeAndBalance(Person aPerson)
    if (aPerson.getAge() < 18)
        System.out.println("I'm sorry, you must be 18 to enter.");
    else if (aPerson.getBalance() > 5)
        System.out.println("Right this way!");
    else System.out.println("Sorry, you don't have the cash");
}
```

Robot Bouncer that checks age and balance using a cascaded if-else

This is exactly equivalent to Code Example 7.5 but is perhaps easier to read.

may be important; careful thinking is required to make sure a it will work properly in every case.

a) Example I - a robot aspirin bottle

Imagine that you are assigned to program an aspirin bottle to announce the correct dosage given a person’s age. The dosage for aspirin is as follows: under 5, consult with a doctor;

6-12, one; 13-65, two; over 65, one. Code Example 7.8 shows this coded as a cascading

Code Example 7.8

```
1 if (age<6)
2     System.out.println("consult with your physician");
3 else if (age<13)
4     System.out.println("dosage=1");
5 else if (age<66)
6     System.out.println("dosage=2");
7 else System.out.println("dosage=1");
8
```

Cascaded if-else for aspirin dosage

Notice that in the else clause it must be the case that the previous condition was false (otherwise the else clause would not execute). Thus if execution reaches line 3, age must be >=6, and if execution reaches line 7, age>=66.

if-else. Each else-if clause executes only if the previous boolean expression was false (because of the semantics of an if-else; see BNF 7.1 "The if statement" on page 158, if you have any doubts about this. Once you get used to this notion, it will be very obvious.). It is also possible to make this a bit shorter as shown in Code Example 7.9.

Code Example 7.9

```
1 if (age<6)
2     System.out.println("consult with your physician");
3 else if (age<13 || age>65)
4     System.out.println("dosage=1");
5 else System.out.println("dosage=2");
6
```

A slightly shorter cascaded if-else for aspirin dosage

Notice that if execution reaches line 5, !(age<13 || age>=65) must be true, in other words it must be true that (age>=13 and age <=65).

Which is the right way to structure this? There is not one clear answer. It depends on which way makes sense to the programmer, and which way is clearer to a reader. It's a matter of style.

b) Example II - reporting the score of a tennis game

Imagine you were assigned to build a robot tennis score announcer. In tennis, scores of zero, one, two, three, and four, are announced as love, fifteen, thirty and forty. Assuming the scores are kept as ints, your robot score announcer would be passed two ints. It must convert the two int scores to the appropriate String. E.g. if the score were 3 for the server and 2 for the receiver the correct output is forty-thirty. So a method that announced the score might make use of a method that converted an int to a tennis score for both of the scores, as in Code Example 7.10. In that case the task is to write and test the `convert()`

Code Example 7.10

```java
void announceScore(int serverScore, int receiverScore) {
    System.out.println(convert(serverScore) + "-" +
                                    convert(receiverScore));
}
```

An announceScore method that uses convert twice.

method. As usual, the simplest way to generate a test driver is to create a class with a `main()` method (see NetbeansAppendix L on page 339), as in Code Example 7.11. The

Code Example 7.11

```
1 public class SingleScoreConverter {
2
3    /** Creates a new instance of ScoreAnnouncer */
4    public SingleScoreConverter() {
5    }
6
7    private String convert(int score) {
8        if (score==0)
9            return "love";
10        else if (score==1)
11            return "fifteen";
12        else if (score==2)
13            return "thirty";
14        else if (score==3)
15            return "forty";
16        else return "value out of range:" + score;
17    }
18
19    public static void main(String[] args) {
20        SingleScoreConverter theConverter = new SingleScoreConverter();
21
22        System.out.println("0=" + theConverter.convert(0));
23        System.out.println("1=" + theConverter.convert(1));
24        System.out.println("2=" + theConverter.convert(2));
25        System.out.println("3=" + theConverter.convert(3));
26        System.out.println("4=" + theConverter.convert(4));
27    }
28
29}
```

Testing convert.

The main() method tests all four legal values and one that is out of range. Thus, it tests all five cases. It is important to test *all* the cases, otherwise, the one you didn't test will be a hard to find bug later.

`convert()` method is implemented using a cascaded if-else. It returns the String representing the int it is passed. It does not use System.out.println to display because that's just for debugging or experimenting with code. Notice, on line 20, that an instance of SingleScoreConverter is created and stored in a variable. Then the `convert()` messages are sent to that object. Why is this?

c) Class methods

Almost every method you have seen so far has been an instance method. Instance methods are invoked when a message is sent to an instance of a class; in the context of an instance method, "this" is a reference to the object that the message was sent to. Class methods, like class variables (see "class variables" on page 127), have the keyword static in the heading; they are invoked when a message is sent to a class. Examples include: `int withdrawal = Integer.parseInt(textField1.getText()); from` "When the user hits enter, get the withdrawal amount" on page 44, and `Thread.sleep(50);` from "step()" on page 218.

In the context of a class method, "this" is not declared (since the message was *not* sent to an instance but instead to a class). Therefore, in the body of `static void main()`, we can't write`convert(3)`. If we did, the compiler, knowing that the syntax of every message statement is object.message(parameters), would append "this" on the front, to make, `this.convert(3)`. Then it would notice that "this" was not defined, and would protest.

So, even though we are inside a method in the SingleScoreConverter class, because it is a static method, we cannot directly invoke instance methods in that class. Instead we must instantiate SingleScoreConverter and send the messages to the instance.

iv) The switch statement

Another construct that selects between a number of possibilities is the switch statement. The switch statement does not add any power to the cascaded if-else, but it is a little neater and easier to read. The syntax is shown in BNF 7.2. This is the most complicated

BNF 7.2 The switch statement

```
<switch stmt> ::= switch (<enumerable expression>) {
                        [<case clause>]*
                        [default: <stmt>]
                    }

<case clause> ::= case <constant>: [<statement>]*
```

Semantics

1: Evaluate the <enumerable expression>

2: If the value of the expression equals any of the <constant>s in the <case clauses>, then execute the <stmt>s after that <constant>, and all the rest of the cases!

3: Otherwise execute the <stmt>s after default (if it appears).

statement in Java, it is inherited, so to speak, from C++ and old C. The convert method from Code Example 7.11 is shown in Code Example 7.12, and is rewritten using a switch

Code Example 7.12

```
private String convert(int score) {
    if (score==0)
        return "love";
    else if (score==1)
        return "fifteen";
    else if (score==2)
        return "thirty";
    else if (score==3)
        return "forty";
    else return "value out of range:" + score;
 }
```

convert() using cascaded if-else

Code Example 7.13

```
private String convert(int score) {
    switch (score) {
        case 0: return "love";
        case 1: return "fifteen";
        case 2: return "thirty";
        case 3: return "forty";
        default: return "Value out of range: " + score;
    } // switch
}

```

Convert() using switch.

This `convert()` uses a switch statement instead of a cascaded if-else. Switch adds no power, but some people find it easier to read.

statement in Code Example 7.13.

There are two difficulties with the switch statement. One is that it can only switch on types with are enumerable. Enumerable types include int, and char. String and double will not work. The other is that unless the statements in each case clause end with a break or return statement, *the following cases are all executed too*! This can have surprising (and sometimes upsetting) results.

C. Programming example: using the SingleScoreConverter class in a tennis score keeping program

i) A description of the task

The programming task for this chapter is to write a Java Applet that will keep track of the score in a tennis match. It should announce the score before each point, as well as the winner of each game, set, and finally the match. The winner of a tennis match is, like in Wimbledon Women's Tennis, the first player to win two sets. A player wins a set if they have won at least 6 games and are at least 2 games ahead (we will not handle tie-breakers).

The same player serves for an entire tennis game. Before each point, the score is announced, server's score first. A game is won when one of the players has at least five points and is at least two points ahead. If the score reaches 4-4, then, until the game is decided, the score is announced as "deuce" for ties and "advantage server" or "advantage receiver", depending on who is ahead by one.

Assume for now that the user will push one of two Buttons for each point, one if the server gets the point, the other if the receiver gets the point.

ii) Design

Both the GUI and the classes must be designed.

a) GUI Design

The GUI is simple; two Buttons and somewhere to display the output. Either a TextField or a TextArea could work for that. Which would be better? A TextField only has one line. The user might want to be able to see the history of points, so a TextArea seems more sensible.

Here's how to create a prototype (although, odds are you are familiar with this by now).

1. Create a new GUI AWT Applet named TennisApplet (don't forget to create a directory to store all the files in first).
2. Add, rename, relabel, resize, connect and test two Buttons (don't forget to set the Layout to null so they don't fill the screen).
3. Add, rename, and resize a TextArea - write in the TextArea on a Button press to make sure everything is working so far.

b) Adding images to the GUI (optional)

You can make your GUI look much nicer if you add images to it. For instance, the programmer that wrote this Applet to keep himself amused while writing this, copied images of Maria Sharpova and Serena Williams, the two finalists in the 2004 Wimbledon tournament. Then he added them to his Applet.

This requires:
1. Find and copy the images you want to have in your GUI (save them in the same directory as the code; otherwise it won't work).
2. Declare an Image variable for each one.

Code Example 7.14

```
1 import java.awt.*;
2
3 public class TennisApplet extends java.applet.Applet {
4     Image mariaImage, serenaImage;
5
6     /** Initializes the applet TennisApplet */
7     public void init() {
8         initComponents();
9         // read from code directory
10        mariaImage = getImage(getCodeBase(), "maria.jpg");
11        serenaImage = getImage(getCodeBase(), "serena.jpg");
12    }
13
14    public void paint(Graphics g) {
15        g.drawImage(mariaImage, 250, 100, 150, 200, this);
16        g.drawImage(serenaImage, 50, 100, 150, 200, this);
17    }
```

The TennisApplet

Line 1: import java.awt.*, so it will know what Graphics and Image are
Line 4: Declare two Image variables
Lines 10-11: Read in the two images. The files must be named exactly maria.jpg and serena.jpg, or else (of course) they will not be found.
Lines 14-17: Draw them. The four int parameters specify the rectangle to draw the image in; it will be scaled to that size.

3. Read each image into its Image variable
4. Write a public void paint(Graphics) method to draw them.

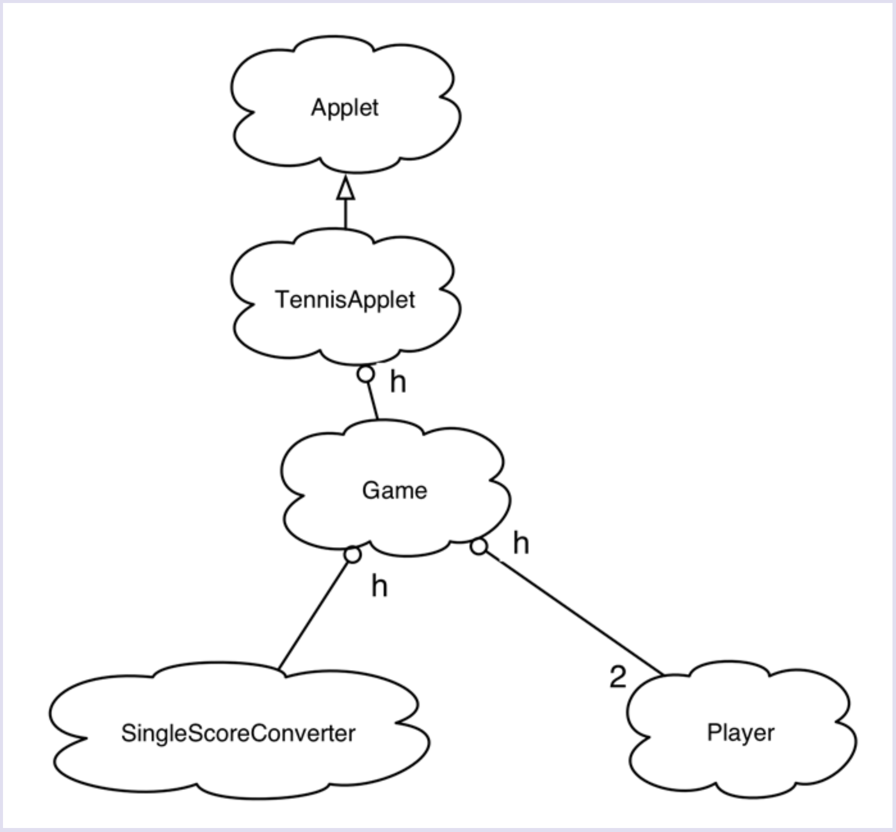
Step 1 can be accomplished either by finding an image on the web, right-clicking it and selecting "Save" (or some such), and then navigating to the directory your tennis code is in. Or, you could copy them from
http://www.willamette.edu/~levenick/SimplyJava/images/
The code to accomplish steps 2-4 is in Code Example 7.14. You will need to arrange your Buttons and TextArea so they are not on top of the Images. Or you can change the x,y coordinates of the rectangles the Images are displayed in. Whatever works for you. If you've forgotten Java rectangles, look back at "Basics of graphics in Java" on page 67.

c) Class Design

The class structure for this problem is not obvious at first glance. Clearly, there will be an Applet. Maybe there should be an Announcer class that uses a SingleScoreConverter (see Code Example 7.10 on page 165) which will announce the score before each point.

In a real tennis game, there are two players; so Game and Player are candidate classes. Perhaps the Game could do the announcing, instead of having a separate Announcer class. If we were going to extend the program to simulate tennis games (instead of pushing a Button for each point), the Player class might include information about a player's attributes, like service consistency and speed, stamina, quickness, strength, or accuracy. If the program kept track of the score for a single Game, the class structure

Figure 7.1



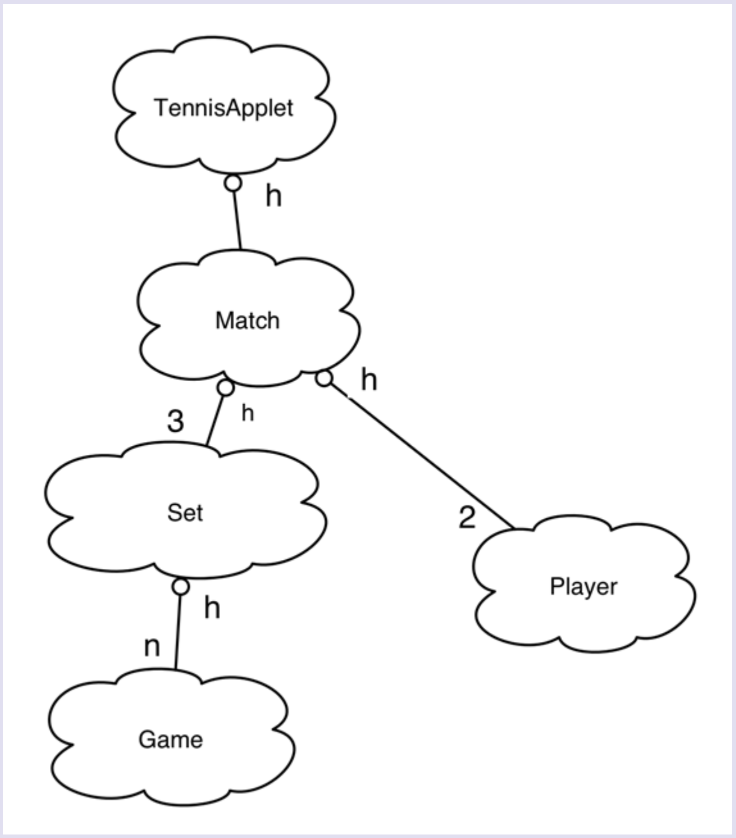
Booch diagram for one game of Tennis
The TennisApplet creates a Game which has a SingleScoreConverter and two Players.

might look like Figure 7.1.

To score a tennis match, something must keep track of the score in individual games, the number of games each player has won in each set, and the number of sets each player has won. At the beginning of each game, the score must be set to 0-0 and there must be code to specify who is the server this game. At the beginning of each set, the games each player has won must be set to zero as well. Plus, there must be code to decide when a game, a set, and a match is over.

This various logic might be in one class, or distributed over several as in Figure 7.2. By building Match, Set, and Game classes the initialization for each can be handled by constructors.

Figure 7.2



Booch diagram for a tennis match

A Match has 2 Players and up to 3 Sets. A Set has n Games (at least 6).

iii) Making it smaller; let's just play a single game

If the description sounds like, and the Booch diagram looks like rather a lot to take on to start, then you know what to do. Make it simpler! Build a prototype first. Figure 7.1, with just a single game seems like a reasonable place to start.

a) Game and Player classes

So, create Player and Game classes; write the code and test it. The Player and Game classes are most easily made with the classmaker. Add a Game variable in the Applet; initialize it in `init()`, and send it messages when the user pushes the Buttons. The Game should announce the score before each point. For now simply have it write to System.out. Don't worry about deuce scoring, or game ending. If you need a hint, see Code Example

Code Example 7.15

```
1 public class TennisApplet extends java.applet.Applet {
2    Game theGame;
3
4    /** Initializes the applet TennisApplet */
5    public void init() {
6        initComponents();
7        theGame = new Game(new Player("Serena"), new Player("Maria"));
8    }
9
     ...code deleted...
10
11   private void serButtonActionPerformed(java.awt.event.ActionEvent evt) {
12        theGame.serverScored();
13    }
14
15   private void recButtonActionPerformed(java.awt.event.ActionEvent evt) {
16        theGame.receiverScored();
17   }
```

The TennisApplet

Line 2: Declare the Game variable
Line 7: Instantiate a Game and store it.
Lines 12 and 16: When the user pushes a button send the appropriate message to the game.

7.15, but try to do it yourself as much as possible. Keep in mind that if you've added the Image code, your TennisApplet will look slightly different.

The Game class needs variables for each player's score, and methods to increment each of them when the user pushes that Button. It also needs a way to report the score. You probably know how to write the Game class. My preliminary class is in Code Example 7.16. Compile and run these two classes; push Buttons until both scores are out of range.

Code Example 7.16

```
1  public class Game {
2     Player server, receiver;
3     int receiverScore, serverScore;
4     SingleScoreConverter theConverter;
5
6     /** Creates a new instance of Game */
7     public Game(Player server, Player receiver) {
8         this.server = server;
9         this.receiver = receiver;
10        theConverter = new SingleScoreConverter();
11    }
12
13    public void receiverScored() {
14        receiverScore++;
15        announceScore();
16    }
17
18    public void serverScored() {
19        serverScore++;
20        announceScore();
21    }
22
23    public void announceScore() {
24        System.out.println(theConverter.convert(serverScore) + "-"
25                          + theConverter.convert(receiverScore));
26    }
27}
```

The Game class

Lines 2-4: Variables
Line 7-11: The constructor saves the two Players and instantiates the converter.
Lines 13-16 and 18-21: Increment the correct score and announce the new score.
Lines 23-26: Announce the score.

Now it is time to add the code to handle deuce games and the game being over. But, before doing that, it will be more efficient to replace the System.out.println in `announceScore()` with code to write to the TextArea in the Applet. That println was for debugging, and the more printlns we add, the more we'll have to change for the final product.

b) Establishing the linkage back to the Applet's TextArea

If the Game had a reference to the TextArea back in the Applet, `announceScore()` could write to it directly. I.e line 24 in Code Example 7.16 could be `theTA.append()` instead of `System.out.println()`. But, where will Game get that reference from? This is a common problem for beginning object programmers, but the solution can be deduced with a little careful thought, assuming you have internalized the constructs covered thus far. Maybe you know how already?

For `theTA.append(...)` to compile in the Game class, Game must have a variable of type java.awt.TextArea named theTA. Easy enough, add: `java.awt.TextArea theTA;`

But, like all instance variables, it will be initialized to 0 which, for a reference is null, and although now `theTA.append(...)` will compile, when it run it will generate the infamous "Null pointer exception". So, the question becomes, how to set theTA to actually reference theTA back in the Applet? The answer is very easy, once you realize it. How does one set the value of a variable in one class from another? Use an accessor. The ClassMaker will write it for you, or you could just type it; whatever is easier.

When should you set theTA in Game? Since it only needs to be done once, it should be done in the constructor, or right after the constructor is invoked; see Code Example 7.17.

Code Example 7.17

```
1   public void init() {
2       initComponents();
3       theGame = new Game(new Player("Serena"), new Player("Maria"));
4       theGame.setTheTA(theTA);
5   }
```

Establishing a reference back to the TextArea from theGame.
Line 4: set theTA in theGame to theTA (so that you can write in it from there).

Make those changes and test your code.

c) Adding game over and deuce scoring code

The `announceScore()` method in Code Example 7.16 assumes that the game is still in progress and the first deuce (40-40) has not been reached. It remains to write code to detect and announce these other two cases, so that we can write `announceScore()`. This is a perfect place to use the ABC method (see "Analysis By Cases (ABC)" on page 161). If you wanted to make sure you've learned this technique, this would be a good time to practice it. On the other hand, it is possible to delay making detailed decisions by pretending there are methods that can distinguish between the cases. This is illustrated in Code Example 7.18. The use of methods makes the code legible. This way the code is

Code Example 7.18

```
1     public void announceScore() {
2        if (gameOver())
3            announceGameOver();
4        else if (simpleScore())
5            announceSimpleScore();
6        else announceDeuceScore();
7    }
```

announceScore() with methods

There are three different cases for the score in tennis: game over, simple scoring, and deuce scoring. This `announceScore()` distinguishes between them by using methods in a cascaded if-else.

simple and easy to understand; the details of whether the game is over or when we can use simple scoring is hidden in the methods.

- **The gameOver() method**

How to write the `gameOver()` method? Oddly, most of it can be written syntactically. I.e. you can do most of the work of writing it given what you know about syntax. Look at line 2 in Code Example 7.18. From that you can infer the type of the `gameOver()` method. Since its type is not void, it must return a value of that type. If you don't know the type, look back at the syntax of an if statement (BNF 7.1 "The if statement" on page 158). With that in mind, a prototype of `gameOver()` can be written while ignoring the details of its logic; see Code Example 7.19. Now we must fill in the logic so that the method will

Code Example 7.19

```
public boolean gameOver() {
    return false;
}
```

gameOver() prototype

The type of `gameOver()` must be `boolean`; otherwise it could not be used in the context `if (gameOver())`. Since it is not void, it must return a value. This prototype method would compile and run, but would always (as you can see) return false.

return true when the game *is* over.

- **Writing simple code**

Logically, the game is over if either player has at least 4 points and is at least 2 points ahead. The first thing some programmers think to write is that the game is over

```
if (serverScore >= 4 && serverScore - receiverScore >=2 ||
    receiverScore >= 4 && receiverScore - serverScore >=2)
```

which is a mouthful. If you were going to write this, it would be good to enclose it in a method, see Code Example 7.20, so that you could use it from various places. This code

Code Example 7.20

```
public boolean gameOver() {
    return (serverScore >= 4 && serverScore - receiverScore >=2 ||
        receiverScore >= 4 && receiverScore - serverScore >=2);
}
```

gameOver() first try

Logically, this method is correct, but compare it for simplicity and ease of understanding with Code Example 7.21.

is not wrong, but it is a bit complicated and difficult to read. It is more elegant to write the test more abstractly as in Code Example 7.21. This way is less prone to errors and

Code Example 7.21

```
public boolean gameOver() {
    return serverWon() || receiverWon();
}
```

gameOver() made simple

A simple, easy to read version of `gameOver()`.

easier to modify when there are errors. This method is far superior in terms of debugging and clarity, but it has a price; now there are two more methods to write.

- **Combining nearly identical methods**

One’s first impulse might be to simply copy and paste the logic from the two lines into the two methods as in Code Example 7.22. But, note that the logic of those two methods

Code Example 7.22

```
public boolean serverWon() {
    return serverScore >= 4 && serverScore - receiverScore >=2;
}

public boolean receiverWon() {
    return receiverScore >= 4 && receiverScore - serverScore >=2;
}
```

serverWon() and receiverWon(), take 1

Notice the logic is identical in the two methods with the variables serverScore and receiverScore reversed.

is identical, with the role of the variables serverScore and receiverScore reversed. Anytime you discover nearly identical code like this you can combine it. The resulting methods are shown inCode Example 7.23. Both methods simply return the value of

Code Example 7.23

```
public boolean serverWon() {
    return winner(serverScore, receiverScore);
}

public boolean receiverWon() {
    return winner(receiverScore, serverScore);
}
```

serverWon() and receiverWon(), take 1

The commonalities are combined by calling the same method with the variables serverScore and receiverScore reversed.

winner, but with the actual parameters exchanged. This is a useful technique to learn; it has the benefit of putting the decision of whether one player or the other has won the game in the same code. This is useful because if the logic is wrong, it is only wrong in one place, and can be fixed in one place. The downside is that now `winner(int, int)` must be written; see Code Example 7.24. Notice that it returns true just if the first

Code Example 7.24

```
public boolean winner(int x, int y) {
    return x >= 4 && x >= y;
}
```

winner()

A player with score x wins over a player with score y if x is at least 4 and is at least 2 more than than y.

parameter, x, is at least 4 and at least 2 more than the second, y. This concludes `gameOver()`. Writing it produced several other methods, but they are all one line long. Looking back at Code Example 7.18, it remains to write `announceGameOver()`, `simpleScore()`, `announceSimpleScore()`, and `announceDeuceScore()`.

- **simpleScore()**

As long as we have not reached the first deuce score, then we can use the simple score announcer from before. The first deuce score is 3-3, so as long as either score is less than

3, simple scoring will work (see Code Example 7.25). The `announceSimpleScore()`

Code Example 7.25

```java
public boolean simpleScore() {
    return receiverScore < 3 || serverScore < 3;
}
```

simpleScore()

If either score is less than 3 we can use simple scoring.

method is just the old `announceScore()` method, as shown in Code Example 7.26.

Code Example 7.26

```java
public void announceSimpleScore() {
    theTA.append(theConverter.convert(serverScore) + "-"
                     + theConverter.convert(receiverScore) + "\n");
}
```

announceSimpleScore()

Simply add the current score to theTA.

- **announceGameOver() & announceDeuceScore()**

The method `announceGameOver()` is shown in Code Example 7.27. It checks who won and announces that. Finally, `announceDeuceScore()` is shown in Code Example 7.28. There

Code Example 7.27

```java
public void announceGameOver() {
    if (serverWon())
            theTA.append("Game Server!\n");
    else theTA.append("Game Receiver!\n");
}
```

announceGameOver()

The game is over; announce who won.

Code Example 7.28

```
public void announceDeuceScore() {
     if (receiverScore==serverScore)
         theTA.append("Deuce\n");
     else if(receiverScore < serverScore)
         theTA.append("Advantage Server\n");
     else theTA.append("Advantage Receiver\n");
 }
```

announceDeuceScore()

There are three cases; deuce, advantage server and advantage receiver.

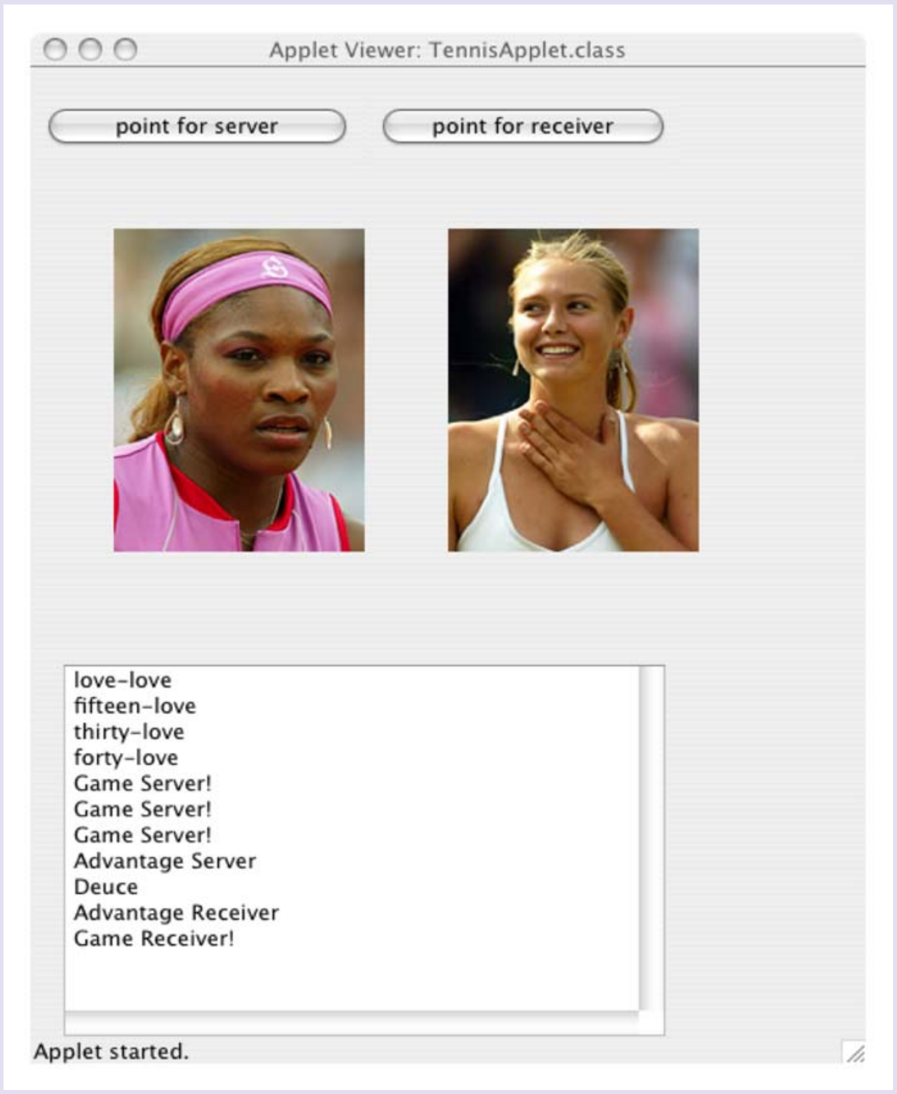
are three cases in deuce scoring, so the correct construct is a cascaded if-else. If it were difficult to decide how to write the code, ABC would be relevant.

That's all the code. There were quite a few methods, but they are all quite simple. There is a trade-off between writing simple, easy to debug code and the number of classes and methods one must type. The typing is rather tedious, but the time spent designing and implementing a simple solution will be paid back many times by the time not spent debugging, especially in complicated programs.

iv) Testing

Now it's time to test the play one game code. After you fix all the small mistakes, you should see it displaying in the TextArea. There are two problems with the code written above (see Figure 7.3). First, it does not announce the score before the first point, i.e. it

Figure 7.3



```
love-love
fifteen-love
thirty-love
forty-love
Game Server!
Game Server!
Game Server!
Advantage Server
Deuce
Advantage Receiver
Game Receiver!
```

The complete output for 4 presses of the left Button, then 6 of the right. Two bugs are apparent.

never says love-love. Second, after the game is over, it still allows points to be scored. The output shown is for four clicks of "point for server" then six of "point for receiver". It would be better to stop scoring after the game is over! Fortunately, because the code is written well, these are both easy to fix.

d) Announcing the score before the first point

Fixing bugs can be easy or difficult. The better you understand the code the easier it is. The simpler the code, the easier it is to understand what's going wrong. Well written code has modules that make it easy to modify.

The reason it does not announce the score before the first point is obvious if you look at the code. The only time it announces the score is when the user pushes a Button (check it, it's true). So, how to announce the score before the first point?

It would be easy to cause the game to announce the score if there were a method that announced the score in Game. Is there?

Where should the announceScore message be sent to theGame? Look at the TennisApplet code you've written. Where should you send that message? In `init()`, right after theGame is created, see Code Example 7.29.

Code Example 7.29

```
1    public void init() {
2        initComponents();
3        theGame = new Game(new Player("Serena"), new Player("Maria"));
4        theGame.setTheTA(theTA);
5        mariaImage = getImage(getCodeBase(), "maria.jpg");
6        serenaImage = getImage(getCodeBase(), "serena.jpg");
7        theGame.announceScore();
8    }
```

announcing the initial score

Line 7: Announce love-love.

e) Preventing points after the game is over

When the user pushes a "point for" Button, you don't always want to add a point; only sometimes. That's what if statements are for. You should only add a point if the game is not over. Here's the payoff from having written the `gameOver()` method. You can use it from the Applet to check if the game is over and only add a point if it is not, see Code

Example 7.30. Make these changes (guard both the Buttons!) and rerun your code. With

Code Example 7.30

```
1   private void  serButtonActionPerformed( java.awt.event.ActionEvent evt)
       {
2        if (!theGame.gameOver())
3            theGame.serverScored();
4  }
```

only add points if the game is not over - !over

Line 2: Guard sending the `serverScored()` message with `!theGame.gameOver()`. I.e. if the game is over, do not send the `serverScored()` message.

any luck, you now have a working Tennis game.

D. Conclusion

The if statement allows different statements to execute depending on the situation. It has two forms, if without an else and if-else. If you want your program to do something only under certain conditions, use an if statement.

```
    if (condition is true)
        doSomething
```

If you want your program to do either one thing or another, use an if-else statement

```
    if (whatever condition determines when to do the first thing)
        oneThing();
    else anotherThing();
```

Multiple cases can be handled either by a cascaded if-else, or a switch statement. In complex cases, the Analysis By Cases (ABC) technique can help clarify your thinking.

E. End of chapter material

i) Review questions

7.1 If you want to either execute a statement or not depending on some condition, what statement do you use?
7.2 If you want to execute either one statement or another depending on circumstances, what statement do you use?
7.3 Why is the if statement called a conditional statement?
7.4 Write a statement that prints "yes" is x is greater than zero and "no" otherwise.

7.5 Reverse the logic in Code Example 7.6. I.e. exchange the statements before and after the else; then reverse the logic of the boolean expression so it still does the right thing. If in doubt, make a truth table.

7.6 Make a truth table for (!p||q) && !(r&&!q)

7.7 In craps, on the first roll, if you roll 7 or 11, you win, if you roll 2 or 12, you lose. Otherwise, you must roll the same value you rolled the first time before you roll a 7 to win. Write an cascaded if-else statement that tests a variable rollValue and prints one of three things: "you win!", "sorry, you lose", or "roll again" depending on its value.

ii) Programming exercises

7.8 The dangling statement. Here's some code with what may be a subtle bug. What would it print if age were 14? If age were 41? How to fix it?

```
if (age > 18)
    System.out.println("Major");
else
    System.out.println("Minor");
    System.out.println("You may not enter!");
```

7.9 Write a method named exclusiveOr, which is passed two boolean parameters and returns true if exactly one of them is true.

7.10 An exam is graded as follows: 91-100: A, 81-90: B, ...61-70: D, <61: F. Write a method that is passed an int score and returns the appropriate grade as a String. First use an if-else, then a switch statement. Hint: what values would `(score-1)/10` take on for various scores? Remember how integer division works.

7.11 Modify announceWinner to announce the winner's name. Like "Game Serena!". Hint: use `getName()`.

7.12 Add an error message if the user pushes a Button after the game is over.

7.13 A better solution is to disable the Buttons when the game is over. Do that. Perhaps the easiest way is to send them the `setVisible(false)` message.

7.14 Modify your code so that it plays a set instead of a game. Create a Set class; model it on Game. Play an entire set.

7.15 Modify your code so that it plays a match instead of a set.

Chapter 8: Iterative statements and Strings

A. Introduction

i) Repetition

Repetition is part of life. Everyone is familiar with it. The sun comes up, the sun goes down. The sun comes up, the sun goes down. Spring turns to summer; fall arrives and students gather in classrooms - over and over. Babies grow to children, mature to adults, have more babies, and die; for the last million years or so. Civilizations rise and fall; again and again. The galaxy turns, but very slowly to our eyes.

ii) On being conscious

If parents are unthinking, not conscious of what they are doing, they tend to raise their children as they were raised. They do the things that were done to them as children to their own children. This is not always optimal. Similarly, teacher, if they have not reflected on the process of teaching and learning, may have their student do the same tedious, unproductive exercises they were forced to do. If the world, or the field has changed, this is sometimes not the best plan.Repetition, for it's own sake is a bad idea.

iii) Imitation and culture, or monkey see, monkey do?

Researchers have shown that humans are much better at imitation than other great apes, even when it makes no sense. Here is a revealing experiment. The subjects (human and chimpanzee) watch a person trying to reach a banana from behind bars. It is out of reach, but there is a rake-like tool lying at hand. On one side, it has 4 tines, on the other 2. The person picks it up and drags over the banana with the 2 tined side. Then the subject is then put behind the bars, and a new banana is placed out of reach. Both people and chimpanzees use the rake to get the banana, but the people use the 2 tined side much more often than the 4 tined side, even though it is much more difficult than the other side. The question is, why?

One hypothesis is the people are biased, unconsciously, towards doing things the way they have seen them done. Why might this be a species trait? One explaination is that this might lead to the emergence of language and culture. If one group of protohumans tends to do things and express things the same way automatically, and another doesn't, the former tends to develop coherent language, which allows them to communicate better and gives them tremendous adaptive advantage. Same with culture in general. Whether of not this is true, it is an interesting speculation.

iv) This chapter

This chapter introduces Java statements that repeat. Together with conditional statements, these comprise the bulk of control structure in Java. Good control structure, coupled with well designed class structure, can yield powerful and useful software.

Unlike other chapters, this one will be almost devoid of classes. It focuses on the mechanics of loops and Strings. Perhaps you can learn this material simply by reading the text, but it may help these constructs stick in memory if you type them into a program and run them (and they are all essential to programming). This would also give you a chance to experiment with small changes to them; to play around with them. That's the best way for most people to learn.

The previous chapter introduced conditional statements; this one will introduce the other statements that implement control structure, iterative statements. It also will discuss Strings in more detail.

B. Iteration: Repeated action

Iterate is another word for repeat. To make a program to do something a number of times use an iterative statement. There are three iterative statements in Java; see BNF 8.1 This

BNF 8.1 Iterative statement

<iterative stmt> ::= <while stmt> | <for stmt> | <do-while stmt>

chapter will cover while and for.

C. The while loop

i) Syntax and semantics

The syntax and semantics of a while statement are shown in BNF 8.2. Notice that the

BNF 8.2 The while statement

```
<while stmt> ::= while (<boolean expression>) <stmt>
```

Semantics

1: Evaluate the <boolean expression>

2: If the value of the expression is true, execute the <stmt> after <expression>

3: Go back to step 1.

syntax is very much like an if statement (see BNF 7.1 "The if statement" on page 158). The difference in semantics is that after the <stmt> is executed, the <boolean expression> is evaluated again, and if it is still true, that process repeats, possibly forever. The programmer must remember to make sure that while loops are not infinite!

ii) Examples

a) Counting to 10

Code Example 8.1 shows a while loop that prints the numbers from 1 to 10. This loop

Code Example 8.1

```
1 int count=1;
2
3 while (count<=10) {
4     System.out.println("count=" + count);
5     count++;
6 } // while
```

Counting to 10

Line 1: declare an int variable named count and set it to the value 1.
Line 3-6: the while loop
Line 3: the loop will run while count <= 10
Line 4: Print the value of count (along with a descriptive label).
Line 5: Add one to count.

runs over and over until the boolean expression `count<=10` is false at the top of the loop (it is only rechecked before the entire loop body executes). So, the execution of the loop is controlled by the value of count. Thus, count is the ***control variable*** for this loop. In this case the control variable is also being used to display the count; sometimes the control variable only controls the number of times the loop executes. If you omit line 5, this will print `count=1`, forever! This is called an ***infinite loop***.

One way to understand what a section of code does is analysis; simply look at it and see what it does. If that works, fine. If that doesn't work, one way to discover what it does is called ***hand simulation***. To hand simulate code, you need paper and pencil. Write down the variables involved and step through the code, carrying out the semantics of each statement one by one, recording the values of the variables as you go. Table 8.1 shows the hand trace for Table 8.1; make sure you understand it; then do it yourself. It is

Table 8.1:

line	action	count
1	declare count	1
3	count<=10? yes, do body	1
4	output count=1	
5	count++	2
3	count<=10? yes, do body	2
4	output count=2	2
5	count++	3
3	count<=10? yes, do body	3
...	...	...
5	count++	10
3	count<=10? yes, do body	10
4	output count=10	10
5	count++	11
	count<=10? no!, done!	11

slow work, but sometimes it provides insight into the mysterious inner workings of code.

b) Doing something 10 times

Code Example 8.2 is a generic loop to do something 10 times. What that something is is

Code Example 8.2

```
int count=1;

while (count<=10) {
    something();
    count++;
} // while
```

doing something 10 times

Because it is sitting in a loop that goes around 10 times, something() will be executed 10 times.

up to the programmer.

For example, this loop can be used to output the squares and cubes of the numbers from 1 to 10 as shown in Code Example 8.4. This code does what it is supposed to, but the

Code Example 8.3

```
int count=1;

while (count<=10) {
    sqAndCube(count);
    count++;
} // while

 …

void sqAndCube(int x) {
   System.out.println("x=" + x + "x^2=" + x*x + "x^3=" + x*x*x);
}
```

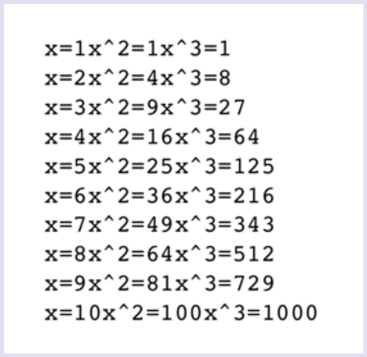
Using that while loop to print a list of squares and cubes.

The control variable is passed as the value to square and cube each time. Notice that there are some formatting issues.

output doesn't look very nice. Figure 8.1 has the output for this loop from my machine.

Although there are spaces in the `println()`, there are none inside the ""s. That println

Figure 8.1



```
x=1x^2=1x^3=1
x=2x^2=4x^3=8
x=3x^2=9x^3=27
x=4x^2=16x^3=64
x=5x^2=25x^3=125
x=6x^2=36x^3=216
x=7x^2=49x^3=343
x=8x^2=64x^3=512
x=9x^2=81x^3=729
x=10x^2=100x^3=1000
```

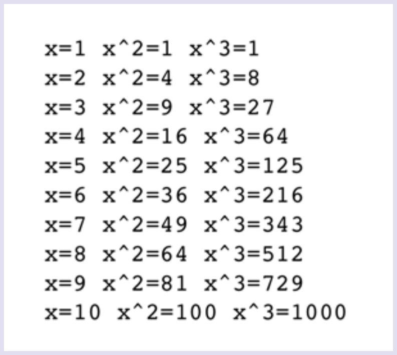
Output from Code Example 8.3 -- obviously in need of spaces.

would create better looking output if it had a few spaces; like this:

```
    System.out.println("x=" + x + " x^2=" + x*x + " x^3=" + x*x*x);
```

Figure 8.2 has the output with that change. It is better, but the different widths of the

Figure 8.2



```
x=1 x^2=1 x^3=1
x=2 x^2=4 x^3=8
x=3 x^2=9 x^3=27
x=4 x^2=16 x^3=64
x=5 x^2=25 x^3=125
x=6 x^2=36 x^3=216
x=7 x^2=49 x^3=343
x=8 x^2=64 x^3=512
x=9 x^2=81 x^3=729
x=10 x^2=100 x^3=1000
```

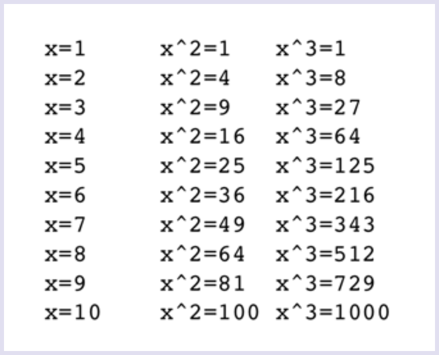
Slightly better output from Code Example 8.3 -- still in need of tabs.

larger numbers rather wreck the readability. This is a good place to use the tab character (`\t`):

```
    System.out.println("x=" + x + "\tx^2=" + x*x + "\tx^3=" + x*x*x);
```

Output from this appears in Figure 8.3 and looks rather better. The general problem of

Figure 8.3



```
x=1     x^2=1   x^3=1
x=2     x^2=4   x^3=8
x=3     x^2=9   x^3=27
x=4     x^2=16  x^3=64
x=5     x^2=25  x^3=125
x=6     x^2=36  x^3=216
x=7     x^2=49  x^3=343
x=8     x^2=64  x^3=512
x=9     x^2=81  x^3=729
x=10    x^2=100 x^3=1000
```

Output from Code Example 8.3 -- with tabs.

formatting text from Java will be postponed indefinitely.

c) Doing something n times

In the more generic loop in Code Example 8.4, the control variable is compared to

Code Example 8.4

```
int N=100000;
int count=1;

while (count<=N) {
    something();
    count++;
} // while
```

A generic while loop

This loop does `something()` N times. Suitable for memorization.

another variable, N, instead of 10. This can make a big difference in a more complex situation. Changing a 10 to 1000 is easy, changing 8 10s to 1000s less so; and what if you forget one of them?

d) Counting to 10 - take II

Code Example 8.1 started the value of count at 1, like a person would. But, it is common for loops in programs to start from 0 instead, for reasons that will be seen below. Code

Code Example 8.5

```
1 int count=0;
2
3 while (count<10) {
4     count++;
5     System.out.println("count=" + count);
6 } // while
```

Counting to 10 - take II

This also prints the numbers 1-10; but using slightly different logic.

Example 8.5 shows the same loop counting from 0. There are three changes: count is initialized to 0, the expression is (count<10) instead of (count<=10) and lines 4 and 5 are exchanged. Convince yourself that this loop does the same thing as the other.

e) Infinite loops

The while loop gives the programmer tremendous power. But with increased power comes increased capacity for error. If the control variable is never updated inside the loop, it is always an infinite loop. If the control variable is not correctly updated, it may be an infinite loop.

D. The for loop

Code Example 8.6 does exactly what Code Example 8.4 did, but uses a for loop instead

Code Example 8.6

```
int N=10

for (int count=1; count<=N; count++) {
    something();
} // for
```

doing something 10 times with a generic for loop

Notice that initialization, checking and incrementing the control variable, all happen in the first line of the for. This makes them much more difficult to forget.

of a while loop. It is a bit more compact than the equivalent while loop, and until you are used to it, more confusing. One young programmer who was familiar with while loops but not for loops and refused to learn for loops because, "They are too complicated!". This was a mistake.

Sometimes learning one thing makes another more difficult to learn. This is called proactive interference. The effect is particularly strong when you know one way to do something and someone wants you to learn another. The way you know (in my case, the while loop) seems simple and obvious; the new way difficult and confusing. When you try to solve a problem with a new technique, you immediately know how to solve it with the old technique (which interferes with applying the new technique). So you resist the new way. This is true at all scales; from iterative statements to programming languages and paradigms, operating systems, and even life-styles. But, change is good; if you stop learning, your life is essentially over. Oops, back to the for loop.

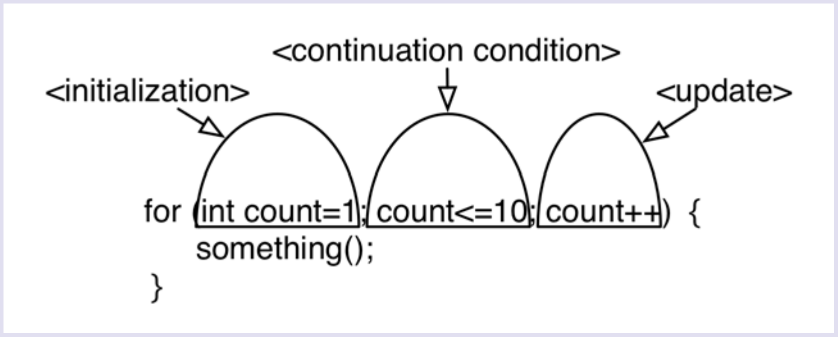
The syntax and semantics of a for loop is very similar to that of a while loop. As BNF 8.3

BNF 8.3 The for statement

<for stmt> ::= for (<initialization>;<continuation condition>; <update>) <stmt>
Semantics
1: Execute <initialization>.
2: If the value of the <continuation condition>. is true, execute the <stmt>
3: Execute <update>.
4: Go back to step 2 (that's right, step **2**!).

shows, it starts with the word for, then some things in ()s then a single statement. There are three things in the parentheses (see Figure 8.4); the one in the middle, the

Figure 8.4



```
for (int count=1; count<=10; count++) {
      something();
}
```

For loop illustration.

continuation condition, is a boolean expression (see BNF 8.3), exactly like the while

<initialization> ::= <stmt> <continuation condition> ::= <boolean expression> <update> ::= <stmt>
Both <initialization> and <update> are simply statements, the <continuation condition> is simply a boolean expression

loop. The first thing, <initialization> statement, is done once, before anything (like initialization always is!). The third, <update>, is done after the body of the loop is executed (each time it is executed).

The big advantage of a for loop is that the initialization and update are included in the statement. That way they are easy to find and hard to omit accidently.

i) Example

a) Task:

Write a program that will count to 1000 by twos, displaying the count to `System.out`.

Several methods might be employed to accomplish this task. First, since we already have a loop that counts to 10 by 1's, we could simply change the 10 to 1000 and only print the even numbers, as in Code Example 8.7. Notice here that the first part of the String

Code Example 8.7

```
int N=1000;

for (int count=1; count<=N; count++) {
    if (count is even)
        System.out.println("" + count);
} // for
```

Counting to 1000 by 2's - method 1 concept

The for loop iterates 1000 times, with count going from 1 to 1000. Only even numbers print.

parameter is "". That way it doesn't say "count=" every time. On some compilers if you omit that and just write `System.out.println(count);` it will generate a compiler error complaining that an int is not a String (recall "int to String" on page 123).

How can we determine if count is even? Here's a hint: even numbers are evenly divisible by 2. Need another hint? The % operator yields the remainder after int division. One more hint? If a number divides another evenly, the remainder is zero. Okay?

A second approach to this problem is to change the update so that each time around the loop it adds 2 to count instead of 1. I.e. change the `count++` to `count=count+2`; -- see Code

Code Example 8.8

```
int N=1000;

for (int count=2; count<=N; count=count+2) {
    System.out.println("" + count);
} // for
```

Counting to 1000 by 2’s - method 2

By changing both the initialization and the update, we get clean simple code.

Example 8.8.

Third, we could just count from 1 to 500 and print twice the count each time -- see Code

Code Example 8.9

```
int N=500;

for (int count=1; count<=N; count++) {
    System.out.println("" + count*2);
} // for
```

Counting to 1000 by 2’s - method 3

Count goes from 1-500, count*2 is printed each time.

Example 8.9. Which of these methods would be better? It’s a matter of style.

b) Cleaning up the output.

If you run that code, it produces 500 lines of output. That’s a bit excessive. it would be nicer if more than one number were printed on each line. The `println()` method ends the line. There is another method, `print()`, that does not end the line. Thus, the simple solution would seem to be to send `print()` instead of `println()` to `System.out`; do you know what would go wrong in that case?

Using `print()`, all the numbers are concatenated without any spaces on one line. The beginning of the output is shown in Figure 8.5. That's two problems. The first is very

Figure 8.5

```
2468101214161820222426283032343638404244464850525456586062646668707274767880
828486889092949698100102104
```

Using print() instead of println() produces more compact output. Possibly too compact!

easy to solve; see Code Example 8.10. The second is more complicated and not entirely

Code Example 8.10

```
for (int count=2; count<=N; count=count+2) {
    System.out.print(" " + count);
} // for
```

Counting to 1000 by 2's - on one line

Notice the print() instead of println(). The space in the ""s puts spaces between the numbers.

obvious how to solve.

- **Printing 10 numbers per line**

One idea would be to print a fixed number of numbers on each line; say 10. Pseudo-code for this is shown in Code Example 8.11. How can we tell if 10 numbers have been

Code Example 8.11

```
for (int count=2; count<=N; count=count+2) {
    System.out.print(" " + count);
    if (have printed 10 on this line)
        System.out.println();  // go to next line
} // for
```

Pseudocode for printing 10 numbers on each line
How shall we keep track of how many numbers have been printed on this line?

printed on this line? Here's two possibilities: 1) add a counter; initialize it to 0, and each time we print a number, add one; when we do the `println()`, reset it to 0. Or, 2) Figure out some clever way to discern when 10 numbers have been printed. The former is rather more code, but more general; the latter is, well, clever, but likely to be hard to generalize (i.e. use in another context).

- **Adding another counter**

With a counter, the pseudocode appears in Code Example 8.12. Of course, the variable

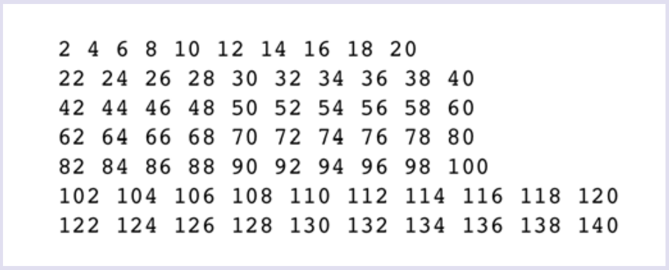
Code Example 8.12

```
for (int count=2; count<=N; count=count+2) {
    System.out.print(" " + count);
    wordCount++;     // increment wordCount
    if (wordCount == 10) {
        System.out.println();  // go to next line
        reset wordCount to 0;
    } // if
} // for
```

Pseudocode for printing 10 numbers on each line with a counter
Realize that wordCount must be initialized outside the loop before it starts.

wordCount must be declared and initialized to 0 before the loop. This produces the output shown in Figure 8.6.

Figure 8.6



```
2 4 6 8 10 12 14 16 18 20
22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60
62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94 96 98 100
102 104 106 108 110 112 114 116 118 120
122 124 126 128 130 132 134 136 138 140
```

Ten even numbers on each line; too bad the alignment is so bad.

- **A clever trick**

If there were some property shared by all the final numbers on a line, the code could check that property instead of keeping a counter to tell when it is time to end the line. Looking at that output, can you see anything unique about the last number on each line? Right, they are all multiples of 20. So, to tell if it is time to end the line we could just check if `count%20==0`, as in Code Example 8.13. This kind of coding trick is fun, but can

Code Example 8.13

```
for (int count=2; count<=N; count=count+2) {
    System.out.print(" " + count);
    if (count % 20 == 0)
        System.out.println();  // go to next line
} // for
```

Pseudocode for printing 10 numbers on each line with a trick.

Since every 10th number is a multiple on 20, this works too.

lead to disaster if later the code must be modified. Nonetheless, it is useful to remember.

ii) The empty statement

Here’s my favorite statement, the empty statement. BNF 8.4 shows its syntax and

BNF 8.4 The empty statement

<empty stmt> ::=
Semantics
Does nothing

semantics. It consists of nothing and does nothing. What good is it? It allows the programmer to omit a statement in a place where syntactically a statement must appear and still match the grammar. In other words it allows the programmer to get around the rigidity of the compiler, which can be a very good thing.

iii) An infinite for loop

The format of a for loop makes it difficult to forget to update the control variable and so, one is less likely to write an infinite for loop. But, sometimes you want to write an infinite for loop (as you will see in the next chapter). Code Example 8.2 shows how to

Code Example 8.14

```
for (;;) {
    something();
} // for
```

doing `something()` forever

This for loop will execute forever; doing `something()` over and over.

send the something() message forever. Note that the {}'s are not necessary. The line `something();` is a message statement, so Code Example 8.15 is legal as well.

Code Example 8.15

```
for (;;)      // forever!
    something();
```

Code Example 8.14 without the {}s

This is also legal because `something();` *is* a single statement.

The empty statement sometimes causes nasty bugs (yet another example of the fact that most things have two sides). Can you see what is wrong with the code in Code Example

Code Example 8.16

```
for (;;);    // forever!
    anything();
```

An infinite loop that doesn't do anything

Can you see why?

8.16? If you don't see it immediately, try matching the symbols one by one with the BNF 8.3. It starts like this: for matches for, (matches (, absolutely nothing matches the empty statement, which is a <statement> which is an <initialization>...

If you ran this code it would never do anything, but unlike the empty statement, which does nothing immediately, it would do nothing, forever!

E. Strings: a very brief introduction

The Java String class is used to store literal sequences of characters. There are many methods that operate on Strings; you can read about them in the documentation; it's about time you got used to reading Sun's API documentation. This section will introduce a bare minimum of String methods with the signatures: `int length()`, `char charAt(int)`, `boolean equals(String)`, and `String toUpperCase()`. Notice that these four methods return `int`, `char`, `boolean` and `String` values, respectively. Then it will present a better way of breaking lines of even numbers.

i) A few String methods

`a) int length()`

As you might guess, if you send `length()` to a String, it will return its length. So the code:

```
    String s = "hello";
    System.out.println("s=" + s + " s.length()=" + s.length());
```

will print `s=hello s.length()=5`. No surprises.

`b) char charAt(int)`

A String is a series of characters. The Java type for characters is `char;` each `char` holds a single character. Character constants have single quotes around them, like 'a' or '$'. The first character in a String is at location 0. That's zero, not one, but*zero*. Forgetting this will cause little annoying bugs. Rather like mosquitoes. Not really harmful, but pesky. The reason the first char is at zero is that Strings are implemented as arrays, which are coming up in a few chapters. Arrays in Java, like in C++ and C, start at zero. No way around it. Deal with it. Accept it. Remember it!

With `String s="hello";` the following are true:

```
    s.charAt(0) returns 'h'
    s.charAt(1) returns 'e'
    s.charAt(2) returns 'l'
    s.charAt(3) returns 'l'
    s.charAt(4) returns 'o'
```

You could write a loop to produce those five lines. Pseudocode for it is shown in Code

Code Example 8.17

```
String s="Howdy!";
for (each char in s) {
    output the next char with suitable description
} // for
```

Pseudocode to print all the chars in a String

See Code Example 8.18 for the code that implements this.

Example 8.18. This is a pattern for many methods that process Strings. If you wanted to become expert at Java programming it would be worth committing to memory; both the pseudocode and the code. Your choice.

When you wish to access each of the chars in a String from first to last in order, you use the idiom shown in Code Example 8.18. Line 2 there is the generic for loop you write to

Code Example 8.18

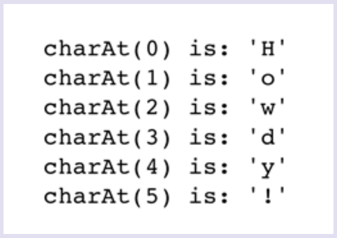
```
1  String s="Howdy!";
2  for (int i=0; i<s.length(); i++) {
3      System.out.println("charAt(" + i + ") is: '" + s.charAt(i) + ' ');
4  } // for
```

Printing the chars in a String one per line

Notice the various parts of the String parameter that produce the readable output.

access the chars in the String, s, sequentially. In the body of the loop, `s.charAt(i)` is each char, one after another each time around the loop. In pseudocode `s.charAt(i)` is "the next char". The output of Code Example 8.18 is shown in Figure 8.7. Notice how the String

Figure 8.7



```
charAt(0) is: 'H'
charAt(1) is: 'o'
charAt(2) is: 'w'
charAt(3) is: 'd'
charAt(4) is: 'y'
charAt(5) is: '!'
```

Output from code in Code Example 8.18

"`charAt(0) is: `" is constructed in the `println()` on line 3. It is inconvenient, but highly informative to output such descriptive text along with the data one is outputting. If you are printing chars, the delimiters can be surprisingly important. The difference between printing nothing and printing an invisible character is important, but invisible. The ’s bracketing the char let the reader see this difference.

`c) boolean equals(String)`

You cannot compare Strings with ==. Instead you must use `equals()`. Worse, if you forget, and use ==, no error message will appear, because Java will compare the memory locations of the two Strings. Not what you expected, or wanted. So, if you wanted to know if the same text is in two TextFields, you must write code like:

```
String s1 = aTF.getText();
String s2 = bTF.getText();
if (s1.equals(s2))
    "yes! they are the same";
else "no. not equal"
```

`d) String toUpperCase()`

This method does what you might expect, it makes an uppercase copy of the String it is sent to and returns that. So:

```
String s = "abcDEFghi";
System.out.println(s.toUpperCase());
```

would output: `ABCDEFGHI`.

There are many useful methods that you should learn if you were going to do any extensive programming with Strings, including: `substring()`, `replaceAll()`, and `subSequence()`.

ii) Breaking lines using Strings

The technique to print 10 even numbers per line in Code Example 8.12 on page 204 worked, but it was rigid and produced somewhat ugly output. This example will do the same job more generally and elegantly. It will count to 1000 by 2's and print as many numbers on each line as will fit.

- **Decoupling the output**

An important step to writing elegant, general code is to decouple the logic and the output. This is true of all code. Input and output are inherently messy and idiosyncratic. A standard technique is to consolidate output in a method call `emit()`, as shown in Code

Code Example 8.19

```
void countTo_N_Improved() {
    for (int count=2; count<=N; count=count+2) {
        emit(" " + count);
    } // for
```

Using `emit()` to decouple the output

Each String to be output is sent to emit which handles the formatting and output.

Example 8.22. This way the logic of the loop is kept simple and whatever needs to be done with output Strings happens in one place, `emit()`.

- **Buffers**

An area of working storage is called a ***buffer***. The job of `emit()` here is to put as many output Strings on each line as will fit. It will accomplish this with an output buffer. The buffer will start empty. Whenever a String is sent to emit, if it fits in the buffer, it will be concatenated onto the end. If it overfills the buffer two things will happen: first the buffer will be output (or flushed, as is sometimes said), second the String will be added to the now empty buffer as the first thing on the next line. Pseudocode for `emit()` appears in

Code Example 8.20. Notice that lines 4 and 6 are identical. When the same thing is done

Code Example 8.20

```
1   void emit(String nextChunk) {
2       if (nextChunk would overfill the buffer) {
3           flush buffer
4           add nextChunk to the buffer
5       }
6       else add nextChunk to the buffer
7   }
```

Pseudocode for emit - first try

at the end of both the if and else parts of an if else you can do what is called ***bottom factoring***.

- **Bottom factoring**

Since either the if part or the else part of an if else is bound to execute, when the last thing in both is the same, it is always the last thing done before the next statement. Thus, it can be moved after the if-else. This is illustrated in Code Example 8.21. Having

Code Example 8.21

```
void emit(String nextChunk) {
    if (nextChunk would overfill the buffer)
        flush buffer
    add nextChunk to the buffer
}
```

Pseudocode for emit - after bottom factoring

Since the next chunk is always added to the buffer, this code does the same thing as Code Example 8.20. It is simpler and easier to read.

thought through the logic and written detailed pseudocode, it is fairly easy to write the actual code; see Code Example 8.22. Make sure you understand how this implements the

Code Example 8.22

```
1    final int MAX_LINE_LENGTH=70;
2    String buffer = "";
3    void emit(String nextChunk) {
4        if (buffer.length() + nextChunk.length() > MAX_LINE_LENGTH) {
5            System.out.println(output);
6            buffer = "";  // empty the buffer
7        }
8        buffer += nextChunk;
9    }
```

Breaking lines at 70 chars.
Line 2: declare and initialize the buffer.
Line 4: check if adding this next chunk would overfill it.
Line 5: if so; flush it - first output it...
Line 6: ...then empty the buffer
Line 8: always add the new output to the end of the buffer.

pseudocode; these are standard techniques you will need to know.

The output using this line breaking scheme is shown in Figure 8.8. It is much better

Figure 8.8

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94
96 98 100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130
132 134 136 138 140 142 144 146 148 150 152 154 156 158 160 162 164
166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198
```

Output from code in Code Example 8.18

balanced and adaptable than the original. This technique could be easily adapted to a word processing application.

F. Conclusion

This chapter introduced loops and Strings. Once you are familiar with classes, methods, instances, Strings, chars, conditional and iterative statements, you have the skills to solve a great number of problems. The next three chapters will introduce StringTokenizer, file I/O, and lists (both arrays and Vectors). Those are the last new topics in this text.

G. End of chapter material

i) New terms in this chapter

control variable - a variable that controls the execution of an iterative loop 187
hand simulation - simulating code by hand; performing the semantics of each statement, one by one to discover how code works, or when it is broken, why it doesn't 187
infinite loop - a loop that executes forever 187

ii) Review questions

8.1 What is the difference between a while loop and a for loop syntactically?
8.2 What can you do with a while loop that you can't do with a for loop?
8.3 If String s="stuff"; what is s.charAt(1)?
8.4 What is a buffer?
8.5 What is bottom factoring?
8.6 Hand simulate Code Example 8.5 on page 197.

iii) Programming exercises

8.7 Write and run the three methods for counting to 1000 by 2's.
8.8 Write a method, `String reverse(String)`, that returns its parameter backwards.
8.9 Write a method `int countEs(String)`, that returns the number of `'e'`s in the parameter.
8.10 Write a method `int countVowels(String)`, that returns the number of vowels in the parameter.
8.11 Write a method `boolean palindrome(String)`, that returns true just if its two parameter is the same backwards as forwards. E.g. "noon", "madam" and "aha" are palindromes.
8.12 Extend the previous method to handle spaces, punctuation and capital letters. E.g. "A man, a plan, a canal. Panama!" is a palindrome. Hint: use a filtering scheme. First make everything uppercase (`toUpperCase()`), then remove punctuation and spaces (`Character.isLetter(char)`), then use the method from above.
8.13 Write a method `boolean anagram(String, String)`, that returns true just if its parameters are anagrams.

8.14 Write a program to print all the print numbers less than 1000.

Chapter 9: Simulation and animation

A. Introduction

This chapter will present a program that implements a discrete-time simulation and animation of a ball being dropped and bouncing. Perhaps that sounds daunting, but it is relatively simple once you know how. The only new programming construct required for simulation and animation is threads. The only new concept is discrete time simulation, which may turn out to be more familiar than you expect.

B. An introduction to Threads

As discussed throughout, information processing is accomplished by sending messages to objects. A message invokes a method; first the parameter linkage is performed, then the method body is executed. When a method body is executed, first the first statement is executed, then the second, and so on, until the last statement finishes execution; then the method returns -- returns control to wherever the message that invoked it was sent.

Imagine the path of execution through a program over time. It includes the sequential execution of statements in method bodies and the transfer of control from one method to another by sending messages.

At any one time, one particular statement is being executed. If it is a message statement, it invokes a method, which executes until it reaches the end of its block statement. For example, in the Eye Applet, when the user pushes the shrinkButton, the run-time system invokes its `ActionPerformed()` method (see Code Example 9.1), which sends a

Code Example 9.1

```
1 private void shrinkButtonActionPerformed(java.awt.event.ActionEvent evt) {
2     leftEye.shrinkPupil();
3     rightEye.shrinkPupil();
4     repaint();
5 }
```

`shrinkButtonActionPerformed()` from EyeApplet.

`shrinkPupil()` message to first the leftEye, and then the rightEye, and then finally sends a `repaint()` message to itself (since `repaint()` is changed to `this.repaint()` by the compiler.

The `shrinkPupil()` method, in the Eye class (see Code Example 9.2) sends a `setRadius()`

Code Example 9.2

```
public void shrinkPupil() {
    pupil.setRadius(pupil.getRadius() - 2);
}
```

`shrinkPupil()` from the Eye class

message to the pupil object, which is a FilledCircle, but to do the parameter linkage, first it must evaluate the parameter, `pupil.getRadius() - 2`. To do that, it first sends the `getRadius()` message to the pupil object, and then subtracts 2 from whatever value that returns. Both `getRadius()` and `setRadius()`, although sent to a FilledCircle, end up in Circle (as FilledCircle inherits them from Circle).

So, the sequence of messages sent (and methods executed) is:

```
1.EyeApplet:actionPerformed()
2.        Eye:shrinkPupil()
3.                    Circle:getRadius()
4.                    Circle:setRadius()
5.        Eye:shrinkPupil()
6.                    Circle:getRadius()
7.                    Circle:setRadius()
8.        EyeApplet:repaint()
```

This sequence of statements is executed in order; each statement has control while it is executing. If you were to print the whole program, tape all the sheets together, and draw a line from the first statement executed to the second, on to the third, through the 8th; you would have a record of the path of execution when the shrinkButton is pushed. It would lead from one method to another across various classes, and would look a bit like a thread running through fabric. By following this path you can see which statements in the program are executing in which order. This imaginary line, this path through the program, is referred to as the ***thread of control,*** or thread, for short.

Each program you have seen thus far has had a single explicit thread, but to implement animation a second thread is needed. Fortunately Java has a built-in Thread class you can extend. Here's how.

i) Simplest threaded animation

The simplest threaded animation is an animated counter. It has two classes, an Applet and a Thread. The Applet creates and kicks off a Thread that repeatedly increments a counter (in the Applet) and has the Applet display its current value. To begin implementation of this example, create a GUI Applet named ThreadedApplet. Next, create a Controller class and copy the code from Code Example 9.3. The Controller class has a single instance

Code Example 9.3

```
public class Controller extends Thread {
   private ThreadedApplet theApplet;

    /** Creates a new instance of Controller */
   public Controller(ThreadedApplet theApplet) {
        this.theApplet = theApplet;
    }

   public void run() {
        for(;;) {
            step();
            pause();
        }
    }

    private void step() {
        theApplet.incCounter();
        theApplet.repaint();
    }

    private void pause() {
        try {
            Thread.sleep(50);
        } catch (Exception e) {}
    }
}
```

Threaded Controller class

variable named theApplet, which is used to both increment the counter in the Applet and to send the `repaint()` message to it (in the `step()` method).

a) Controller(ThreadedApplet)

The initializing constructor takes a single parameter, of type ThreadedApplet, and simply stores it in the theApplet instance variable. This is a standard technique to establish a two-way linkage between objects; the Applet object has a Controller variable which points to the Controller object and the Controller object has a ThreadedApplet variable which points back to the Applet. It is common to need a reference back to the object that created another object and that is how it's done.

b) run()

When the `start()` message is sent to the Controller from the Applet, because Controller extends Thread, it first ***spawns*** a new thread of control, and then sends the Controller in that Thread the `run()` message. This new Thread exists until the `run()` method returns. The run method here is an infinite loop (see "An infinite for loop" on page 205). It runs forever (or until the user closes the Applet), and it does just two things, `step()` and `pause()`, over and over.

c) step()

The `step()` method sends first `incCounter()` and then `repaint()` to the Applet. The pause method does only one thing, it sleeps for 50 milliseconds, but it looks rather complicated because the `sleep()` method, which is sent to the Thread class throws an exception, which must be caught (see ???). The simplest thing to do right now, is anytime you need a program to pause, simply copy and invoke this method. The `Thread:sleep()` method takes an int parameter and sleeps for that many milliseconds.

Add the bold lines in Code Example 9.4 to your Applet. That's all it takes! Run it, and if

Code Example 9.4

```java
1 public class ThreadedApplet extends java.applet.Applet {
2     private Controller theController;
3     private int counter;
4     public void incCounter() {counter++;}
5
6     /** Initializes the applet BallApplet */
7     public void init() {
8         initComponents();
9
10        theController = new Controller(this);
11        theController.start();
12    }
13
14    public void paint(java.awt.Graphics g) {
15        g.drawString(""+counter, 100,100);
16    }
```

Changes to the ThreadedApplet class

Line 2: A variable of type Controller; this is the new Thread.
Line 3: A counter; will start, by default as 0.
Line 4: A method to increment (add 1 to) the counter; the Controller will use this.
Line 10: Create and store the Controller. Notice `this` as a parameter; `this` is the Applet.
Line 11: Spawn a new Controller thread and send the `run()` message to it.
Lines 14-16: The `paint()` method; draws the counter value in the Applet.

you've made no mistakes you should see a counter counting... forever. Close the Applet to make it stop.

To make it seem more animated modify `paint()` as in Code Example 9.5. Or, if you'd

Code Example 9.5

```
  public void paint(java.awt.Graphics g) {
      g.drawString(""+counter, 100,100);
      drawOval(counter%200, counter%200, counter%177, counter%177);
 }
```

Addition to the ThreadedApplet:paint() to draw a circle

like the circle to stay the same size, use a constant (like 20) for the 3rd and 4th parameters. Do you see why this does what it does? Recall that the % operator gives the remainder after integer division (see "Arithmetic operators" on page 119), so the first two parameters range from 0-199. When counter is 200, or 400, or any even multiple of 200, `counter%200` is zero, that's why the circle goes back to the upper left periodically.

C. The programming task

Your task for this chapter is to implement and animate a user controlled simulation of a ball falling under the influence of gravity and bouncing until it stops. Let the user start and stop the simulation by pressing a button. Make the radius of the ball 20 pixels and its color red. Assume the elasticity of the collision with the floor is 90%; thus if the downward velocity of the ball were 100 when it hit the floor, its subsequent upward velocity would be 90.

i) Design

As always, before starting to write code, you should do enough design work to avoid wasting many frustrating hours going down blind alleys. Never start programming before you have a clear idea what you are trying to accomplish and how you will approach the problem. To design this program, in addition to sketching the GUI and deciding which classes to use, you must also have some idea what discrete time simulation is. These three will be taken up in the next three subsections.

a) GUI

The problem description says there must be a Button to start and stop the simulation; so, obviously you will need a Button (or two). It also says the ball should bounce when it hits the floor; thus drawing a line for the floor, or perhaps a box around the ball would make it less mysterious for the user when the ball changes direction.

How big should the Applet be? The bigger it is, the farther the ball will be able to fall before it hits the floor; but the particular size is not critical.

b) Discrete time simulation

Models are built to learn about systems of interest. A model allows you to practice with and/or experiment with a system in a safe and inexpensive manner. A simulation of a jetliner allows pilots to train without risking lives and expensive equipment. A simulation of an economy allows planners to try out different measures without disrupting the actual economy. A simulation of the interaction of greenhouse gasses, temperature and glaciation allows scientists to make predictions about the likely effect of various levels of greenhouse gas emissions. A model is always simpler than what it models (and so, no model is ever perfect). Simulation is the implementation of a model in software. Thus, the essence of simulation is simplification.

A discrete time simulation is named that because time moves in discrete steps. The size of the step is up to the modeler; a simulation of a computer might have a time step of a nanosecond, whereas one of continental drift might use a time step of a century or a millennium. In the real world, time is entirely beyond our understanding or control. In a simulation the modeler has complete control over time; it is whatever time the time variable says it is. Welcome to cyberspace!

A simulation has some set of simulated objects. The state of each simulated object is completely specified by its state variables. Similarly, the state of a simulation is completely specified by the values of all its state variables. To move from one time step to the next, the modeler provides a transition function. Given the values of all the state variables at one time step, it calculates their values at the next time step. This transition function may be simple or extremely complex depending on the application.

In the Snowman program, each time the user pushed the button the Snowman melted some and the puddle under it grew some. If you think of that as a simulation of a snowman melting, the state variables for a Snowman were x, y, size, and the size of the puddle. The transition function decreased the size by 10% and increased the size of the puddle by 10 pixels. The location of the snowmen never changed (unless you made them move!). There was no interaction between the snowmen.

In a more complicated simulation, for instance one implementing a model of planetary motion around the sun, the various elements of the simulation affect each other. For a famous old simulation of a cellular automaton, Conway's Life see ???.

A dropped ball bouncing straight up and down under the effect of gravity will need two state variables; one for its height from the floor and one for its velocity (up or down). Given a position and velocity and a gravitational constant, its position and velocity at the next time step can be computed using equations from elementary physics. Using standard physics notation; s stands for position, v for velocity, and a for acceleration. As you might guess v_t is the velocity at time t, $v_{t+1}$is the velocity at the next time step. Thus the transition function may be written as two equations:

$$s_{t+1} = s_t + v_t$$
$$v_{t+1} = v_t + a_t$$

So the ball's height at the next time step is just its current height plus its current velocity; its velocity at the next time step is just its current velocity plus the acceleration due to gravity.

If you have not studied physics, those equations may seem a bit mysterious; but they are quite simple once you understand them. If a ball is travelling at 10 pixels/step and is currently at location 100, after the next step it will be at 110. In symbols, $s_t = 100$, $v_t = 10$, $s_{t+1} = s_t + v_{t\,=}$ 100+10 = 110. Velocity is updated similarly.

c) Classes

The only thing in this program is the bouncing ball, so an obvious choice for a class is Ball. If Ball extends FilledCircle, the position, color and `paint()` code is already written. The only additional variable needed is velocity in the y direction. The only additional method needed is `step()`, which implement the transition function from one time step to the next.

ii) Implementation

Given your experience with the counter animation, you have seen code that does almost everything the code for this program must do. You only need to create three classes; the Applet, Controller and Ball (although you will need to copy FilledCircle and Circle from your previous project).

a) The Applet

The Applet needed for this program is almost identical with Code Example 9.4. Create a GUI Applet, name it BallApplet, and copy the code from Code Example 9.6. Or, since

Code Example 9.6

```
1 import java.awt.*;
2
3 public class BallApplet extends java.applet.Applet {
4     Controller theController;
5
6     /** Initializes the applet BallApplet */
7     public void init() {
8         initComponents();
9         theController = new Controller(this);
10         theController.start();
11     }
12
13    public void paint(Graphics g) {
14        g.drawRect(1,1,300,750);
15        theController.paint(g);
16    }
17 }
```

Initial BallApplet

As you can verify, this is identical with Code Example 9.4 with the counter removed, and the addition of line 14.

Line 14: Draw a rectangle for the Ball to bounce in. The height is 750 so that with an Applet height of 800 the bottom of the rectangle still shows.

you just typed most of this code in doing the previous program, simply copy it from there, delete the counter code and add the `drawRect()` message (line 14). Notice that this draws a rectangle 750 pixels high; you could chose another size if you wanted.

b) The Controller class

The Controller class is very much like the Controller in Code Example 9.5. Copying and pasting it, then editing it would be the most efficient, but feel free to retype it if that would help you remember it better. The code you need is in Code Example 9.7 with the

Code Example 9.7

```
1 import java.awt.*;
2 import java.applet.*;
3
4 public class Controller extends Thread {
5     private Applet theApplet;
6     private Ball theBall;
7
8     /** Creates a new instance of Controller */
9     public Controller(Applet theApplet) {
10        this.theApplet = theApplet;
11        theBall = new Ball(100,100,20,Color.RED);
12    }
13
14    public void run() {
15        for(;;) {
16            step();
17            pause();
18       }
19    }
20
21    public void paint(Graphics g) {
22        theBall.paint(g);
23    }
24
25    private void step() {
26        theBall.step();
27        theApplet.repaint();
28    }
29
30    private void pause() {
31        try {
32            Thread.sleep(50);
33        } catch (Exception e) {}
34    }
35}
```

Controller for the BallApplet

As you can see, this Controller is very similar to the one in Code Example 9.5 with the addition of the Ball and the `paint()` method.

Line 11: The Ball will initially be centered at (100,100).

changes from the previous Controller indicated in **bold**. As you can see, you must declare a Ball variable (line 6), instantiate it in the constructor (line 11) and send it the `step()` method when the Controller steps (line 26). Plus, you must `paint()` it when the Controller is painted. That's all. All that remains is writing the Ball class.

Be sure you are familiar with the Controller code. Read each method, line by line. Think about how they interact (i.e. which invokes which, when). After doing that, read the descriptions below. Pay special attention if there are any surprises -- surprises are clues for what to focus on.

- **Controller(Applet)**

The constructor stores the reference to the Applet in the variable named theApplet. Then it instantiates a red Ball of radius 20, centered at (100,100), and stores it in the variable named theBall.

- **run()**

An infinite for-loop with two statements. Each ***iteration*** it steps and then pauses (for 50 msecs).

- **paint(Graphics)**

Sends paint(Graphics) to theBall. The only thing in the simulation is the Ball, so that's all that needs to be painted.

- **step()**

Sends `step()` to theBall, then repaints the Applet so the Ball's new position will be displayed.

c) The Ball class

The Ball class extends FilledCircle. It needs a variable for the y velocity (say, vy), an initializing constructor, and a `step()` method. The constructor can just use FilledCircle's initializing constructor, i.e. it is one line: `super(x,y,r,c);`. The `step()` method has just two lines; one to update the position (`y = y + vy;`) and one to update the velocity (`vy = vy + GRAVITY;`). Recall that constants by convention are all uppercase (see "Case conventions" on page 128), and gravity is definitely constant!

Gravity is constant, but what value should it have? That decision is entirely arbitrary unless you assign some correspondence between pixels and distances in the world. That would be possible, but not necessary in this case. Let the acceleration due to gravity be one pixel per time step. By the way, the units of velocity is also pixels/time step.

Given that description, you can write the Ball class. If some part of that still seems mysterious feel free to look at Code Example 9.8. But, if you actually want to learn to

Code Example 9.8

```
1 import java.awt.*;
2
3 public class Ball extends FilledCircle {
4     public int GRAVITY=1;
5     private int vy;
6
7     public Ball(int x, int y, int r, Color c) {
8         super(x,y,r,c);
9    }
10
11    public void step() {
12        y += vy;
13        vy += GRAVITY;
14
15        System.out.println("v=" + vy);
16    }
17 }
```

The Ball class

Line 4: Define acceleration due to gravity as one pixel per time step.
Line 12: Update the y-coordinate.
Line 13: Update the y-velocity.
Line 15: Print the velocity so you can get some idea how it changes.

program, instead of copying that example, keystroke for keystroke, take the time/spend the effort to study and understand it; then go to the screen and type it in without peeking (if you have to peek once or twice, that’s okay). Or, just copy it and waste your time; your choice -- it doesn’t really matter, you’re going to live forever, right?

d) Testing

With those three classes written, there’s enough code to test. Make the Applet taller, so you can see the Ball hit the bottom of the rectangle (see "Changing the size of an Applet" on page 341).

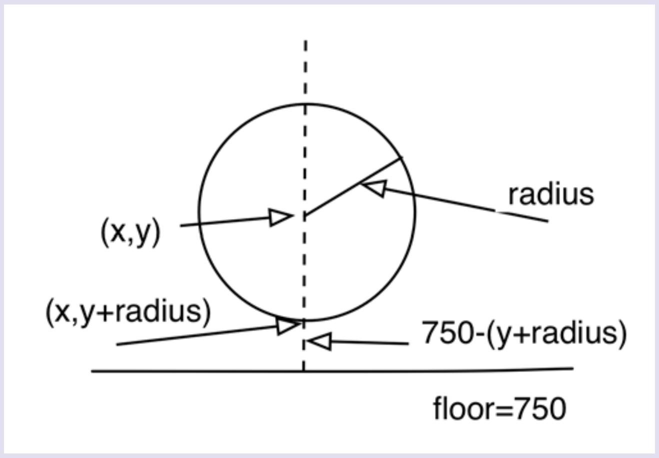
Once you fix all the syntax errors, you should see the Ball falling faster and faster and disappearing off the bottom of the screen. If you look at the output you will see that it is going one unit faster each time step. Now it’s time to add the bouncing off the floor code.

e) Making the Ball bounce - design

There are two steps to making the Ball bounce when it hits the floor: 1) detecting that it hits the floor, and 2) reversing its direction. The latter is simple, just set the y-velocity to its opposite (i.e. `vy = -vy;`). Note that the minus sign is unary minus, it only has one operand. The former is a bit more complicated.

Logically, when the ball hits the floor it should reverse direction. In the world, physics takes care of that; in the simulation, the programmer must decide how to simulate the physics. Here’s an idea. On each time step, after updating the position, if the Ball has hit the floor, reverse its direction. Looking at Figure 9.1 you can see that the distance from

Figure 9.1



the bottom of the Ball to the floor is 750-(y+radius), since y is the y-coordinate of the center of the Ball. Thus the Ball has hit the floor if 750-(y+radius)<0, or equivalently if (y+radius) > 750. In that case you wish to reverse its direction.

f) Making the Ball bounce - implementation

The `step()` method from Code Example 9.8 can be modified to reverse directions when the Ball hits the wall as shown in Code Example 9.9. Add that if statement and run the

Code Example 9.9

```
1     public void step() {
2        y += vy;
3        vy += GRAVITY;
4
5        if (y+radius > 750) // if it hit the wall
6            vy = -vy;       // reverse direction
7    }
```

The Ball:step() method with bounce code

Line 5-6: If the bottom of the Ball is through the floor, reverse the direction.

simulation again. If the rectangle you drew was some other height than 750, use that height instead of 750!

If your machine works like mine (and it might not, exactly), you saw the Ball bounce, but it goes part way into the floor and gradually bounces higher and higher. This is a bit surprising, and not at all like a real ball! What's gone wrong? There are actually two different, interacting causes. One of them stems from using ints which can only represent whole numbers; the other from a flaw in our simulation methodology.

First we will confront a shortcoming of discrete time simulations. Assume the Ball is travelling 10 pixels/step and before the time step it is 3 pixels from the floor. After the time step, if it moves 10 pixels down, it will be 7 pixels into the floor before we check. But, let's ignore that for now until we get the bouncing higher problem solved.

Consider carefully what happens when the `step()` method in Code Example 9.9 is executed with vy=17, and y=721. The first line changes y to 738. The second changes vy to 18. The if compares (738+30) to 750; since 758>750 it evaluates to true and so changes vy to -18. So, not only has the Ball gone 8 pixels into the floor, but also the velocity has *increased*! Perhaps you should only increase the velocity if the Ball does not change directions as in Code Example 9.10. Try out this version. If your machine works

Code Example 9.10

```
1    public void step() {
2        x += vx;
3        y += vy;
4
5        if (y+radius > 750) { // if it hit the wall
6            vy = -vy;       // reverse direction
7            System.out.println("vy=" + vy + " y=" + y);
8        }
9        else vy += GRAVITY;  // no bounce? accelerate
10   }
```

The Ball:step() only accelerate if no bounce

Line 5-9: If the bottom of the Ball is through the floor, reverse the direction otherwise add the influence of gravity to the velocity.

like mine, the ball always bounces the same height (although it continues to go into the floor a bit).

The problem description said the elasticity of the ball/floor collision was 90%, so when it changes direction you really should reduce the speed by 10% as in Code Example 9.11.

Code Example 9.11

```
1    public void step() {
2        x += vx;
3        y += vy;
4
5        if (y+radius > 750) { // if it hit the wall
6            vy = -vy*9/10;       // reverse direction and reduce 10%
7            System.out.println("vy=" + vy + " y=" + y);
8        }
9        else vy += GRAVITY;  // no bounce? accelerate
10   }
```

The Ball:step() 90% elasticity

Line 6: The collision between the ball and the floor has 90% elasticity, so reduce magnitude by 10% when changing direction.

Try this out. It comes to a stop for me, but into the floor by 8 pixels. Here is some code to fix it is shown in Code Example 9.12. This code is a bit complicated and if you don't

Code Example 9.12

```
1 public void step() {
2     x += vx;
3
4     int bottomY = 750-(radius+y);
5     if (vy >= bottomY) {
6         //System.out.println("vy=" + vy + " y=" + y);
7          int bounceHt = vy - bottomY;    // how high it bounces this step
8          y = 750 - (radius + bounceHt);  // y-coord after this step
9          vy = -(vy-1)*9/10;
10    }
11    else {
12        y += vy;
13        vy += GRAVITY;
14    }
15}
```

The Ball:step() improved?

Line 7: Calculate how far it will travel up after bouncing
Line 8: From that calculate the new y-coord of the center
Line 9: Reverse vy and subtract 1 to avoid endless small bounces.

understand it, that's okay. If you want to figure it out, draw a picture. This code causes the Ball to eventually come to rest exactly touching the floor. Without the -1 in vy-1 on line 9 it got caught in a loop and never stopped bouncing due to the way int arithmetic works. This is by no means a perfect simulation, but its close enough for now.

g) Starting and stopping with two Buttons

The last task remaining from the project description is to allow the user to start and stop the simulation by pressing a button. This will be done first with two Buttons, then with one. The former is easier to understand, the latter is less cluttered and illustrates a useful technique.

Assume the two Buttons are named stopButton (which sends a `stop()` message to the Controller) and goButton (which sends a `go()` message). The question is how to implement `stop()` and `go()`? Recall that the `Conroller:run()` method (shown in Code

Example 9.13) is running in a separate Thread. It is an infinite loop that does `step()`, and

Code Example 9.13

```
   public void run() {
       for(;;) {
           step();
           pause();
       }
   }
```

Controller:run() from Code Example 9.3

then `pause()`, over and over. How can you arrange that `stop()` will stop the simulation and `go()` will resume it? What is needed is if `stop()` has been executed since `go()`, the loop should just `pause()` and not `step()`; otherwise it should do both. Do you know what programming construct to use?

Whenever you want to either do something or not, you use an if statement. You would like to modify `run()` as shown in Code Example 9.14 so the `step()` message is only sent

Code Example 9.14

```
1    public void run() {
2        for(;;) {
3            if (last button pushed was go)
4                step();
5            pause();
6        }
7    }
```

An if statement to make step() conditional

Lines 3-4: step() will only happen if the last button pushed was the goButton

if the last button pushed was go. But how to write that in Java? The answer is to use a boolean variable, called perhaps, running. The type boolean is used when you only need to represent two values, true and false (see "types, values, operators" on page 118). This situation is perfect for a boolean variable, the simulation is either running or paused. When the Controller gets the `stop()` message, it should set running to false, when it gets the `go()` message it should set it to true, and the expression guarding the `step()` message should just be the boolean variable named running. This is illustrated in Code Example

9.15. Notice that the stop() method has been renamed to `userStop()`. This was because

Code Example 9.15

```
1     private boolean running=true;
2
3     public void go() {
4         running = true;
5     }
6
7     public void userStop() {
8         running = false;
9     }
10
11    public void run() {
12        for(;;) {
13            if (running)
14                step();
15            pause();
16       }
17    }
```

Modifications to make the run method stoppable

Line 1: Declares a boolean variable named running whose initial value is true (so the simulation will start when `run()` starts).

Lines 3-5: The `go()` method sets running to true.

Lines 7-9: the `userStop()` method sets running to false. Java would not allow a method named `stop()` in a Thread (since there already was one declared final).

Line 13: Guards the `step()` message; it is only sent if running is true.

when it was named stop() the compiler complained that:

```
Controller.java [29:1] stop() in Controller cannot override stop() in
java.lang.Thread; overridden method is final
```

What this means is that the Thread class already has a method with that signature (i.e. `public void stop()`), and it is declared `final`, so it *cannot* be overridden. You will never have to declare any methods final (in the context of introductory programming), so you may safely ignore this, but you do need to learn how to cope when you bump into this kind of error message. The rule is, when an error message says "...cannot override...", or "...access type...", then you have stumbled on a method declared in a superclass. The simplest solution is to rename your method.

That's all you need to know to make it start and stop. Do that now. Then return here to learn how to accomplish it with only one Button.

h) Starting and stopping with only one Button

Using one Button to stop the simulation and another to restart it works, but it is not very elegant. You can unclutter your GUI by writing code to make your Button a ***toggle*** switch. Light switches are sometimes push button toggles; if the light is on, pushing the switch turns it off, if it is off, pushing the switch turns it on. Here's how to do that in Java.

The Button must do one of two things; make the Controller stop if it is going, or go if it is stopped.. The construct in Java that does either one thing or the other, is if-else. So, logically the `actionPerformed()` method might be as in Code Example 9.16; if the

Code Example 9.16

```
private void toggleButtonActionPerformed(java.awt.event.ActionEvent evt) {
    if (the simulation is running)
        theController.userStop();
    else theController.go();

    change the label on the Button to say what it does now
}
```

Logic of actionPerformed() for a toggle Button

simulation is currently running, send the Controller the `userStop()` message, otherwise send `go()`.

One's first idea might be to add a boolean variable in the Applet to keep track of whether the Controller is currently running, and set it to the same value as the running variable in the Controller. This is a natural mistake. The problem is that having two different variable keeping track of the same information creates an unnecessary situation where bugs can occur. There will be plenty of bugs no matter what; not reason to encourage them!

The Controller already knows whether the simulation is running. The job of the Applet is to manage the GUI and pass along information to the Controller. When it needs to know if the Controller is running, it should ask it (and then set the toggleButton's label appropriately). The logic of changing the state of the Controller belongs in the Controller.

Thus, when the toggleButton is pushed, it should send a `toggle()` message to the Controller it change the label on the toggle Button to reflect what action will be performed if it is pushed. See Code Example 9.17 for how to do these two things. Notice

Code Example 9.17

```
1 private void toggleButtonActionPerformed(java.awt.event.ActionEvent evt) {
2     theController.toggleRunning();
3
4     if (theController.getRunning())
5         toggleButton.setLabel("stop");
6     else toggleButton.setLabel("go");
7 }
```

Making a toggle Button

Line 2: Toggle the running variable in the Controller.

Lines 4-6: An if-else. Notice that both the if and else parts have only one statements so they do not need to be enclosed in {}s.

Lines 5: Sets the Button label to "stop"; since theController is running.

Lines 6: Sets the Button label to "go".

that there are no {}s around the statements after the if and else parts. If you wanted to do two (or more) things in either the if or else parts you must put {}'s around them to make the two statements into one, syntactically.

The Controller class must be modified to add these two new methods, but you can eliminate two methods at the same time. See Code Example 9.18 for the changes.

Code Example 9.18

```
1     private boolean running=true;
2
3     public boolean getRunning() {return running;}
4
5     public void toggleRunning() {
6        running = !running;
7     }
8
9     public void run() {
10        for(;;) {
11            if (running)
12                step();
13            pause();
14       }
15    }
```

Controller modifications to support the toggle Button

Line 3: The accessor for running.

Lines 5-7: The `toggleRunning()` method replaces `go()` and `userStop()`.

Lines 6: Set the value of running to the opposite of what it was before. The ! operator is called, not. Not true is false, not false is true.

Compare to Code Example 9.15 on page 233, which it replaces.

D. Recapitulation

To do animation in Java you must spawn a separate Thread. To do this you must create a class that extends Thread, create an instance of it and send it the `start()` message. The `Thread:start()` method spawns the new Thread and then sends `run()` to your class; the new Thread exists until `run()` returns.

Discrete time simulation advances time by some fixed amount each step. Each modelled object has a set of state variables with completely determine its state in the context of the simulation. A transition function calculates the next state of each thing in the simulation from its previous state variables (and possible those of other things in the simulation).

Boolean variables are used to store information when the only possible values are true and false. Boolean expressions are used to determine whether to execute the if part of if statements.

E. Conclusion

This chapter introduced two powerful techniques, simulation and animation. It also introduced Java Threads. Simple animations can greatly improve your GUIs. Simple simulations can provide interesting demonstrations. It also presented a java toggle switch to allow the user to control the simulation.

F. End of chapter material

i) New terms in this chapter

iteration - another word for repetition. See the next chapter for iterative constructs. 219
spawned - technical term for created; used only for Threads. When a new thread of control is initiated, i.e. begins execution, it is said to be spawned. 212
thread of control - the sequence of statements executed when a program executes. A temporal map of where control resides during execution. 210
toggle - a two state switch which changes state each time you activate it 228

ii) Review questions

9.1 What are boolean variables used for?
9.2 Where do boolean expressions appear in Java?
9.3 What is Thread short for?
9.4 If your program has two Threads and is executing on a single CPU machine, since only one instruction can be executed at a time, how can both Threads be running at the same time?
9.5 What is the job of the transition function in a discrete time simulation.
9.6 What are the two unary operators?

iii) Programming exercises

9.7 Experiment with Buttons; `setLabel()` looks exactly like an accessor; try out `getLabel()`.
9.8 A Button is a Component (like an Applet, or a Frame). You saw `setBounds()` sent to a Frame... try sending it to a Button. What happens?

9.9 Modify your code so that the Ball can be throw to start the simulation. All that is needed is an additional initializing constructor that takes two additional parameters, the x-velocity and y-velocity.

```
public Ball(int x, int y, int r, Color c, int vx, int vy) {
    this(x,y,r,c);
    this.vx = vx;
    this.vy = vy;
}
```

Then you can initialize the ball in the Controller with:

```
theBall = new Ball(10,100,20,Color.RED, 2, -20);
```

For this to work, you must update the x-coordinate in Ball:step()

9.10 Add code to keep the Ball in the box. It's just like the bounce code except you must check for hitting the other sides of the box.

9.11 Add another Ball (or two) going in a different initial direction. Add 5!

9.12 Modify your code so that it create a pattern like on the back cover of this text. All this requires is adding an `update(Graphics g)` method to the Applet that simply sends `paint(g)`, as shown in Code Example 9.19. An explanation can be found in

Code Example 9.19

```
public void update(Graphics g) {
        paint(g);
}
```

BallApplet:update(Graphics)

Add this method to prevent `update()` from filling a rectangle the size of the drawable area in the background color before sending `paint()`.

"repaint(), paint() and update()" on page 351.

Chapter 10: Reading and writing files

A. Introduction

i) File I/O

Almost every sophisticated program reads and writes files. Toy programs can run without file input or output, but without ***`file I/O`***, your programs cannot permanently store information, or access information stored on disk. Until you can do file I/O, you don't really know how to program; no, it's more like you are crippled -- or perhaps you are like a pre-literate culture, unable to record information that can be recovered later.

Input and output (I/O) are always idiosyncratic. They are always a messy, fiddly, aspect of programming. Folklore has it that fully half the code in a major project is for the GUI. File I/O is simpler, but requires careful thinking and can be incredibly frustrating until it works. This chapter will introduce file I/O and a very helpful class for parsing input, java.util.StringTokenizer.

ii) The Model-View-Controller pattern

A common (and useful) technique in any complex program is to collect all the I/O into one place. This makes it easier to find and change; and if the program is ever transported to a new context, all the I/O changes can be made at once. This technique is also part of the Model-View-Controller pattern in Design Patterns (a ground breaking book that identified a number of patterns that occur over and over in software -- it is highly recommended if you intend to go on in computing). The View is what the user interacts with, the GUI in many cases. The Model is the program and its data; in a database application the database itself is the model. The Controller is the interface between the two.

iii) On ignorance, stupidity, and utilities

Language evolves constantly. People are born without language, and learn whatever language is spoken around them. They associate words they hear with concepts in their experience; and often make mistakes doing so. New words enter the language, new usages of old words appear; that's why this is not in Latin, or Italian -- the Roman culture spread and regional dialects turned into French, Spanish, even English.

In contemporary American usage, the words "ignorance" and "stupidity" seem have become conflated. When a first grader quickly answers, "What is 7 times 7?", with, "49!", someone might exclaim, "Aren't you smart!". Knowing facts about multiplication isn't an indication of intelligence, but rather of memory. Intelligence and memory are different; in the same way that algorithms and data are different; just as method bodies and values of variables are different. Confusing the two (process and content) only leads to more confused thinking (or calamity and disaster, depending on circumstances).

Intelligence is the ability to learn, and to be able to flexibly apply what you've learned in novel situations. Knowledge is the stuff you've learned. They are different. Completely. Everyone is born entirely ignorant. Ignorance can be cured; it is totally normal to not know something you have never studied. On the other hand, intelligence is genetic; you get what you get. Slugs (common in my garden) can never be frustrated, or bored, or learn to program, or write poetry, or fall in love; they just don't have enough brain cells for it. People do.

How people understand the world, or language, depends on what experiences they have had. No one knows everything, and everyone believes many things that are false. For instance, imagine that you played Monopoly as a child, and learned the word "utilities" in that context (Monopoly has two properties called "The Utilities", which are Waterworks, and The Electric Company) you might imagine that "utility" was just an abstract noun, like "animal". Years later, if you found yourself in a remote cabin without any utilities; and you chopped wood for the fire, carried water from the spring, you might discover that it took the whole day just to feed yourself. Then you might realize that utilities are really, really useful -- they *have* utility.

What's the point of that story? Java has a package,`java.util.*;` that has some very useful utilities. Several will be presented here. You have a choice, you can learn to use them, or you can spend the whole day chopping wood, tending the fire and carrying water! Oh, and also, don't feel bad about lacunae in your knowledge structure; everyone has them -- you can fill them in if you choose.

B. The programming task

Your task for this chapter is to write a Java program to copy one file to another, deleting all the words with less than 4 letters. The user should be able to select the input and output files. Put as many words on each line of the output file as fit without making any line longer than 80 characters. Ignore punctuation (i.e. treat punctuation as letters).

This task requires knowing how to do four things: 1) input from a file (of the user's choice), 2) output to a file, 3) break lines from a file into individual words, and, 4) determine the length of a String. The last is easy: `aString.length();` the other three will be addressed directly.

C. java.util.StringTokenizer

i) Isolating the I/O

You saw, in "Decoupling the output" on page 210, the technique of sending each String to be output to a method called `emit()`. This technique makes it simple to redirect the output; instead of finding and changing every `System.out.println`, you simply change `emit()`.

To similarly simplify input, one thinks of the input as a stream of tokens. A ***token*** is the smallest meaningful unit of input. What is meaningful depends, as always, on the context. It might be letters, or words, or numbers or something else. A tokenizer converts the input into a stream of tokens that are then used by the next component in the program.

There are a number of classes in java.util that are very useful (That's why they are called "utilities"!), including: Vector, Iterator, Hashtable, and StringTokenizer. The first three will be introduced in the next two chapters; the last, immediately. To break up a String into a series of smaller Strings (tokens), StringTokenizer is the class to use.

ii) Simplest example

The Sun documentation (http://java.sun.com/j2se/1.4.2/docs/api/) includes the following example:

The following is one example of the use of the tokenizer. The code:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

prints the following output:

```
this
is
a
test
```

This is mostly self-explanatory, assuming you are familiar with constructors, while loops, and realize what the StringTokenizer methods do. The constructor, `new StringTokenizer(String)`, creates a StringTokenizer that will separate the String it is passed into a number of tokens; it breaks the String at each space, so if there are no spaces it will only return a single token (the entire String). Each time you send the tokenizer `nextToken()`, it will return the next token, as long as there is one (if you send `nextToken()` to a StringTokenizer that is out of tokens, it will throw an Exception). To check if there are more tokens to be had, use `hasMoreTokens()`; it returns true just if the tokenizer has more tokens. These two methods are well named and once you understand what they do StringTokenizer is very easy to use.

iii) Sample exam question

Here is the kind of question you might expect on an exam in a CS1 class.

Write a method which is passed a String and prints each word in it on a separate line. Assume words are separated by spaces.

Could you answer such a question? If the answer is "yes", feel free to skip this next section.

a) Making progress without thinking very hard - or, Syntax is your friend!

When writing a method, there are a number of things which can be done even if you have no idea what the logic of the method will be. The method body contains the algorithm that method implements; it may require careful, creative thinking. The heading, by contrast, is mostly syntactic.

The question says it is passed a `String` and outputs to `System.out`, so you can write the heading with almost no thought, as shown in Code Example 10.1. On an exam, that

Code Example 10.1

```
void printOneWordPerLine(String input) {

}
```

Using syntax to your advantage

Knowing that the method is passed a String and returns nothing, you can write this much with minimal thought.

would get you lots more points than an empty space, and while you were writing it you could be thinking about how to implement it.

b) Writing the body of the method

Logically, the method will output a line for each word in the input, so there will be a `System.out.println()` in a loop with initialization at the top. The example from the Sun documentation, above, does almost exactly what is needed and can be copied almost verbatim, as in Code Example 10.2. This is a fine solution, but there are many other ways

Code Example 10.2

```java
void printOneWordPerLine(String input) {
    StringTokenizer st = new StringTokenizer(input);
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
}
```

The complete method

The only change needed from the Sun documentation example is to use the parameter being sent into the method as the parameter to the StringTokenizer constructor.

you could write this method. It could also be written as a for loop, as shown in Code Example 10.3. Make sure you understand how this for loop works, it’s a bit of a trick (see

Code Example 10.3

```java
void printOneWordPerLine(String input) {
    for (StringTokenizer st = new StringTokenizer(input);
            st.hasMoreTokens();
            System.out.println(st.nextToken())); // empty loop body
}
```

An equivalent (if peculiar) for loop

This does the same thing, but it is rather peculiar. Recall that a for loop heading has three parts. Here, the initialization instantiates the StringTokenizer; the continuation condition is exactly the same as in the while loop; and the update does the output. The loop body is an empty statement. See the next example for a more readable version.

"The for statement" on page 199 if you need to refresh your memory on the syntax of a for loop). Sometimes programmers do things like this, because they can. For a more ordinary usage, see Code Example 10.4.

Code Example 10.4

```
void printOneWordPerLine(String input) {
   for (StringTokenizer st = new StringTokenizer(input);st.hasMoreTokens(); )
        System.out.println(st.nextToken());
}
```

An equivalent, ordinary for loop

This does the same thing in a more usual form. Here, the initialization and the continuation condition are exactly the same; and the update does nothing. The loop body does the output, as you might expect.

iv) The other StringTokenizer constructors

a) Delimiters beside spaces

Sometimes you want to use other characters besides just spaces to delimit tokens. For instance if a user typed in:

```
John,Mary,Frank,Sue
```

and you used the default constructor, you would only get one token, namely the String:

```
"John,Mary,Frank,Sue"
```

when what you wanted was four tokens: `"John"`, `"Mary"`, `"Frank"`, and `"Sue"`. Fortunately StringTokenizer includes another constructor which will give you just that.

The line:

```
StringTokenizer st = new StringTokenizer("John,Mary,Frank,Sue");
```

is functionally equivalent to:

```
StringTokenizer st = new StringTokenizer("John,Mary,Frank,Sue", " ");
```

The second parameter is all the characters that you wish the StringTokenizer to use as delimiters. It very likely that the one argument constructor is implemented as shown in

Code Example 10.5. This is a standard technique to make coding and maintenance simple

Code Example 10.5

```
public StringTokenizer(String input) {
    this(input, " ");
}
```

Implementation of StringTokenizer(String)

The one argument constructor very likely is implemented by invoking the two argument constructor, using just a space for the second argument.

(see "this()" on page 116).

So, if you say:
`StringTokenizer st = new StringTokenizer("John,Mary,Frank,Sue", ",");`
the tokenizer will return the tokens `"John", "Mary", "Frank", and "Sue".` Unfortunately, if the input were `"John Mary Frank Sue",` the tokenizer will return just one token `"John Mary Frank Sue".` This can be solved by using `" ,"` as the second parameter:
`StringTokenizer st = new StringTokenizer("John,Mary,Frank,Sue", " ,");`
That would work with either input.

b) Delimiters as tokens

Using either of those constructors, you never see the delimiters; but, sometimes you want to. For instance, say you were reading input and wanted to replace certain abbreviations with what they stood for and preserve the punctuation. Then you would need to know what the delimiters had been (so you could output them). The third constructor does this with a third parameter, a boolean; if it is true, it will return the delimiters as tokens. So:
`StringTokenizer st = new StringTokenizer("A,B, C,DEF.", " .,", true);`

would return the sequence of tokens: `{"A", ",", "B", " ", "C", ",", "DEF", "."}`. Chances are the two parameter constructor is implemented as shown in Code Example 10.6.

Code Example 10.6

```java
public StringTokenizer(String input, String delimiters) {
    this(input, delimiters, false);
}
```

Implementation of StringTokenizer(String, boolean)

The two argument constructor very likely is implemented by invoking the three argument constructor, using false for the third argument.

StringTokenizer is a very handy class, but it is quite simple. For complicated tokenizing you should use a StreamTokenizer, but not today!

D. The MyReader class

i) Why MyReader?

There are many classes in Java that do I/O. Too many for beginners. It is a remarkable class structure that was designed to handle I/O from just about anywhere: the web, the keyboard, or files (compressed or uncompressed). It is worthy of the time to learn; but not here. For now you just want to input files one line at a time, with a minimum of effort. Here is a class called MyReader to read files easily without having to catch exceptions (here is a link to the Sun Tutorial on exceptions during compilation of file input code:
http://java.sun.com/docs/books/tutorial/essential/exceptions/firstencounter.html
By using MyReader, you can avoid that). MyReader can read either from wherever the user specifies on the local machine (if it runs from an Application), or from the directory an Applet loaded from.

ii) Echoing a user specified file

The first thing to do when you are trying to process a file, is to make sure you can read it. The easiest way to do that is to read it in and display it on the screen; this is referred to as echoing the file. Code Example 10.7 shows how to ***echo a file*** with a MyReader.

Code Example 10.7

```
MyReader mr = new MyReader();

while (mr.hasMoreData()) {
    System.out.println(mr.giveMeTheNextLine());
}

mr.close();
```

Echoing a file with MyReader

Reads each line from whatever file the user specifies, and echoes each to System.out.

That's all there is to it. You might want to adjourn to the keyboard and try that out, to convince yourself that it works and that you understand how to use it. Don't forget that you must be running an Application to read from files (see "Creating a GUI Application" on page 334). Now to the internals of MyReader.

iii) MyReader internals

The MyReader class is essentially a wrapper for the BufferedReader class, along with a method that uses a FileDialog to allow a user to select a file at runtime. A ***wrapper*** is a class that exists to hold a class or a data item without adding much functionality. Wrappers are written to make working with a class or data simpler for the user and/or the programmer.

a) MyReader: constructors, imports, and wrapped variable

Code Example 10.8 is the beginning of MyReader. Notice that it has just one variable, a

Code Example 10.8

```java
import java.io.*;
import java.awt.*;
import java.net.*;
import java.applet.*;

public class MyReader {

    private BufferedReader br;

    public MyReader() {
        openIt(getFileName());
    }

    public MyReader(String filename) {
        openIt(filename);
    }

    public MyReader(String filename, Applet theApplet) {
        try {
            URL theURL = new URL(theApplet.getDocumentBase(), filename);
            InputStreamReader isr = new InputStreamReader(theURL.openStream());
            br = new BufferedReader(isr);
        } catch (Exception e) {System.out.println("MyReader -- bad file
      from net" + e);}
    }
```

Beginning of MyReader

There is just one variable, the BufferedReader that MyReader is the wrapper for. The imports are needed for: BufferedReader, FileDialog, URL, and Applet, respectively.

BufferedReader (a class in java.io), called br (for the first letters of **B**uffered and **R**eader). It is declared private for two reasons: to make it clear that it is only used from within this class, and to prevent any other code from modifying or accessing it. A MyReader is essentially a more convenient form of a BufferedReader. There are three constructors. The default constructor opens whatever file the user wants (see "Opening the file" on page 249). If you pass a filename as a String, it will open that. From an Applet, pass a filename and `this`, and it will read from the directory the Applet bytecode is in. Recall that Applets can *only* read from there.

b) Opening the file

Code Example 10.9 shows the two methods that are used by the constructors to open the file. The `getFileName()` method is very useful if you ever want to prompt a user for a file

Code Example 10.9

```
25    void openIt (String filename) {
26        try {
27            br = new BufferedReader(new FileReader(filename));
28        } catch (Exception e) {
29             System.out.println("MyReader -- can't open " + filename + "!" + e);
30        }
31    }
32
33    String getFileName() {
34        FileDialog fd = new FileDialog(new Frame(), "Select Input File");
35        fd.setFile("input");
36        fd.show();
37        return fd.getDirectory()+fd.getFile();  // return the complete path
38    }
39
```

MyReader:openIt(String) and getFileName()

`openIt(String)` has the magic that creates a BufferedReader to read a file one line at a time. `getFileName()` uses a FileDialog to get a filename from the user at runtime.

to read; feel free to copy it. It is amazing the first time you use a FileDialog; it was so easy! If you omit line 36, the FileDialog will be invisible. Line 36 is a bit unusual in that it does not finish execution until the user closes the FileDialog. The Dialog box the FileDialog opens is modal, so like many Panels it seizes control of the Thread and refuses to relinquish it (until closed). Notice there are two parts to a complete path-name (line 37); the directory and the filename.

The `openIt(String)` method has the incantation to open a file for reading. Notice the nested constructor on line 27 (this line does all the work of this method). This is a good example of how constructors are used in object programming (see "Constructors" on page 111).

c) Reading and closing the file

Code Example 10.10 has the three methods that show MyReader to be a wrapper. Notice

Code Example 10.10

```
40    public String giveMeTheNextLine() {
41        try {
42            return br.readLine();
43        } catch (Exception e) {System.out.println("MyReader -- eof?!" + e);}
44        return "";
45    }
46
47    public boolean hasMoreData() {
48        try {
49            return br.ready();
50        } catch (Exception e) {System.out.println("MyReader -- disaster!" + e);}
51        return false;
52    }
53
54    public void close() {
55        try {
56            br.close();
57        } catch (Exception e) {System.out.println("MyReader: can't close!" + e);}
58    }
59 } // MyReader class
```

MyReader:giveMeTheNextLine(), hasMoreData() and close()

These methods just send `readline()`, `ready()`, and `close()` to the Buffered Reader. They bracket them with try-catch blocks, as required. This way, the programmer does not have to use try-catch in the code that uses the MyReader

Line 44: Is required, since if the `readLine()` throws an exception, *something* needs to be returned.

Line 51: Same thing.

that inside the try/catch block, `close()` just sends `close()` to the BufferedReader; `hasMoreData()` simply returns `br.ready()`; and `giveMeTheNextLine()` returns `br.readLine()`. The advantage of using MyReader is that you don't have to worry about the try-catch blocks around those three methods (which are annoying). Notice also that MyReader has `hasMoreData()` and `giveMeTheNextLine()` whereas StringTokenizer has `hasNext()` and `nextToken()`. Perhaps it would be easier to remember the method names in MyReader if they were more like those in StringTokenizer. You could just change the names in MyReader, but if you had old code that used the old names, it would break (not that you couldn't change all the names, it's just time consuming). Another solution is to write wrappers for the methods with new names, as shown in Code Example 10.11.

Code Example 10.11

```
public String nextLine() {
    return giveMeTheNextLine();
}

public boolean hasNext() {
    return hasMoreData();
}
```

MyReader:nextLine(), and hasNext()

These methods don't add any functionality; they just wrap other methods with different names. You might do this if you wanted to use the shorter, more familiar names.

iv) Emitting the tokens one per line

Now that you understand StringTokenizer and MyReader, it should be simple to modify the file echo code (Code Example 10.7 on page 247) to output one word on each line. Do that now. You can find the MyReader class at:
http://www.willamette.edu/~levenick/SimplyJava/io/
After you solve the problem, then look at Code Example 10.12 to see how the author did it. No fair peeking! It really does help you to learn if you actually type in, compile and execute little sample programs. At least if you're like most people; if you just read code, it doesn't stick -- but, if you run it and modify it and wrestle with it a bit, you can

remember it later. Okay, here's the code to solve that little problem. It turns out, that with the `printOneWordPerLine()` method to be, well, trivial!

Code Example 10.12

```
        MyReader mr = new MyReader();

        while (mr.hasMoreData()) {
            printOneWordPerLine(mr.giveMeTheNextLine());
        }

        mr.close();
```

Printing the words in a file one on each line

The `printOneWordPerLine()` method from Code Example 10.2 on page 243 does just what we need. Ain't software reuse great?

E. Writing to a file

It is simple to write to a file if you use the MyWriter class (it is simple enough without it as well). Code Example 10.13 shows how to write two lines into whatever file the user selects. The MyWriter class (which is in the directory with MyReader) has only three

Code Example 10.13

```
        mw = new MyWriter();

        mw.println("Hello...");
        mw.println("Here's the second line of a file");

        mw.close();

```

Writing to a file using a MyWriter

A tiny test of a MyWriter.

methods, `println(String)`, `print(String)`, and `close()`. It is important to close files after you finish working with them; otherwise sometimes the data in them cannot be read correctly (although sometimes it makes no difference).

The MyWriter class is shown in Code Example 10.14. Like MyReader there are

Code Example 10.14

```
import java.awt.*;
public class MyWriter {
    protected PrintWriter pw;

    public MyWriter() {
        openIt(getFileName());    }

    public MyWriter(String filename) {
       openIt(filename);    }

    void openIt (String filename) {
        try {
            pw = new PrintWriter(new FileWriter(filename));
        } catch (Exception e) {System.out.println("Can't open " + filename + "!" + e);}
    }

    public void print(String s) {
        pw.print(s);    }

    public void println(String s) {
        print(s+"\n");    }

    public void close() {
        pw.close();    }

    private String getFileName() {
       FileDialog fd = new FileDialog(new Frame(), "Output File", FileDialog.SAVE);
        fd.setFile("output");
        fd.show();
       return fd.getDirectory()+fd.getFile();  // return the complete path
     }
}
```

MyWriter

You probably don't need to know how this works, but making sure you do will be good review of the various Java constructs.

constructors with and without a filename specified as a parameter. The `openIt()` method has the magic nested constructor to open a file for writing. The `print()` and `println()` methods do what they do in `System.out`, which, by the way, is a PrintStream. Notice on line 27, that there is a third parameter in the FileDialog constructor; this allows you to enter a filename that does not yet exist.

F. Putting it all together

You have all the pieces to accomplish the task. You must still decide where to filter out the short words. There are several place you might do this:

1. As soon as you get the tokens from the StringTokenizer
2. In `emit()` (assuming you are using `emit()` -- you can leverage the code from Code Example 8.22 on page 212 with one minor alteration).
3. In a filter that sits in front of emit().

Here's how that last choice might be implemented. In the loop where you are reading tokens from the input file and writing them to the output file, instead of sending `emit(nextToken)`, you might send `emitIf(nextToken)` where `emitIf()` is as in Code Example 10.15. This has two advantages: 1) The `emit()` method may be left untouched.

Code Example 10.15

```
public void emitIf(String s) {
    if (s.length() > 4)
        emit(s)
}
```

emitIf(String)

By encapsulating the print criteria in a method, it is easy to find and change later, plus there's no need to alter `emit(String)`.

That way its logic does not need to be cluttered up with deciding whether to emit the word or not, it just does what its name says, emits. 2) If you want to implement some other filtering scheme (like only printing words that begin with 'x', or whatever), you can make the changes in `emitIf()` and not have to look anywhere else.

Still, maybe it would be easier to put that if statement in the method where you are getting the tokens. Maybe. It could be argued though, that writing an `emitIf()` method is cleaner and easier to understand and modify later. Plus, mixing the filtering criteria with the input violates the principle of having I/O code do nothing but I/O.

G. Conclusion

This chapter introduced file I/O, the StringTokenizer class, and two convenience classes, MyReader and MyWriter. They simplify reading and writing files, by: 1) wrapping up the Exception laden BufferedReader and PrintWriter classes from the java.io package, and 2) utilizing a FileDialog to prompt the user for input and output files.

H. End of chapter material

i) New terms in this chapter

echo a file - to read a file and display it on the screen 243
file I/O - file input and output 235

ii) Review questions

10.1 What is a token?
10.2 What are the two StringTokenizer methods?
10.3 What are the three StringTokenizer constructors?
10.4 What is a wrapper?
10.5 When might you wrap a method?
10.6 What is a FileDialog used for?

iii) Programming exercises

10.7 Write a small test Application to experiment with a FileDialog. Print the Strings that come back from `getDirectory()` and `getFile()`.
10.8 Comment out the imports in MyReader, one at a time and see what compiler error you get.
10.9 Write a fourth constructor of MyReader, namely `MyReader(Applet)`, that prompts the user and opens whatever file they choose. Hint: the body is a single line of code.
10.10 Imagine you are using the code in Code Example 10.12 on page 252, but something is going wrong with the input. Modify that code to echo each line input (with descriptive text, like: "And the next line is : ==>   <== tokens follow:".
10.11 Maybe you're tired of typing System.out.println all the time, and would rather just type out.println. You can do this by declaring an instance variable of type java.io.PrintStream called out. Do that in a class and test it out.
10.12 Maybe you're tired of typing `System.out.println("whatever")` all the time, and would rather just type `emit("whatever")`. You can do this by declaring an `emit(String)` method. Do that in a class and test it out.

10.13 Maybe you're tired of typing `System.out.println("whatever")` all the time, and don't want to have to declare `emit(String)` in every class, but would rather just declare it once. Create a class, called, say, MyUtils and make `emit()` a class method in it (see "Class methods" on page 167 if you've forgotten how). Then you can just say `MyUtils.emit("whatever")` in any class. Do that.

10.14 Write an Application that creates 10 files named junk0, junk1, junk2,...,junk9. The file junk0 should have just one line with a "0" on it; junk1 should have the lines

```
0
1
```

and junk9 should have the lines

```
0
1
2
3
4
5
6
7
8
9
```

Don't forget to close those files! Close your program and open those files in the NetBeans editor. Note: you will need to mount the directory they are in first. Hint: you must specify a directory (see `getFilename()` in MyWriter. One way to learn the directory you are reading/writing from would be to modify the `getFilename()` code to send the directory out to System.out when you select a file in the current directory. Then you could copy that back into the program.

The following refer to the file copy program.

10.15 Modify the file copy program to ignore punctuation in determining whether a word is long enough. I.e. as the program stands it would copy "this", because, with the quotes it has length 6.

10.16 Add a GUI control so if the user chooses the program will remember that last file read and read from it every time the user pushes the Read button. Hint: you will need to store the file name/path and use the `MyReader(String)` constructor if they want to reread.

10.17 Add a Choice to allow the user to select the minimum word length to copy. I.e. allow them to output only words with 5 letters or more, or 8.

Chapter 11: Data structures

A. Introduction

Up until now, we have been declaring variables one at a time. But, sometimes you want 10, or 1000 variables. If you were implementing software for a real bank, it might have thousand of accounts. If you wanted to program an army of 100 snowpeople, it would he hopeless to write 100 statements for each action you wanted them to perform. A big advantage of computing is that the machine doesn't mind doing the same thing over and over thousands of times. Data structures allow you to declare and store as many variables as you need with a minimum of effort. This chapter will show you how to use two similar data structures, array and Vector.

B. Arrays

Arrays are part of most programming languages. An array is a list of variables, all with the same type and name, but distinguished by an index. The declaration:

```
        int [] anArray = new int[100];
```

declares 100 variables of type int, all named anArray[something], where something is an int between 0 and 99. Thus, the first int variable is named anArray[0], the next, anArray[1], and the hundredth, anArray[99]. Each one acts exactly like an int variable, because each one *is* an int variable.The value in the square brackets is called teh index. It may be a constant, but usually it is a variable.

i) Simplest examples

Code Example 11.1 declares an array of five ints and then displays them to `System.out`.

Code Example 11.1

```
1  int [] list = new int[5];
2
3  for (int index=0; index<5; index++) {
4      System.out.println("index=" + index + " list[index]=" + list[index]);
5  }

        index=0 list[index]=0
        index=1 list[index]=0
        index=2 list[index]=0
        index=3 list[index]=0
        index=4 list[index]=0
```

Declaring and printing an array of five ints: code and output

Line 1: declares an array, called list, of five ints.
Line 2: a for loop to iterate over the five elements of the list
Line 4: inside the loop, since index will take on the values 0-4, in order, list[index] will be the five different int variables (in order, each time around the loop).

The loop on lines 3-5 is a standard method of accessing the elements of an array one at a time; it is said to "iterate over the elements of the array". As you can see (by the output), they are autoinitialized to zero when the array is declared.

If you want values besides zero in the array element, you must put them there, as shown in Code Example 11.2.

Code Example 11.2

```
1  int [] list = new int[5];
2  list[0] = 7;
3  list[3] = 33;
4
5  for (int index=0; index<5; index++) {
6      System.out.println("index=" + index + " list[index]=" + list[index]);
7  }
```

```
index=0 list[index]=7
index=1 list[index]=0
index=2 list[index]=0
index=3 list[index]=33
index=4 list[index]=0
```

The same array with list[0] set to 7 and list[3] to 33: code and output

Line 2: assign the value 7 to the first element of the array

Line 3: ...and 33 to the fourth

Code Example 11.3 illustrates setting the values of the array in a loop. It uses the current

Code Example 11.3

```
1  int [] list = new int[5];
2  for (int i=0; i<5; i++) {
3      list[i] = i*i;
4  }
5
6  for (int i=0; i<5; i++) {
7      System.out.println("i=" + i + " list[i]=" + list[i]);
8  }

       i=0 list[i]=0
       i=1 list[i]=1
       i=2 list[i]=4
       i=3 list[i]=9
       i=4 list[i]=16
```

The same array with each element set to the square of its index

Line 3: assign each element the value of the square of its index

Notice that "i" stands for "index".

value of the index squared as the value stored in each element.

ii) Printing a String backwards

Arrays may be of any type, primitive, built-in or user defined. An array of chars could be used to print a string backwards as shown in Code Example 11.4. If `length()` and

Code Example 11.4

```
1       char [] letters = new char[100];
2       String s = "pals";
3       for (int i=0; i<s.length(); i++) {
4           letters[i] = s.charAt(i);
5       }
6
7       System.out.println("frontwards, it's: ");
8       for (int i=0; i<s.length(); i++) {
9           System.out.println("i=" + i + " letters[i]=" + letters[i]);
10      }
11
12      System.out.println("backwards, that's: ");
13      for (int i=s.length()-1; i>=0; i--) {
14          System.out.println("i=" + i + " letters[i]=" + letters[i]);
15      }
```

```
frontwards, it's:
i=0 letters[i]=p
i=1 letters[i]=a
i=2 letters[i]=l
i=3 letters[i]=s
backwards, that's:
i=3 letters[i]=s
i=2 letters[i]=l
i=1 letters[i]=a
i=0 letters[i]=p
```

Printing a String forwards and backwards, one char per line
Lines 3-5: assign each char in the String to an element of the array
Lines 7-10: print them frontwards
Lines 12-15: and backwards

`charAt()` seem unfamiliar you might look back at "A few String methods" on page 207.

To print the String frontwards and backwards, all on the same line, just change the printlns to prints and remove some of the text, as in Code Example 11.5.

Code Example 11.5

```
1        char [] letters = new char[100];
2        String s = "pals";
3        for (int i=0; i<s.length(); i++) {
4            letters[i] = s.charAt(i);
5        }
6
7        System.out.print("the word ");
8        for (int i=0; i<s.length(); i++) {
9            System.out.print(letters[i]);
10       }
11
12       System.out.print(" backwards, is ");
13       for (int i=s.length()-1; i>=0; i--) {
14           System.out.print(letters[i]);
15       }

      the word pals backwards, is slap
```

Printing a String forwards and backwards, all on one line

Lines 3-5: assign each char in the String to an element of the array
Lines 7-10: print them frontwards
Lines 12-15: and backwards

iii) An array of Accounts

If you were writing a bank simulation with 1000 Accounts, you could declare an array like this:

```
        Account[] accountList = new Account[1000];
```

This will give you 1000 Account variables, each initialized to zero, which, when the variable is a reference (as any Object variable is), is interpreted as null. Thus, if you wish to have 1000 Accounts in those 1000 Account variables, you must do the second step of instantiating them all; like this:

```
        for (int i=0; i<1000; i++)
            accountList[i] = new Account();
```

If you forget to do this (and, *everyone* does when they start working with arrays of Objects), you will be confronted with Null Pointer Exceptions the first time you send a message to one of them; and if you're not paying attention, it could be very confusing. *Don't forget*!

Although this could work, it is better to use Vectors for lists of Objects.

C. Vector and Iterator

Java provides the Vector class to keep track of lists of objects. A Vector stores a list of variables of type Object. Thus it can store any type of object, since very object is an instance of some class and every class extends Object (directly or indirectly). This is very convenient, but has a downside. When you get objects back out of the list, the complier considers them to be of type Object. So, it will only allow messages that are defined in Object to be sent to them. To pacify it you must cast the Object to whatever type it actually is, as will be shown directly.

Vector has a number of useful methods, but to start we will focus on just two: `add(Object)` and `iterator()`. That will be enough to demonstrate that we can add things to the list and access them in order.

i) add(Object)

To add an Object, any Object at all, to a Vector, use `add(Object)`; i.e. simply send the list the `add()` message with that object as a parameter. The object will be added to the end of the list. For example, to make a list containing three Accounts, you could say:

```
java.util.Vector theList = new java.util.Vector();
theList.add(new Account("xena", 1234567));
theList.add(new Account("abe", 100));
theList.add(new Account("bea", 10000000));
```

Then first Account would be xena's, the last bea's.

ii) iterator()

Iterators have only two methods, `hasNext()` and `next()`. The former tells if there are any more items, the latter hands back the next one; it works just like StringTokenizer (odds are StringTokenizer is a wrapper for an iterator initialized by its constructor).

To access each item in a Vector, from first to last in order, use an Iterator, like this.

```
for (Iterator theIterator=theList.iterator(); theIterator.hasNext();) {
     Account nextAccount = (Account) theIterator.next();
     System.out.println("\n\nnext account..." + nextAccount);
4         }
```

The form of this loop is always the same; it is an idiom. The initialization declares a variable, theIterator, of type Iterator and initializes it to all the items in theList, by sending theList the `iterator()` message and storing what it returns (the for loop is described at "The for statement" on page 199). The loop continues as long as the Iterator has any more Objects.

Line 2 above:

```
            Account nextAccount = (Account) theIterator.next();
```

illustrates a generic technique used when iterating over a set of Objects. Each time around the loop, it gets the next Object from the Iterator, casts it as an Account and stores it in an Account variable named nextAccount. Once there, you can send it any message Account defines. If the Object returned from the Iterator is not an Account, it will throw a ***`Class Cast Exception`***.

In English, this loop iterates over the Vector called theList; each time around the loop it stores the next Account from the list in the nextAccount variable, and then (implicitly) sends it `toString();` whatever `toString()` returns is the parameter to System.out.

iii) Simplest test program

As usual, to convince yourself that a programming technique works and, more importantly, to become familiar with it before trying to use it for anything difficult, you should write a tiny test program. There are many ways one might do this, but here it is done with an Application, as shown in Code Example 11.6. Adjourn to the keyboard,

Code Example 11.6

```
1 import java.util.*;
2
3 public class VectorTest {
4
5     Vector theList;
6
7     /** Creates a new instance of VectorTest */
8     public VectorTest() {
9         theList = new Vector();
10        theList.add(new Account("xena", 12345));
11        theList.add(new Account("abe", 100));
12        theList.add(new Account("bea", 10000000));
13
14        for (Iterator it=theList.iterator(); it.hasNext();) {
15            Account nextAccount = (Account) it.next();
16            System.out.println("\n\nnext account..." + nextAccount);
17        }
18    }
19
20    /**
21     * @param args the command line arguments
22     */
23    public static void main(String[] args) {
24        new VectorTest();
25    }
26}
```

Simplest use of a Vector

Lines 9-16: Add three Accounts to a Vector and print them.

Line 24: Create a VectorTest which will invoke the default constructor and so run the test code in lines 9-16.

input and run this program. It will require that there is an Account class in that directory (you could use the Classmaker to generate one, or look around and find the one you used before).

That's all you need to know to use a Vector. The next example will use a Vector as the database for a bank simulation.

D. A simple bank database

According to *dictionary.com*, database means, "A collection of data arranged for ease and speed of search and retrieval.". Thus a database management system, or DBMS, is software that manages a collection of data. Somehow that's not as impressive sounding as "database management system". So it goes.

i) The database

In the simple Bank program in Chapter 3, the database consisted of three Account variables. There were always exactly three Accounts and the only thing a user could do was select the current account and withdraw money from it. It was hardly a database at all. The bank database here will have a variable number of Accounts. The bank administrator will be able to add, delete, or edit accounts, and then save the changes to disk. The data structure used will be a list of all the Accounts in the Bank. A Vector is suitable to implement this list.

ii) Inputting the database: load

Assuming there were hundreds or thousands of accounts in a bank, they should not have to be input by hand each time you start the program. Even for a small test bank DBMS, you would not want to type in all the data every time you run the program. Instead, account information should be stored on disk, in files. Program initialization would include inputting the database. An obvious place to input the database would be in the Bank constructor.

a) File format

The code to read the data from the file will expect it in a particular order and format. There are many ways to write files, but if they are human readable, then they are easy to maintain (since you can simply edit them!). Thus, MyReader and MyWriter from Chapter 10 are well suited for this job.

It doesn't really matter what format the data is stored in, but you*must* decide what that format will be. For simplicity, let's store the data for each Account on one line; first the name of the person, then the balance, with spaces in between. So, if there were 4 Accounts, the file might look like:

```
Amy 17
Zoe 9898
Joe  98
Bea    1000000
```

If there were more data fields, like an account number, or social security number, address, phone number, ATM number and password; they could be appended. Two fields is enough for illustrative purposes.

b) Encapsulation! Input in the Account constructor

The code to open the file and build the database belongs in the Bank class (since that Bank will be working with the database). Conceptually, it will look something like Code Example 11.7. Your first idea might be to write the inside of the loop as shown in Code

Code Example 11.7

```
1       while (more data in the input file) {
2           read the data for the next account
3           create and store the new account
4       }
```

Pseudocode for reading and building the database

Example 11.8. Code Example 11.8 is the inside of the loop from Code Example 11.7 with

Code Example 11.8

```
// read the data for the next account
StringTokenizer st = new StringTokenizer(mr.giveMeTheNextLine());
String name = st.nextToken();
int balance = Integer.parseInt(st.nextToken());

// create and store the new account
theList.add(new Account(name, balance);
```

First idea for inputting and creating the accounts in that loop.
This is the inside of the loop in Code Example 11.7 (the details of lines 2 and 3). Like many first ideas, this one has some problems.

the pseudocode mde into comments and the actual code written beneath it. The use of pseudocode as comments for the actual code is good form; you can type the pseudocode right into the class and then comment it out as you implement it. That way the compiler will remind you if you haven't implemented everything (since the pseudocode will generate compiler errors) and the comments will remind you what you were thinking when you wrote it. Logically, Code Example 11.8 is impeccable (assuming mr is a

MyReader and has been properly initialized). But, it violates the principal of encapsulation. What if later, more fields are added to the Account class? Then it would be necessary to edit the Bank class as well. It would decouple Bank and Account better, and be simpler for the programmer adding the fields to Account, if all the changes could be made in Account.

The standard technique to accomplish this is to write a constructor that is passed the input stream, and that reads the data it needs from that, as shown in Code Example 11.9. To

Code Example 11.9

```
Account(MyReader mr){   //empty default constructor
    StringTokenizer st = new StringTokenizer(mr.giveMeTheNextLine());
    name = st.nextToken();
    balance = Integer.parseInt(st.nextToken());
}
```

Account(MyReader) constructor

Reads and stores information for one Account from the parameter. By doing the I/O for Account *in* Account, encapsulation is increased and the programmer’s job is simplified.

test this code (stepwise implementation!) use the Bank class in Code Example 11.10.

Code Example 11.10

```
1  import java.util.*;
2
3  public class Bank {
4     private Vector accountList;
5
6     /** Creates a new instance of Bank */
7     public Bank() {
8         accountList = new Vector();
9         inputAccounts();
10    }
11
12    private void inputAccounts() {
13        MyReader mr = new MyReader();
14        while (mr.hasMoreData())   // read and store database
15            accountList.add(new Account(mr));
16
17        displayAccounts();
18    }
19
20    private void displayAccounts() {
21        for (Iterator it=accountList.iterator(); it.hasNext(); )
22            System.out.println(it.next());
23    }
```

Minimal proto-Bank class

Reads, stores and displays a database from a user selected file. Written to test input, storage and retrieval of a database of Accounts.

Line 15: Reads in the entire file and stores it in the database!

Notice the `display()` method. It is written as a method (instead of just a loop) so it can be reused; and uses the idiom to iterate over an Vector. Line 15 would also be good to focus on; this is the line where the database is constructed -- oddly enough. It adds one new Account to the list, by invoking the `Account(MyReader)` constructor (which reads the information for this Account from the file associated with the MyReader, mr). Since it is in a while loop, it will read all the account information, one at a time and store them all in the list, in the same order as they were in the file. That's a lot of functionality for one line, and elegantly done (although, likely aesthetics are personal).

iii) Outputting the database: save

To test the save method (once we have written it!), one line can be added, as shown in Code Example 11.11. Can you tell which class this method goes in? If you don't know,

Code Example 11.11

```
1   public BankDBMS() {
2       initComponents();
3       setBounds(100,100,500,500);
4       theBank = new Bank();
5       theBank.save();
6   }
```

Testing the Bank:save() method

Line 5 will output the database to a file of the user's choice.

answer this question; "What type does the `BankDBMS()` method return?". It is not specified. Ordinary methods must specify a return type, or void if there is none (see "return types" on page 102). Since there is no return type, this must be a constructor, and the name of a constructor is the name of the class it appears in.

c) Bank:save()

Save seems a good name for a method that saves the database to a file. The Bank must iterate over the Vector that is the internal database and write each Account to the disk file. A MyWriter will do the job perfectly; see Code Example 11.12.

Code Example 11.12

```
public void save() {
    MyWriter mw = new MyWriter();

    for (Iterator it=accountList.iterator(); it.hasNext(); ) {
        Account nextAccount = (Account) it.next();
        nextAccount.save(mw);
    }

    mw.close();
 }
```

Bank:save()

Iterates over the account list, sending each Account the `save(MyWriter)` message. Don't forget to close the file when done with it!

d) File format

The file format is entirely arbitrary, but it must be compatible with the input method. The input method expects first the name and then the balance on one line separated by at least one space.

A common pitfall is to write into the same file you are reading from before the `save()` code is completely debugged. Then the next time you try to read the data, your program crashes. You can ameliorate this by writing to a different file, or making a backup of the input file (*before* trashing it!).

e) Encapsulation: Output in the Account class

Just as input is best done within the class, so is output. Code Example 11.13 is the `save()`

Code Example 11.13

```
public void save(MyWriter mw) {
    mw.println(name + " " + balance);
}
```

Account:save(MyWriter)

Not much to it. Simply print the name and balance with a space between them.

method -- tough work, eh? This looks (and *is*) simple, but if it were written as, `mw.println(name + balance);` what would have gone wrong? Attention to detail while programming will save hours of frustration.

iv) Enhancing the DBMS

We now have a DBMS that loads and saves a list of Accounts. Its only usefulness is to demonstrate that we can do that. There are many things you might do to enhance such a database. The most obvious are to allow the user to select an account and withdraw or deposit funds. Other functions include adding and deleting Accounts, changing information in an Account, and transferring funds between Accounts.

a) Adding a java.awt.Choice

Back in Chapter 3 you selected the current Account by pushing one of three buttons. If there are dozens or hundreds of Accounts, that would make a very cluttered GUI. A Choice would be a better choice. As the name implies, it is a Component for allowing the user to make a choice between a number of options. Follow the instructions in "Adding and using a Choice" on page 338 to add one to your Application.

Following those instructions will give you a Choice with "this", "that" and "the other thing" in it. What you want in it for the Bank database is the names from all the Accounts. The easiest place to add them is when you read in the Accounts, in `Bank:input()`; that means you will need access to the Choice there. The easiest way to have a reference to it there is to pass it as a parameter with the Bank constructor (see Code Example 11.14). Remember, the Bank class will not know what Choice is unless `import java.awt.*;` is added. Notice that the reference to the Choice is passed along as a parameter to `inputAccounts()`.

b) Selecting an Account given a name

Like in the Chapter 3 example, Bank will have a `withdraw(int)` method that withdraws the amount passed in the parameter from the current account. This will require that there is a variable containing a reference to the current Account; so it must be declared and initialized. It cannot be initialized until after the database is read in, so the logical spot to do that is right after input; see Code Example 11.14. The `elementAt(int)` method returns

Code Example 11.14

```
1  import java.util.*;
2  import java.awt.*;
3
4  public class Bank {
5      private Vector accountList;
6      private Account currentAccount;
7
8      /** Creates a new instance of Bank */
9      public Bank(Choice theChoice) {
10         accountList = new Vector();
11         inputAccounts(theChoice);
12         currentAccount = (Account) accountList.elementAt(0);
13     }
14
15     private void inputAccounts(Choice theChoice) {
16         MyReader mr = new MyReader();
17         while (mr.hasMoreData()) {
18             Account newAccount = new Account(mr);
19             accountList.add(newAccount);
20             theChoice.addItem(newAccount.getName());
21         }
22         mr.close();
23     }
24
25     public void withdraw(int withdrawalAmt) {
26         currentAccount.withdraw(withdrawalAmt);
27     }
```

Initializing theChoice and currentAccount.

Line 11: Fills it with Accounts and initializes theChoice (see line 20).

Line 12: Initializes currentAccount to the first thing in the Vector. Note the cast; remember why?

the Object at the ith position in the Vector. The first position in the Vector is 0, just like an array. The cast is required for the assignment on line 12, since you cannot assign an Object to an Account. The (Account) persuades the compiler that it should do the assignment; see "iterator()" on page 263 if this seems unfamiliar.

When the user selects a new name from the choice, presumably the code looks something like Code Example 11.15. Notice that the type of item is String, so on the Bank side the

Code Example 11.15

```
1    private void newChoice(java.awt.event.ItemEvent evt) {
2        String item = theChoice.getSelectedItem();
3        System.out.println("new choice from the choice: " + item);
4        theBank.setCurrentAccount(item);
5        displayCurrentBalance();
6    }
```

itemSelected handler for theChoice

Line 2: get the selected item
Line 3: diagnostic; delete when the code works
Line 4: the point of this example
Line 5: so the user can always see the balance of the current Account

`setCurrentAccount()` method will have a String parameter and will have to search through all the Accounts looking for one with that name. As always, to iterate through a Vector, copy and paste the Iterator loop; see Code Example 11.16. This code will work,

Code Example 11.16

```
public void setCurrentAccount(String name) {
    for (Iterator it=accountList.iterator(); it.hasNext(); ) {
        Account nextAccount = (Account) it.next();
        if (nextAccount.getName().equals(name))
            currentAccount = nextAccount;
    }
}
```

Bank:setCurrentAccount(String)

Iterate over the Accounts, and if one is found with the parameter as its name, set currentAccount to it.

but could be improved in two ways. First, the loop will continue through the entire list even if it finds the name in the first Account; so it will waste fewer cycles if it exits the loop as soon as it finds the name (although, since searching the entire list takes less than a millisecond, this hardly matters). Second, if it never finds the name, that's a bug, and it would be good to report the problem, instead of blithely ignoring it. Code Example 11.17 fixes both of these by returning when the name is found and complaining if control falls out the bottom of the loop without having found the name. This loop with an internal return is an idiom that is worthwhile remembering; you will see it again (assuming you continue on in computing!).

Code Example 11.17

```
1    public void setCurrentAccount(String name) {
2        for (Iterator it=accountList.iterator(); it.hasNext(); ) {
3            Account nextAccount = (Account) it.next();
4            if (nextAccount.getName().equals(name)) {
5                currentAccount = nextAccount; // found it
6                return;                       // exit the method NOW!
7            } // if
8        } // for
9
10        System.out.println("Error! Name not found! name=" + name);
11    }
```

Bank:setCurrentAccount(String) improved

Line 4-7: If the name is found, set currentAccount and exit the method.

Line 10: Complain about not finding it; realize it will only get here if line 6 never executes.

c) Editing an Account

Part of database management is the ability to modify the data, both to correct errors, and simply to update changing information. This may be done by the software (like updating the balance when money is withdrawn) or by hand (like changing a misspelled name). Conceptually here's what to do to allow editing.

1. Add an edit button (it's action is to tell the Bank to edit).
2. Open a new window to edit the current account.
3. Reflect changes the user makes in the edit window in the database.

Step 1 is easy, you know how to add and connect a Button. The other two steps need some explaining.

- **Opening a new window**

Opening a new window is very useful when you wish to present or input information under certain circumstances, but don't want to clutter up the GUI. Displaying a new Frame is described in "Adding a pop-up Frame" on page 336.

- **Editing from the new window**

Once you have created a GUI Frame Form, you can use the FormEditor to add TextFields for each field in the Account to be edited; as of now that would just be name and balance. Rename them and connect them so that you get control when the user hits enter in them.

NetBeans will write the shell of the event handling code, but you must write the code to do the editing. When the user enters a change in the name TextField, you would like to simply set the name of the current account to what they have entered, as shown in Code Example 11.18. If you simply type this code, the compiler will inform you that it cannot

Code Example 11.18

```
private void nameTFActionPerformed(java.awt.event.ActionEvent evt) {
    theAccount.setName(nameTF.getText());
}
```

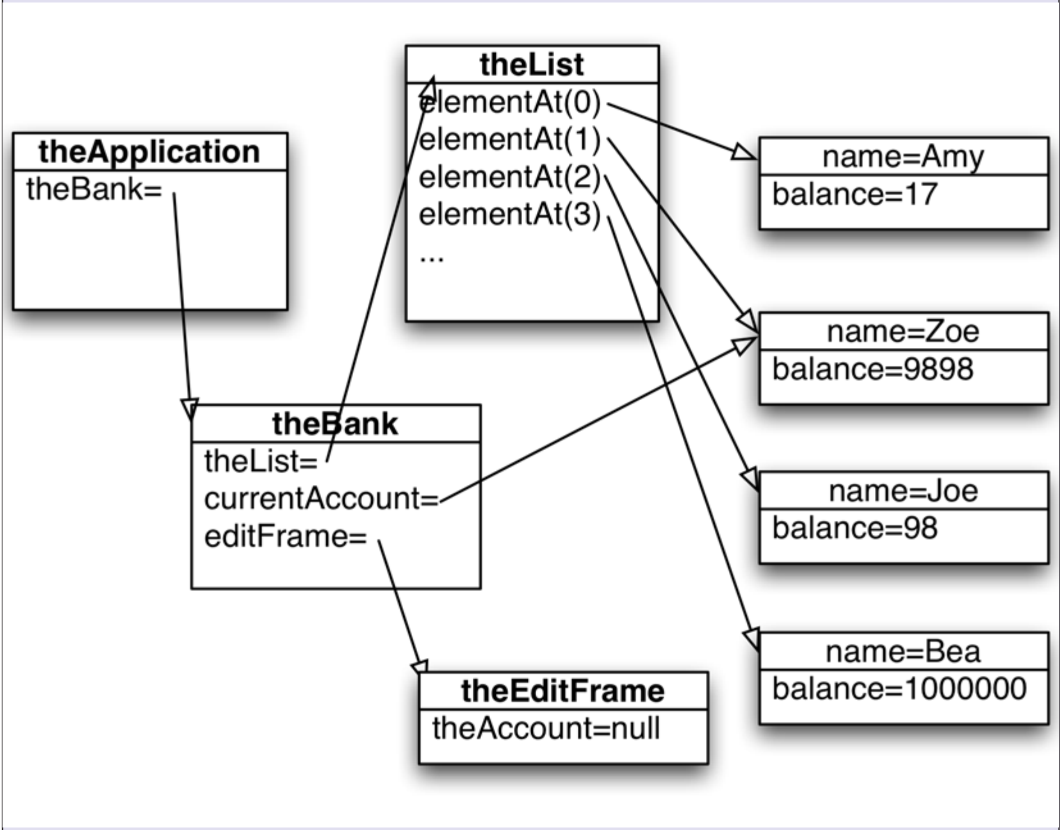
Editing the name field

Changing the value of the name in theAccount. But, how to access the currentAccount back in the Bank from the EditFrame?

resolve the symbol "theAccount", and for good reason; it is not declared in this class! You could declare it as an instance variable (Account theAccount;), and then it would compile; unfortunately at runtime this, alone, would generate a Null Pointer Exception -- again for good reason, since you have never changed the default null. Some people find themselves stuck at this point.

The solution is obvious if you think carefully. Or maybe draw a picture? Figure 11.1 depicts the data structure after the program has read in those four Accounts from before and the user has selected "Zoe" with the Choice and pressed the Edit Button. The

Figure 11.1



The state of the program when the user hits edit.

The Application has a theBank variable which has a database of Accounts, a current account, and an EditFrame. When the user pushes the Edit Button, an EditFrame is opened; but, how can the EditFrame get access to the current Account?

problem is that theAccount in theEditFrame is null. It should have been set to point to Zoe's Account, which is the currentAccount back in theBank. How could this information be passed from theBank to theEditFrame? You already know, right? Either with an accessor, or as a parameter to the constructor. That easy.

Once we have access to the current account from the edit window, Code Example 11.18 will work perfectly. The code to update the balance is similar, except that the String from the TextField must be converted to an int (see "String to int" on page 123 if it has slipped your mind).

- **Error checking**

The code above assumes that the user will type a legal int into the balanceTF in the EditFrame. What if they accidently type some letters? Try it out. It throws an Exception. It would be good to catch that Exception. It would also be nice to read the values from both TextFields whenever the user hits enter (imagine how annoying it would be to type a new name an balance, hit enter, and only update the balance). But, that is deferred to the exercises; it's just details.

E. Molecules in box

i) The programming task

In Chapter 9 a single ball bouncing under the influence of gravity was simulated. The task for this section is to simulate a number of molecules in a box. Most of the code from the BallApplet can be leveraged for this new task.

ii) The Molecule class

The Ball class checked for bouncing on the floor, but not on the ceiling or the two side walls. Thus, the Molecule class will need to check for those other three cases as well. Perhaps you are ready to write that code now; in that case, do it, and when you're done go on to "Changes to the Controller" on page 288. Otherwise read on.

a) Designing the bounce code

The code to check whether a Molecule will collide with any of the four walls on this time step is a little bit complicated. Whenever you are writing code that is not simple, it is important to think clearly before starting, otherwise you can waste many frustrating hours debugging, sometimes with nothing to show for it when you are done.

First, you need a clear conception of the problem. Second, you need to formulate a simple way to solve the problem. Start with a simple approach that you understand, or writing and debugging the code is likely to be a disaster.

Two problem solving techniques spring to mind here: Draw a Picture, and Analysis By Cases; if those don't seem familiar, you might want to review them before continuing (see "Draw a picture." on page 75 or, "Analysis By Cases (ABC)" on page 161). Then, using the picture for the bottom as a model (see Figure 9.1 on page 228), draw the pictures for the top, right and left.

b) Applying ABC

- **Step 1: Distinguishing the cases**

Now, apply the ABC technique. Use the pictures you have drawn for the first step. Once you have decided how to determine which of the five cases is applicable, then you can decide what to do in each case. Only then should you start writing code.

Are there really five cases? At least. The molecule may bounce on any of the four walls this time step, or none. That makes five cases. Perhaps there are four more cases: bouncing off the left *and* top, the left *and* bottom, right *and* top, and right *and* bottom. It is possible to check the four bounce cases so as to handle all eight cases; nevertheless it is something to keep in mind.

- **Step 2 - Deciding on actions for each case**

The action required in each case is similar. If the molecule would hit the wall this time step, reverse its velocity (x or y, depending) and calculate its position after the bounce.

c) Writing the code

- **Adapting the Ball bounce code**

You already have code for the floor bounce (see Code Example 11.19). Odds are you can

Code Example 11.19

```
1 public void step() {
2     x += vx;
3
4     int bottomY = 750-(radius+y);
5     if (vy >= bottomY) {
6         //System.out.println("vy=" + vy + " y=" + y);
7          int bounceHt = vy - bottomY;    // how high it bounces this step
8          y = 750 - (radius + bounceHt);  // y-coord after this step
9          vy = -(vy-1)*9/10;
10    }
11    else {
12        y += vy;
13        vy += GRAVITY;
14    }
15}
```

The Ball:step() improved? (Copied from: Code Example 9.12 on page 231)
Line 7: Calculate how far it will travel up after bouncing
Line 8: From that calculate the new y-coord of the center
Line 9: Reverse vy and subtract 1 to avoid endless small bounces.

adapt this code for the other three cases. Take a minute to review this code and get it in your head so you'll know how to modify it. ... Okay. Give it one more try... Huh. Maybe you can't make head nor tail of that code. Not good. Too much tricky code. Sorry about that. Perhaps the way to make it make sense of it is by using stepwise refinement (see "Stepwise refinement" on page 41).

- **Stepwise refining the bounce code**

Let's start with the `step()` method for Molecule without worrying about whether the Molecule will hit the wall. It simply updates the x and y-coordinates and the y velocity of the Molecule, as shown in Code Example 11.20. One of the reasons `step()` in the

Code Example 11.20

```java
public void step() {
    x += vx;
    y += vy;
    vy += g;
}
```

Molecule:step() without considering bounces

previous example is complicated is that the code for `step()` and `bounce()` is intertwined. These two parts can be separated as shown in Code Example 11.20. Here the use of

Code Example 11.21

```java
    public void step() {
       if (!willHitWall()) {
           x += vx;
           y += vy;
           vy += GRAVITY;
       }
       else bounce();
   }
```

Molecule:step() considering bounces

If the Molecule will not hit the wall this step just update x, y, and vy; otherwise it bounces.

methods with descriptive names makes the logic clear. The downside is that now the `willHitWall()` and `bounce()` methods must be written. Code Example 11.20 shows those

Code Example 11.22

```java
     private boolean willHitWall() {
        return willHitSide() || willHitTopOrBottom();
    }

    private void bounce() {
        if (willHitTopOrBottom())
            handleYBounce();
        if (willHitSide())
            handleXBounce();
     }
```

willHitWall() and bounce()

Lines 1-3: willHitWall() returns true just if willHitSide() or willHitTopOrBottom is true.
Lines 5-10: handles a y-bounce, or an x-bounce, or both

two methods. Notice that both of them depend on `willHitSide()` and `willHitTopOrBottom()`. The Molecule will hit a wall just if it hits a side wall or the top or bottom; thus `willHitWall()` returns that. The `bounce()` method may have to handle a bounce in the y-direction or the x-direction, or both. If it were written as an if-else, it would fail in the cases where the Molecule were going to hit two walls on the same time step.

There are now four more methods that need to be written (stepwise refinement tends to generate a number of methods). They are shown in the next two listings. Code Example

Code Example 11.23

```
1     private boolean willHitTopOrBottom() {
2         if (vy > 0) {
3             return vy > distanceToBottom();
4         }
5         else {
6             return -vy > distanceToTop();
7         }
8     }
9
10    private boolean willHitSide() {
11        if (vx > 0) {
12            return vx > distanceToRightWall();
13        }
14        else {
15            return -vx > distanceToLeftWall();
16        }
17    }
```

checking for upcoming bounces

Two methods to tell whether a Molecule would hit the wall this time step.

11.23 has the two methods that detect upcoming bounces. The logic is the same for both, so only the first of the four will be explained. If the y-component of velocity is downwards (i.e. if vy > 0, see line 2) then it would hit the bottom in the next time step just if the distance it will move in the y direction (vy) is more than the distance to the bottom (line 3). The distances are calculated by the methods in Code Example 11.24.

Code Example 11.24

```java
private int distanceToBottom() {
    return HT-(radius+y);
}

private int distanceToTop() {
    return y-radius;
}

private int distanceToRightWall() {
    return WIDTH - (x+radius);
}

private int distanceToLeftWall() {
    return x-radius;
}
```

Calculating the distance to the wall the Molecule is headed towards.
Four methods that implement the four distance cases.

Refer to your pictures to make sure these make sense. You will notice the use of WIDTH and HT in these methods. You no doubt recall that identifiers in all caps are constants (see "Case conventions" on page 128). It is better to define variables for values like this to avoid embedding numbers like 750 in the code (as was carelessly done in the first version of `step()` in Ball). The reason is, if you change the size of the display, it will be done automatically (instead of having to search for the numbers that represent the size of the display in various classes. This is accomplished as shown in Code Example 11.25.

Code Example 11.25

```
1  // from MoleculeApplet.java
2  public class MoleculeApplet extends java.applet.Applet {
3      Controller theController;
4      public static final int WIDTH=900;
5      public static final int HT=900;
6
7
8  // from Molecule.java...
9  public class Molecule extends FilledCircle {
10     private int HT=MoleculeApplet.HT;
11     private int WIDTH=MoleculeApplet.WIDTH;
12     private double ELASTICITY=1.0;
13
```

Setting and accessing the display dimensions.

WIDTH and HT are declared static and public in Molecule, so they are accessible from everywhere (as shown on lines 10 and 11).

The only remaining methods are `handleXBounce()` and `handleYBounce()`. The logic of these two methods is a little tricky. The author got it wrong twice and spent more hours than he wants to admit to to get it right. There are still four cases to consider (two in each method. The logic for calculating the new values for x and y is shown in Code Example

Code Example 11.26

```
handleYBounce() {
    case 1: bottom -- y = HT - reboundDistance() - radius;
    case 2: top    -- y =      reboundDistance() + radius;

handleYBounce() {
    case 3: right -- x = WIDTH - reboundDistance() - radius;
    case 4: left  -- x =         reboundDistance() + radius;
```

Pseudocode for handling bounces

Logically, if it hits the top or the top or the left the new position is just the distance it rebounds plus the radius. In the other two cases the rebound distance and the radius must be subtracted from the wall position.

11.27. The only remaining question is how to calculate the distance the Molecule rebounds on this time step. If you care about that detail, it is included in Code Example

Code Example 11.27

```
    private void handleYBounce() {
        if (vy >= distanceToBottom()) {
            y = HT - (vy - distanceToBottom()) - radius;
            vy = - (vy-1)*9/10;
        }
        else { // top
            y = -vy -(distanceToTop() - radius) + radius;
            vy = -vy * 9/10;
        }
    }

    private void handleXBounce() {
        if (vx > 0) // right wall
            x = WIDTH - (vx - (WIDTH - x - radius)) - radius;
        else x = -vx - (x-radius) + radius;

        vx = -vx * 9/10;
    }
```

handleYBounce() and handleXBounce()

The detailed code to handle bounces; *finally* the stepwise refinement bottoms out!. To understand this code, draw a careful picture. Or ignore it, it's hardly pivotal to understanding computing.

11.27. The 9/10's should be ELASTICITY, but if it is declared as:

```
    int ELASTICITY = 9/10;
```

something terrible happens. Do you know what? See Code Example 5.16 on page 124 for a clue. And if it is declared as

```
    double ELASTICITY = 0.9;
```

then it will not compile unless `vx * ELASTICITY` is cast as an int; like this:

```
        vx = (int) (-vx * ELASTICITY);
```

Which clutters up the code (although it would be okay to do).

Finally we are done writing Molecule and can turn to testing it.

iii) Changes to the Controller

a) A single Molecule

To make the Controller from the BallApplet simulate and animate a single Molecule instead of a Ball is very simple. The changes needed are shown in Code Example 11.28.

Code Example 11.28

```java
import java.awt.*;
import java.applet.*;

public class Controller extends Thread {
    private Applet theApplet;
    private Molecule theMolecule;
    private boolean running=true;

    /** Creates a new instance of Controller */
    public Controller(Applet theApplet) {
        this.theApplet = theApplet;
        theMolecule = new Molecule(10,40,20,Color.RED, 1, -5);
    }

    public void paint(Graphics g) {
        theMolecule.paint(g);
    }

    private void step() {
        theMolecule.step();
        theApplet.repaint();
    }

```

Modified Controller from the BallApplet

This Controller is very similar to the one in Code Example 9.7 on page 224; the differences are that every Ball has been replaced by a Molecule.

All that has been done is to replace Ball with Molecule everywhere. The methods that are unaffected are omitted.

b) Many Molecules

To make the Controller simulate and animate many Molecules is slightly more complicated. Instead of a Molecule variable, it needs a Vector (which will contain all the

Molecules). Then, everywhere the original code did something with a Molecule, the new code must iterate over all the Molecules. There are three methods that need to be changed, the constructor, `paint()` and `step()`.

- **Changes to the constructor**

Code Example 11.29 shows the changes needed to create and store NUM_MOLECULES

Code Example 11.29

```
1  public Controller(Applet theApplet) {
2      this.theApplet = theApplet;
3      for (int i=0; i<NUM_MOLECULES; i++)
4        addOneMolecule();
5      }
6  }
7
8  private void addOneMolecule() {
9    theList.add(new Molecule(100,100,20,Color.RED, rand(7), rand(7)));
10 }
```

Changes to the constructor

Line 2: Loop NUM_MOLECULES times (if you have questions about this, see Code Example 8.6 on page 198).
Line 3: Each time around the loop, uh, add one Molecule.
Lines 8-10: The `addOneMolecule()` method.
Line 9: Create a red Molecule at 100, 100, radius 20, with random vx and vy (between 0 and 6)

Molecules in a Vector. Notice that they all start at the same place, are the same color and size, but have random velocities. The use of `rand(int)` requires that it be defined (see Code Example 5.19 on page 126 for a reminder). You might want to experiment with random sizes and colors once the code is working.

This code assumes that NUM_MOLECULES and theList are defined, and that `java.util.*` is imported: see Code Example 11.30 for the changes needed at the beginning of Controller.

Code Example 11.30

```
1 import java.awt.*;
2 import java.applet.*;
3 import java.util.*;
4
5 public class Controller extends Thread {
6     private final int NUM_MOLECULES=20;
7
8     private Applet theApplet;
9     private Vector theList = new Vector();
10    private boolean running=true;
11
12
```

Changes to the beginning of Controller

Line 3: imports java.util so that Vector and Iterator are defined.
Line 6: declare the constant NUM_MOLECULES to be 20.
Line 9: declare and initialize a Vector called theList.

- **Changes to paint() and step()**

The `paint()` method for the bouncing ball had just one line:

```
    theMolecule.paint(g);
```

To make it instead send `paint(g)` to every Molecule in the list requires an Iterator in a loop, as shown in Code Example 11.31. The changes required for `step()` are very similar;

Code Example 11.31

```
    public void paint(Graphics g) {
        for (Iterator it=theList.iterator(); it.hasNext();) {
            Molecule nextMolecule = (Molecule) it.next();
            nextMolecule.paint(g);
        }
    }
```

Modified paint() to paint every Molecule in the list

Uses the idiom for iterating over all elements of a list (see "iterator()" on page 263).

instead of sending `step()` to theMolecule, it must be sent to every Molecule in the list; see Code Example 11.32.

Code Example 11.32

```
  private void step() {
    for (Iterator it=theList.iterator(); it.hasNext();) {
        Molecule nextMolecule = (Molecule) it.next();
        nextMolecule.step();
    }
    theApplet.repaint();
}
```

Modified step() to step every Molecule in the list

Like the previous Example, this again uses the idiom for iterating over the elements of a list.

iv) Experimenting with the program

Having made those changes, the program is ready to run. Experiment with different elasticities, turning off gravity, different delay times, different ranges of velocities, or different numbers of molecules. How many molecules can there be before it stops looking like animation? Note: if you wish, you may access the code at:
http://www.willamette.edu/~levenick/SimplyJava/code/molecules/
It would be more educational to write it yourself, but if you don't have the time or inclination, experimenting with my code is better than nothing.

F. Conclusion

This chapter introduced Vector and Iterator from java.util, Choice and Frame from java.awt, and techniques to read/write a database from/to disk files. It included a lot of new material and combined most of the material from the previous chapters to make two substantial programs. If you understand both of those examples well, congratulations! If not, condolences; you might consider rereading and working through the chapter again. Or, perhaps, doing the exercises will help solidify your understanding.

G. End of chapter material

i) Review questions

11.1 What are the two methods you use with an Iterator?
11.2 How do you add an Object to the end of the list in a Vector?

11.3 Write the idiom to access every Object in a Vector and send it to System.out.
11.4 How were the 9 ways a Molecule could bounce collapsed into 5? What are those 5?

ii) Programming exercises

The next exercises refer to the BankDBMS:

11.5 Add a Save button. Save to a file the user specifies. Then (after that works) save to the same file that the database was input from. Hint: save the file name and path when it is input; one String variable will do it.
11.6 Modify the EditFrame code so that both the name and balance fields are input and both the name and balance are updated when the user hits Enter in either. Hint: write an update() method that is invoked from both.
11.7 Add error checking code so that if the entered balance is not an int the user is notified in a reasonable manner; maybe pop up a Panel?
11.8 Add an addAccount button. You could reuse the EditFrame class to get the info for the new Account.
11.9 Add a deleteAccount button. You will find the `Vector:remove(Object)` method very useful here; as in the`AccountList.remove(currentAccount);` Don't forget to remove the deleted Account from the Choice. Reading the Sun documentation will help with both of these.

The next exercises refer to the Molecules program:

11.10 Take out the -1 in (vy-1) in `handleYBounce()` inCode Example 11.27 on page 287. What happens? Explain why. Hint: add debugging printlns to display vy and y at each step.
11.11 Add code to catch mouse clicks and when the user clicks on a Molecule, make it bigger. See "Catching mouse clicks" on page 342.
11.12 Start with just two large molecules and color them solid red when they are overlapping. Hint: two circles overlap when the distance between their centers is less than the sum of their radii.
11.13 Modify your code to detect overlap in a simulation with n Molecules. Hint: you will need a loop that checks every pair of Molecules for overlap.
11.14 Difficult! Modify your code to keep the Molecules from overlapping (it is the same kind of logic as anticipating bounces off the wall. If two Molecules are going to overlap, reverse the signs on their vx and vy variables.
11.15 More difficult! Make the collisions realistic, as if the molecules were billiard balls. If two Molecules hit head on, they should rebound the way they came, but if they hit a glancing blow, their directions change in a more complicated way. Don't forget to

conserve momentum! Note: the mass of the Molecules matters -- do large Molecules have more mass?

The next exercises refer to a completely new different project

11.16 Fun! Create a crowd of Snowfolk. Make them all melt each day. This should be just like the Molecules program.

11.17 More fun! Create a crowd of Snowfolk. Then, when the user clicks on a particular SnowPerson, make all the rest of the army move to surround that one.

Chapter 12: Writing a list class

A. Introduction

The previous chapter illustrated the use of arrays and Vectors to implement lists. There are many applications where those two generic data structures will do just what you want, but there are others when it is more convenient to have a list class that serves your needs more precisely. In that case, writing your own list class makes sense. The next chapter is about sorting lists of ints. It will be convenient to have a familiar list class with elements of type int to use there.

This chapter presents two different list classes, one based on an array, the other based on a Vector; it will be up to you to decide which you like better. The style of this chapter is rather more terse, and assumes rather more sophistication of the reader, than the earlier chapters. You are encouraged to try out all of these methods as you go along, if you actually want to understand them. Or perhaps this is so obvious that you don’t need to?

B. Designing a list class

As always, the first question in designing a class is, "What must it do?". Think about what you can do with an ordinary paper list. You can add, delete, or replace elements, and you can read elements from the list. Each of those actions can happen anywhere in the list; i.e. you can add (or read, or delete, or replace) an element at any position in the list. If someone asked you how many things were on the list, you could count the elements. The list we write must be able to do those things; additionally, we will need `toString()` and a constructor.

Next, we must settle on the signatures of the methods. Every action involves the position of the element in the list, so there will be an index as a parameter. Every element in the list will be of type int, so if an element is being passed in or returned it will have type int as well. Care must be taken not to accidently exchange the two parameters (since finding bugs stemming from having the actual parameters in one order and the formal parameters in the other, can be very elusive).

The signatures of the methods follow:

```
void addElementAt(int nuElement, int i);
void deleteElementAt(int i);
int elementAt(int i);
void replaceElementAt(int nuElement, int i);
int length();
```

Although it is not strictly necessary, a method that adds an element at the end of the list is also convenient:

```
void addElement(int nuElement);
```

These signatures will be common to the two implementations, and would make sense collected together in an interface, but that will be deferred.

C. Implementation using an array: MyArrayIntList

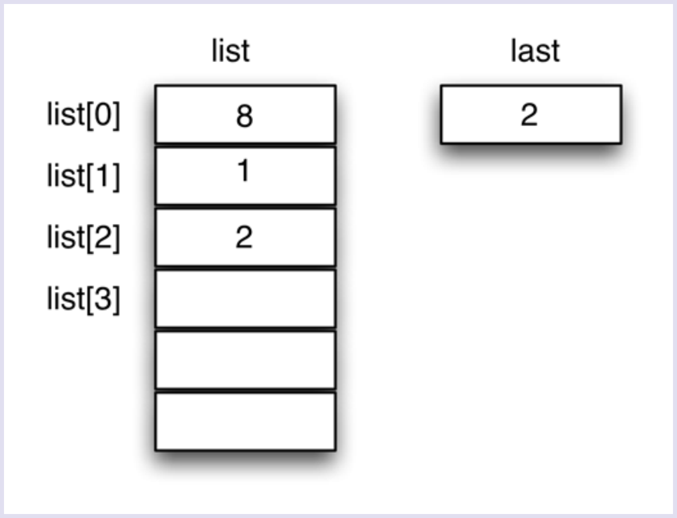
Since you can add and delete elements, the number of things in the list is variable. An array must be declared as a fixed size. If you declare an array as `int [] list[] = new int[10];` you have 10 int variables named `list[0], list[1], ... list[9]`, and no more. The array cannot contain more than 10 ints and if there are less than 10 int values to store, the rest of the variables are unused. The array examples in Chapter 11 sidestepped this issue. Here it must be confronted.

i) Managing a variable sized list in an array: representation

One simple technique to manage an array of variable size, is to keep all the elements at the beginning of the list and add a variable, named last, that keeps track of the index of the last element in the list. If there is only one element in a list, it will be at list[0] and last will be 0. If there are three elements in the list, they will be at list[0], list[1], and list[2]; and last will be 2. If the list currently contains {8,1,2}, its state will be as in Figure 12.1.

If 4 is then added to the end of the list, it will be added at list[3] and last will change to 3,

Figure 12.1

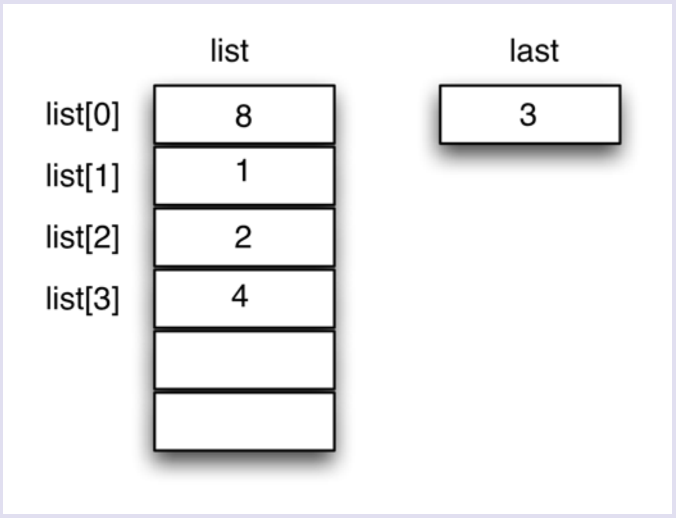


The state of the list

The list is {8,1,2}. There are three numbers in the list, so last is 2 (the index of the last number).

as shown in Figure 12.1. This simple technique allows you to keep track of a variable

Figure 12.2



The state of the same list after adding 4 to the end.
The list is now {8,1,2,4}. There are 4 numbers in the list, so last is 3.

sized list using an array. Only one variable, last, is needed in addition to the array.

Assuming you understand this list representation, it is now time to turn to implementation.

ii) Declarations and initialization

Two variables must be declared and initialized, the array, named list, and the variable that keeps track of the index of the last element in the list, named last. To make sure the array will not become full, it is allocated with space for 10000 elements, see Code Example 12.3.

Code Example 12.1

```
public class MyIntArrayList {

   private static final int MAX_VALUE = 10000;
   int [] list;
   int last;  // where is the last thing in this list

   /** Creates a new instance of List */
   public MyIntArrayList() {
       list = new int[MAX_VALUE];
        last = -1;
    }
}
```

MyIntArrayList: Declaration and initialization of variables.

Notice the use of the `static final int MAX_VALUE`. You may recall that this is how one defines a constant in Java; final means it cannot be assigned a new value. The constructor initializes the two instance variables. You may wonder why last is initialized to -1. If so, a moment's thought may yield insight; otherwise, this may help you understand it. When there are 3 things in the list, last is 2; when there are 2 things in the list, last is 1; when there is one thing in the list, last is 0. I.e. the value of last is always one less than the number of things in the list. See `makeEmpty()`, below for a dramatic use of this fact.

iii) Adding an element to the end of the list

To add an element, call it nuElement, to the end of the list, its value should be assigned to the variable list[last+1]. Since there is now one more thing in the list, last should be incremented. So, one might write the code as:

```
list[last+1] = nuElement;
last++;
```

But it would be clearer to increment last first, and then use it directly, as in:

```
last++;
list[last] = nuElement;
```

Or, shorthand would be to use the pre-increment, as shown in Code Example 12.2:

Code Example 12.2

```java
public void addElement(int nuElement) {
    list[++last] = nuElement;
}
```

MyIntArrayList:addElement()

iv) Adding an element at a particular location

Adding an element at a specific index is a bit more complicated, since first the rest of the list must be shifted down by one (to get it out of the way). Each element below the index where the new element will be inserted must be shifted down by one location. After they have all been shifted the new element may be inserted. Assuming putHereIndex is the parameter which tells where to insert the new element, every element from last up to putHereIndex, must be moved down one space. See Code Example 12.3 for how to accomplish that

Code Example 12.3

```java
public void addElementAt(int nuElement, int putHereIndex) {
    for (int i=last; i>=putHereIndex; i--)
        list[i+1] = list[i];

    list[putHereIndex] = nuElement;
    last++;
}
```

MyIntArrayList:addElementAt()

v) toString()

The toString() method for a list should display all the elements of the list in order. We have not written an iterator for this list, so Code Example 12.4 shows how to do it by

Code Example 12.4

```
public String toString() {
    String returnMe="{";

      for (int i=0; i<length(); i++)
        returnMe += "," + list[i];

    return returnMe+"}";
}
```

MyIntArrayList:toString()

hand. That for loop is a generic loop for dealing with a list represented with last. In the context of that loop, list[i] is "each element in the list". Notice the use of {}'s and commas to format the list a bit. This code has a bug in that with the list {8,1,2}, it returns "{,8,1,2}". For now the extra comma will be ignored, but will be fixed up in the exercises.

vi) Deleting an element

When an element in the middle of the list is deleted, all the elements below it must be shifted up by one position to fill the gap. Code Example 12.5 illustrates how this is done.

Code Example 12.5

```
public void removeElementAt(int removalIndex) {
    for (int i=removalIndex; i<last; i++)
        list[i] = list[i+1];
    last--;
}
```

MyIntArrayList:removeElementAt()

After deletion, i.e. shifting all the elements after the deletion point up by one, there is one less thing in the list, so last must be decremented.

vii) Replacing an element

Unlike insert, and delete, replace does not require any shifting, so it is just one line, as shown in Code Example 12.6

Code Example 12.6

```
public void replaceElementAt(int nuValue, int i) {
    list[i] = nuValue;
}
```

MyIntArrayList:replaceElementAt()

viii) length

The length of the list is one more than last, thus length is extremely simple, as in Code Example 12.7.

Code Example 12.7

```
public int length() {
    return last+1;
}
```

MyIntArrayList:length()

ix) Accessing an element

The accessor for an element at a particular location simply returns that element, as in Code Example 12.8.

Code Example 12.8

```
public int elementAt(int i) {
    return list[i];
}
```

MyIntArrayList:elementAt()

x) Making the list empty

Since all operations on the list depend on the value of last, all that is necessary to make the list empty is to set last to -1, as shown in Code Example 12.9. If this seems

Code Example 12.9

```
public void makeEmpty() {
    last = -1;
}
```

MyIntArrayList:makeEmpty()

mysterious, draw a picture and make sure you understand what all the methods would do with last=-1.

D. Implementation using a Vector: MyVectorIntList

The vector class automatically handles lists of variable sizes, so there is no need for our Vector based class to keep track of that information and we can eliminate the variable named last. On the other hand, a Vector stores Objects, and int is a primitive type; thus, to store an int in a Vector it must first be wrapped up in an object. When the wrapper is taken out of the Vector, the int value must be unwrapped. Fortunately, both those operations are simple, if mildly annoying.

Java has a built-in int wrapper, named Integer. Given an int variable, x, you can create an Integer that contains the value of x, by:

```
new Integer(x);
```

That’s all it takes.

The Integer class has a method, `intValue()`, that returns the int value that is stored inside the Integer.

```
Integer anInteger = new Integer(17);
int x = anInteger.intValue();
System.out.println("x=" + x);
```

Will output "x=17".

The My VectorIntList class is what is called an adapter class; it adapts Vector for use with ints. Every method below, except toString(), has a body with just one line, which sends the appropriate message to the Vector that MyIntVectorList wraps up. You would do well to glance at the documentation for Vector before reading the next section; either at the Sun site, or in the IDE.

i) Declarations and initialization

Since this class uses a Vector, `java.util.*` must be imported, but initialization consists only of instantiating the Vector, see Code Example 12.10.

Code Example 12.10

```java
import java.util.*;

public class MyIntVectorList {

Vector list;

/** Creates a new instance of List */
public MyIntVectorList() {
    list = new Vector();
}
```

MyIntVectorList: Declaration and initialization of the variable.

ii) Adding an element to the end of the list

Vector has an `addElement(Object)` method, so all that has to be done is wrap the int in an Integer, as shown in Code Example 12.11:

Code Example 12.11

```java
public void addElement(int nuElement) {
    list.addElement(new Integer(nuElement));
}
```

MyIntVectorList:addElement()

iii) Adding an element at a particular location

Vector has an `addElementAt(Object, int)` method, so that can be used, once the int is wrapped, as shown in Code Example 12.12.

Code Example 12.12

```
public void addElementAt(int nuElement, int putHereIndex) {
    list.insertElementAt(new Integer(nuElement), putHereIndex);
}
```

MyIntVectorList:addElementAt()

iv) toString()

The toString() method can use Vector's `iterator()`, as shown in Code Example 12.13.

Code Example 12.13

```
public String toString() {
    String returnMe="MyIntVectorList: {";

    for (Iterator it=list.iterator(); it.hasNext();)
        returnMe += "," + it.next().toString();

    return returnMe + "}";
}
```

MyIntVectorList:toString()

Aside from that, this is identical with Code Example 12.4, including the extra leading comma bug.

v) Deleting an element

Vector has an `removeElementAt(int)` method, so that can be used, as in Code Example 12.4.

Code Example 12.14

```
public void removeElementAt(int removalIndex) {
    list.removeElementAt(removalIndex);
}
```

MyIntVectorList:removeElementAt()

vi) Replacing an element

Vector has an `replaceElementAt(Object, int)` method, so that can be used once the int is wrapped, as shown in Code Example 12.15.

Code Example 12.15

```
public void replaceElementAt(int nuValue, int i) {
    list.setElementAt(new Integer(nuValue), i);
}
```

MyIntVectorList:replaceElementAt()

vii) length

Vector has an `size()` method, so this method returns that; see Code Example 12.16

Code Example 12.16

```
public int length() {
    return list.size();
}
```

MyIntVectorList:length()

viii) Accessing an element

Vector has an `elementAt(int)` method, so that is returned. But first the Integer must be unwrapped. The `intValue()` method will return the int the Integer wraps; but `elementAt()` returns an Object, which must first be cast as an Integer (recall "iterator()"

on page 263); see Code Example 12.17. This is the only method where you must cast the

Code Example 12.17

```java
public int elementAt(int i) {
    return ((Integer) list.elementAt(i)).intValue();
}
```

MyIntVectorList:elementAt()

Object returned from Vector, and then unwrap that Integer to obtain an int. If you used a Vector to store ints and did not write an adaptor class like this you might end up doing it all over your code and get very tired of it.

ix) Making the list empty

Vector has an `removeAllElements()` method, which does the job we need done, as shown in Code Example 12.18.

Code Example 12.18

```java
public void makeEmpty() {
    list.removeAllElements();
}
```

MyIntVectorList:makeEmpty()

E. Testing

If you were writing this code from scratch (which would be a good idea, if you wanted to internalize it and be ready to use it, or take a test involving it), you would naturally only write one or two methods at a time, and test them before writing more. This would help you avoid making the same mistake over and over in every method and having to fix them all once you realized the problem. You would write the constructor, `addElement()` and `toString()` first (since that is the minimum needed for testing) and then add one or two more at a time. That code might look like Code Example 12.19. Once this code

Code Example 12.19

```
void initList() {
    theList.addElement(8);
    theList.addElement(3);
    theList.addElement(2);
    theList.addElement(1);
    theList.addElement(4);
}

void testList() {
     theList = new MyIntVectorList();
     initList();
     System.out.println("after initializing, it's:" + theList);
 }
```

Initial testing code

produces correct output, you know the constructor, `addElement()` and `toString()` work, so it would be time to write and test the other methods. Code Example 12.20 shows such a test.

Code Example 12.20

```
void testList() {
    theList = new MyIntVectorList();
    initList();
    theList.addElementAt(17,0);
    theList.addElementAt(177,3);
    theList.replaceElementAt(222,2);
    theList.removeElementAt(3);
    theList.removeElementAt(4);

     System.out.println("{17,8,222,177,4}?" + theList);
 }
```

Testing three more methods

Notice that the `println()` has the correct output as a String constant, so it will be easy to tell if the output is correct.

F. JavaDoc

Code Example 12.21 shows the addElementAt() method with and without Javadoc

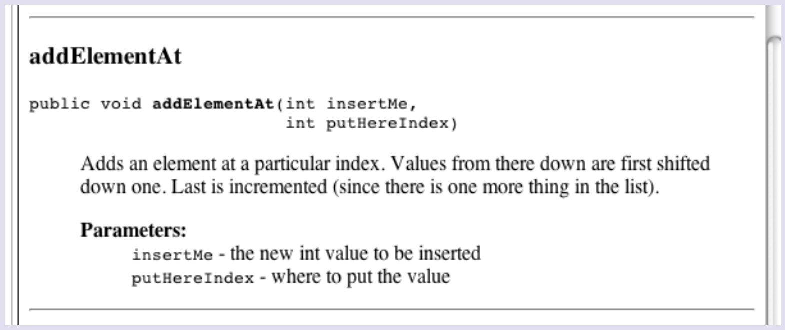
Code Example 12.21

```
public void addElementAt(int nuElement, int putHereIndex) {
    list.insertElementAt(new Integer(nuElement), putHereIndex);
}

/**
 * Adds an element at a particular index.
 * Values from there down are first shifted down one.
 * Last is incremented (since there is one more thing in the list).

 * @param nuElement the new int value to be inserted
 * @param putHereIndex where to put the value
 */
public void addElementAt(int nuElement, int putHereIndex) {
     list.insertElementAt(new Integer(nuElement), putHereIndex);
 }
```

Code with and without comments

comments. Figure 12.3 is the portion of the beautifully formatted documentation these

Figure 12.3



addElementAt

```
public void addElementAt(int insertMe,
                         int putHereIndex)
```

Adds an element at a particular index. Values from there down are first shifted down one. Last is incremented (since there is one more thing in the list).

Parameters:
`insertMe` - the new int value to be inserted
`putHereIndex` - where to put the value

Javadoc output from Code Example 12.21

comments produce. If your goal is to produce beautifully formatted, professional quality documentation, Javadoc is a great tool! Its use is described in NetbeansAppendix Q on page 343.

G. Conclusion

This chapter has presented two implementations for a list of ints, one based on an array, the other based on a Vector. These will be used in the next chapter. It also introduced Javadoc for documenting Java programs. Now that we have a working int list class, we can move on to sorting lists in Chapter 13.

H. End of chapter material

i) Review questions

12.1 What is an adapter class?
12.2 How do you get the int value into and out of an Integer wrapper?
12.3 Which of the two int list implements do you prefer? Why?
12.4 Why is it good practice to only write a few methods at a time when implementing a large class using an unfamiliar data structure?
12.5 What s wrong with this code?

```
Account [] accountList = new Account[1000];
System.out.println("First Account name is=accountList[0].getName());
```

12.6

ii) Programming exercises

12.7 What would go wrong with the `addElementAt()` method in Code Example 12.3 if the loop were rewritten as shown?

```
for (int i=0; i<=last; i++)
    list[i+1] = list[i];
```

12.8 Fix the extra comma bug in the `toString()` method in Code Example 12.4. Hint: the problem is that the comma is always added before the element, but it shouldn't be added before the first element. Thus, the action of adding a comma should only be performed under certain conditions.
12.9 Fix the extra comma bug in the `toString()` method in Code Example 12.13. Hint: the problem is identical with the previous exercise, but it is not as easy to tell what the condition is. Worst case, you could add a variable to tell you if it is the first time around the loop, but, a more elegant solution appends the comma after the int only if there is another int coming (you can check `it.hasNext()`)

12.10

Chapter 13: Sorting lists

A. Introduction

Sometimes lists need to be sorted, for a number different reasons. Mailing lists must be sorted by zip code to reduce postage costs. Lists of names are sorted before being displayed to allow a human to find names alphabetically.

Before reading the descriptions of the sorts below, take a minute and think of an algorithm to sort a list of ints. If nothing comes to mind use the following.

Problem Solving Technique

How would you do it without a computer?

If you can't think of an algorithm for a problem, start solving it yourself and then convert the technique you would use into an algorithm.

So, write down five single digit numbers in a list and sort them. What did you do? Odds are you used one of the first two techniques that follow.

B. Intuition for three sorts

Sorting algorithms must work on lists of any length, so speaking abstractly, the length of the list is some fixed number, n. For concreteness, assume you have an unsorted list of five ints, say {8,3,2,1,4}. Sorted, this list would be either {1,2,3,4,8}, or {8,4,3,2,1}; let's use the former, smallest element first. For insertion and selection sorts it is sometimes easier to create a second list, although traditionally these sorts are done in place. If we create a second list, the original list will be referred to as unsorted and the new list as sorted. Each element from the original list will be added to the sorted list in order (using different techniques in the different sorts).

i) Insertion sort

The plan here is to start with an empty list, and repeatedly insert the next element from the unsorted list at the correct location; thus, always keeping the list in order. Starting with {8,3,2,1,4}, first we take the first element, 8, and insert it in the empty sorted list.

Thus, we will have a sorted list of length one, {8}. Then taking the next element from unsorted, 3, and inserting it in order before the 8, we will have a sorted list of length two, {3,8}. Finally we will have a sorted list of length n, {1,2,3,4,8}. Before each insertion the list is in order, and after each insertion it is still in order and one element longer. The list is always in order; sometimes this fact is referred to as an ***invariant***; invariants figure prominently in some styles of programming.

During the insertion sort, the state of the lists will be as follows (each pair is `unsorted::sorted`):

```
{8,3,2,1,4}::{}
{3,2,1,4}::{8}
{2,1,4}::{3,8}
{1,4}::{2,3,8}
{4}::{1,2,3,8}
{}::{1,2,3,4,8}
```

ii) Selection sort

This is the sort most people invent. First, select the smallest element in the unsorted list and put it the empty sorted list, yielding, {1}. Then select the second smallest and add it to the end of sorted, yielding {1,2}. Continue until all the elements are in their proper positions.

During the selection sort, the state of the lists will be as follows (again, each pair is `unsorted::sorted`):

```
{8,3,2,1,4}::{}
{8,3,2,4}::{1}
{8,3,4}::{1,2}
{8,4}::{1,2,3}
{8}::{1,2,3,4}
{}::{1,2,3,4,8}
```

iii) Bubble sort

This much maligned sort is the simplest to code. It is based on the fact that if every pair of adjacent elements are in the correct order, then the entire list is in order. Thus a local property produces a global property. The plan is to scan through the list, comparing each pair of adjacent elements and exchanging them if they are out of order. After one pass over the list (from first to last), the largest element is guaranteed to be in the last position.

After two passes, the second largest element will be in the second to last position as well. After n-1 passes, the entire list will be in order.

Thus the state of the list during the first pass will be (**bold** elements are about to be compared):

{***8,3***,2,1,4}, {3,***8,2***,1,4}, {3,2,***8,1***,4}, {3,2,1,***8,4***}, {3,2,1,4,8}

Notice that every comparison results in swapping the 8 to the right (since it is the largest item in the list and started in the first position. The states of the list during next pass will be the following:

{***3,2***,1,4,8}, {2,***3,1***,4,8}, {2,1,***3,4***,8}, {2,1,3,***4,8***},{2,1,3,4,8}

On this pass there are only two swaps as the 3 moves right twice. On the third pass there is even less movement, only the first two elements are swapped:

{***2,1***,3,4,8}, {1,***2,3***,4,8}, {1,2,***3,4***,8}, {1,2,3,***4,8***},{1,2,3,4,8}

Notice that the list is in order after only 3 passes; in general n-1 passes are required, since if the smallest element is in the last position, it can only move one position left on each pass.

No human would ever sort like this (one would hope!), but the machine does the mindless repetition, well, mindlessly, and this algorithm is very easy to code.

C. Algorithm/Pseudocode

The next step, after understanding the mechanics of an algorithm is to write pseudocode describing the operation of the algorithm to carry out that technique. Notice that for this level of description the representation of the list is not specified; it might be an array, a Vector, or some other list representation. To understand this pseudocode, you should get a pencil and paper and draw the states of the lists and keep track of the value of the indices as the loops repeat. Just glancing at the pseudocode is not likely to work; if that's what you're going to do, it might be time to put this down and do something constructive.

i) Insertion sort

```
create an empty list, called sorted
for each element of unsorted
    find where it goes in sorted*
```

```
        insert it there*
```

The *s indicate that this step requires additional specification. These subalgorithms are candidates for methods when the code is written.

```
    :*: find where an element, insertMe, goes in sorted
    for each element in sorted (call it current element)
        if insertMe < current element
            return location of current element
    return one past the end of the list (since current >= all of them)

    :*: insert an element at location i
    shift the elements from i down, down by 1
    store the element at i
```

ii) Selection sort

```
    create an empty list, called sorted
    iterate n times (with index, i, moving from first to last in unsorted
       find the location of the smallest item remaining in unsorted*
       remove it from unsorted and add it to sorted
```

Finding the minimum element is a list is something that must be done time and again. There are various ways to accomplish this. Here, we need to know where the minimum element is, as opposed to just what it is; otherwise, deleting it from the list will require finding it again. Thus, this algorithm keeps track of the index of the minimum value. Note that minIndex is initially set to the first location.

```
   :*: find the smallest item remaining in unsorted
   set minIndex to 0
   for look=1 to last location of unsorted
       if elementAt(i) < elementAt(minIndex)
           minIndex = i
```

iii) Bubble sort

```
    iterate n times
        do one pass*

    :*: do one pass
    for each element in the list (except the last)
        if it is > the next element
            exchange them
```

D. Implementation

All of these sorts can be written as nested loops. They can also be written as single loops that invokes a method or two. The latter is perhaps easier to understand. Which is preferable depends on the context and is a matter of taste.

i) Bubble sort implement ion

Code Example 13.1 shows a decomposed bubble sort. There is no way anyone who

Code Example 13.1

```
1     public void bubbleSort() {
2         // iterate n times
3         for (int pass=0; pass<list.length(); pass++) {
4             onePass();
5         }
6     }
7
8     private void onePass() {
9         // for each element in the list (except the last)
10        for (int look=0; look<list.length()-1; look++) {
11            // if it is > the next element
12            if (list.elementAt(look) > list.elementAt(look+1))
13               swap(look, look+1);    // exchange them
14       }
15   }
```

Bubble Sort decomposed

Lines 3-5: invokes `onePass()` n times (where n is the length of the list).

Lines 12-14: compares adjacent elements (at look and look+1), n-1 times, and if they are out of order, swaps them

The pseudocode was entered as comments, and the code written around it.

understands Java could be confused about what the `bubbleSort()` method does; it iterates n times (n being length of the list) -- each iteration it does one pass. By contrast Code

Example 13.2 presents a nested loop bubble sort implementation. It does exactly the same

Code Example 13.2

```
public void bubbleSort() {
    for (int pass=0; pass<list.length(); pass++) {
        for (int look=0; look<list.length()-1; look++) {
            if (list.elementAt(look) < list.elementAt(look+1))
                swap(look, look+1);
        }
    }
}
```

Bubble Sort as a single nested loop

This code does exactly what Code Example 13.1 did, but as a nested loop.

thing, except the body of `onePass()` is inserted in the loop in `bubbleSort()`. Which is better? It depends.

Notice that both alternatives use `swap(int, int)`, illustrated in Code Example 13.3. It

Code Example 13.3

```
private void swap(int here, int there) {
    int pocket = list.elementAt(there);
    list.replaceElement(there, list.elementAt(here));
    list.replaceElement(here, pocket);
}
```

Swap

Exchange the elements at here and there. Notice that the type of the elements is not specified.

swaps the elements at the two indices passed as parameters. Swap is a familiar computing idiom. The use of a temporary variable is necessary; what would go wrong with the version in Code Example 13.4?

Code Example 13.4

```
private void swap(int here, int there) {
    list.replaceElement(there, list.elementAt(here));
    list.replaceElement(here, list.elementAt(there));
}
```

Broken Swap

What's wrong with this code?

This code relies on there being a `replaceElement()` method in whatever class list instantiates (which the list classes from Chapter 12 do). It also assumes there is a variable named list in the class it exists in. What class should that variable, and the sort methods be declared? It depends. For testing purposes they could both be declared in an Applet written simply to do that testing, as shown in Code Example 13.5.

Code Example 13.5

```
1 public class SortTest extends java.applet.Applet {
2     MyList list;
3
4     /** Initializes the applet SortTest */
5     public void init() {
6         initComponents();
7         testBubbleSort();
8     }
9
10     private void testBubbleSort() {
11         list = new MyList(true);
12         System.out.println("before bubblesorting, it's:" + list);
13         bubbleSort();
14         System.out.println("after bubblesorting, it's:" + list);
15     }
```

Applet for testing bubble sort.

Code to test `bubbleSort()`.

Lines 12 & 14: Notice that these assume MyList defines `toString()`.

ii) Insertion sort implementation

Code Example 13.6 presents a decomposed insertion sort. As you can see, the code has

Code Example 13.6

```
1   public void insertionSort() {
2       //create an empty list, called sorted
3       sorted = new MyList();
4
5       //for each element of unsorted
6       for (int nextI=0; nextI<list.length(); nextI++) {
7           int nextNum = list.elementAt(nextI);
8           //find where it goes in sorted*
9           int putHereIndex = findInsertIndex(nextNum);
10          // insert it there
11          sorted.addElementAt(nextNum, putHereIndex);
12       }
13
14       list = sorted;
15   }
16
17   public int findInsertIndex(int insertMe) {
18       //for each element in sorted
19       for (int here=0; here<sorted.length(); here++) {
20           //if insertMe < current element
21           if (insertMe < sorted.elementAt(here))
22               //return location of current element
23               return here;
24       }
25       //return one past the end of the list (current >= all of them)
26       return sorted.length();
27   }
```

Insertion sort decomposed

This sort is rather more complex to code than bubble sort.

been added to the pseudocode (which was first commented out). Look back at the pseudocode to refresh your memory on the plan, before looking closely at the code. In lines 9 and 11, the value returned from `findInsertIndex()` is stored in a variable, putHereIndex, which is then passed as a parameter to `addElementAt()`. Thus, these lines could be combined to: `sorted.addElementAt(nextNum, findInsIndex(nextNum);` It is just a question of style.

Notice also that the control idiom from Code Example 11.19 is used in `findInsertIndex()`, which returns as soon as it finds an element in the list that is bigger than the value to be inserted.

The code for selection sort is left to the reader. Realize that if you type "selection sort java" into a search engine, it will find numerous implementations.

E. End of chapter material

i) Review questions

13.1 Write a method, called `min()` that returns the minimum of its two int parameters.
13.2 Write a method, called `min()` that returns the minimum of its 4 int parameters.
13.3 Write a method, called `min()` that returns the minimum of its 8 int parameters.
13.4 Write a method that is passed a list of ints (your choice which kind) and returns the maximum value of those ints.

ii) Programming exercises

13.5 Implement and test selection sort.
13.6 Add a sort button in your BankDBMS. Sort alphabetically by name. Be sure to rearrange the names in the Choice to be alphabetical as well.

Appendix 1 : Netbeans 3.6 Appendix

Netbeans 3.6 Appendix A: Getting started with Netbeans and the greetings program

a) download and install NetBeans

If NetBeans is not installed on the machine you are using (lab machines will all have it installed already), download and install NetBeans and the current JDK from the web (*netbeans.org*)

b) Start up NetBeans

Open the NetBeans Launcher to start the IDE (integrated development environment). On a PC, double-click the icon, on a Mac, single-click

c) Create a new project and mount a directory to use for it

1. Create a new project (Project/Project manager/new) with any name you want (no spaces or special characters; just letters and numbers). Netbeans will open a new project and display a blank screen with FileSystems in the upper left.
2. Create a directory for this project. All the files for this project will be stored there. It does not matter what you name it, or where it is exactly, but it will be simpler in the long run if you keep your files in a directory called programming, or some such. You can use the Explorer in a PC environment, or the Finder in a Mac environment to create a new directory (also called a folder).
3. Select File/Mount Filesystem... (this means click the File menu in the extreme upper left, then click Mount Filesystem in the list that opens).
4. Select Local Directory and click Next
5. Select the directory you created in step 2. Double-click folder icons to open them (to navigate to where you created that directory). Select that directory; then click Finish. The directory you've chosen should appear below Filesystems in the left pane.
6. Add that directory to the project. Select it (by clicking its icon) - it will then be highlighted. Then Tools/Add to Project (i.e. click the Tools menu item at the top of the screen, and then click Add to Project on the drop down menu).

d) Create a main program

1. Select File/new (note: the directory must be highlighted in the FileSystems window, otherwise there is a bug in the PC implementation that doesn't let you create a new file), open Java classes (by clicking the triangle icon on its left), then select Java Main class, click Next, and type Test as the name (it will be highlighted, so just type), finally press Finish.
2. You should, in a moment, see a file that looks like:

Appendix Code Example 1

```
1/*
2 * Test.java
3 *
4 * Created on April 11, 2004, 5:17 PM
5 */
6
7/**
8 *
9 * @author  levenick
10 */
11public class Test {
12
13    /** Creates a new instance of Test */
14    public Test() {
15    }
16
17    /**
18     * @param args the command line arguments
19     */
20    public static void main(String[] args) {
21         //TODO code application logic here
22    }
23}
24
```

Simplest Application created with Netbeans
What Netbeans writes for you when you create a Java Main class Application

3. Replace `//TODO code application logic here` with `System.out.println("greetings");` in the main method (line 21).
4. Execute the program, (Project/Execute); if you made no typing mistakes you should see a window asking which is the Main Program; click the triangle to open the

directory, then select the only one, Test, dismiss that panel, and then greetings should appear in the output window (the output window is the window that says output in the title bar; the title bar is the bar at the top of a window).

Netbeans 3.6 Appendix B: Creating the simplest Applet in Net-beans

a) Start up NetBeans, (Step b) in NetbeansAppendix A on page 327)

b) Create a new project and mount a directory to use for it

This is Step c) in NetbeansAppendix A on page 327 -- be sure to do all six parts of that step!

c) Create a main program

1. Select File/new, open Java Classes (by clicking the triangle icon on its left), then select Applet, click Next, and type FirstApplet as the name (it will be highlighted, so just type), finally press Finish.

2. You should, in a moment, find yourself looking at a file that looks like:

Appendix Code Example 2

```
1/*
2 * FirstApplet.java
3 *
4 * Created on June 25, 2004, 1:11 PM
5 */
6
7/**
8 *
9 * @author  levenick
10 */
11public class FirstApplet extends java.applet.Applet {
12
13    /** Initialization method that will be called after the applet is
        loaded
14     *  into the browser.
15     */
16    public void init() {
17        // TODO start asynchronous download of heavy resources
18    }
19
20    // TODO overwrite start(), stop() and destroy() methods
21}
22
```

Simplest Applet created with Netbeans
What Netbeans writes for you when you create a non-GUI Applet

3. Replace line 17 with `System.out.println("I wrote an Applet!");`.
4. Execute the program, (Project/Execute); if you made no typing mistakes you should see a window asking which is the Main Program; click the triangle to open the directory, then select the only one, FirstApplet, close that panel, and then "I wrote an Applet!" should appear in the output window (the output window is the window that says output in the title bar; the title bar is the bar at the top of a window).

Netbeans 3.6 Appendix C: Creating a GUI Applet

a) Create a new project and mount a directory to use for it

See NetbeansAppendix A on page 327

b) Create a GUI Applet

1. Select File/new, open Java GUI Forms (by clicking the triangle icon on its left), then open AWT Forms (the same way), then select Applet Form, click Next.
2. Give it any name you want (so long as it is a legal identifier and starts with a capital letter!), for simplicity, use ATM_Applet.

Two windows should open, a Form Editor, and a Source Editor. You will be working with the Form Editor first, and the Source Editor second.

c) Set the Layout to Null

1. In the Form Editor, Open the [Applet] on the Inspector pane (on the right, second pane from the top) by clicking the triangle.
2. right-click BorderLayout, select SetLayout, then select Null Layout.

Continue with the next Appendix to add a Button.

Netbeans 3.6 Appendix D: Adding, connecting and testing a Button

Continuing from the previous Appendix, or, anytime you have the Form Editor open (click the ATM_Applet [form] button in the top bar to open it) and the Layout set to Null Layout already (you can check by opening the [Applet] in the Inspector ("right-click BorderLayout, select SetLayout, then select Null Layout." on page 331).

a) Add an AWT Button

1. Select (click) the AWT tab in the Palette pane (on the right, at the top). Click on the Button icon (top row, second from left, with the OK on it, just right of the big A)
2. Click in the Applet pane (the colored rectangle). A Button should appear where you clicked, with the label button1.

b) Connect it to your program

Double-click the button. You will be taken to the Source Editor, with the cursor positioned in the `private void button1ActionPerformed(...)` method, i.e. line 25 in

Code Example 3 (which was copied from Netbeans and edited to fit on the page.) Notice

Appendix Code Example 3

```
public class ATM_Applet extends java.applet.Applet {

    /** Initializes the applet ATM_Applet */
    public void init() {
        initComponents();
    }

    private void initComponents() {
        button1 = new java.awt.Button();

         setLayout(null);

         button1.setLabel("button1");
         button1.addActionListener(new java.awt.event.ActionListener() {
             public void actionPerformed(java.awt.event.ActionEvent evt) {
                 button1ActionPerformed(evt);
             }
         });

         add(button1);
         button1.setBounds(100, 90, 66, 24);
     }

     private void button1ActionPerformed(java.awt.event.ActionEvent evt) {
         // TODO add your handling code here:
     }

     // Variables declaration - do not modify
     private java.awt.Button button1;
     // End of variables declaration

}
```

Applet GUI with a single Button -- first pass
What Netbeans writes for you when you add a button and double-click it in the Form Editor

that that line reads

```
    // TODO add your handling code here:
```

Anything on a line after a // is a `comment` and is only visible to the programmer, the compiler ignores it.

Because you double-clicked the Button in the Form Editor, Netbeans wrote code that will be executed when you push the Button (when the Applet is running). In particular, the `private void button1ActionPerformed(...)` method will be executed each time the button is pushed. To test that that actually happens, replace `// TODO add your handling code here:`
with `System.out.println("button was pushed");` or some similar message.

c) Test it

As usual, to compile and execute your program, Project/Execute. If it is the first time, you will be asked to specify the main program (Code Example 4 on page 328). Push the Button and it should display the text from the println in the output window. Push the Button several times. Assuming you see the message repeated each time you push the Button, it’s working!

Netbeans 3.6 Appendix E: Creating a GUI Application

This is almost identical with creating a GUI Applet. The only difference is that you must choose Frame Form instead of Applet Form, and you must add a `setBounds()` messages in `init()`. Once that's done, you can use the Form Editor to add components.

a) Create a new project and mount a directory to use for it

See NetbeansAppendix A on page 327

b) Create a GUI Application

1. Select File/new, open Java GUI Forms (by clicking the triangle icon on its left), then open AWT Forms (the same way), then select Frame Form, click Next.
2. Give it any name you want (so long as it is a legal identifier and starts with a capital letter!), for simplicity, use ATM_Applet.

Two windows should open, a Form Editor, and a Source Editor. You will be working with the Form Editor first and the Source Editor second.

c) Set the Layout to Null

1. In the Form Editor, Open the [Applet] on the Inspector pane (on the right, second pane from the top) by clicking the triangle.
2. right-click BorderLayout, select SetLayout, then select Null Layout.

d) Making the Frame appear

Unless you add a `setBounds()` method in the constructor, you will never see the Frame! It has size zero by default and Java does not display things of size zero. Code Example 4 shows how to fix this; the four parameters are the usual for a rectangle, (x, y, width, ht).

Appendix Code Example 4

```
1 public class EditFrame extends java.awt.Frame {
2
3     /** Creates new form EditFrame */
4     public EditFrame(Account theAccount) {
5         initComponents();
6         setBounds(100,100,300,300);
7     }
8
```

Making the Frame not be size zero

Line 6: NetBeans does not write this line for you and if you forget it, you will *never* see your Frame.

Netbeans 3.6 Appendix F: Adding a pop-up Frame

Sometimes you want a frame to display something, or interact with the user, but you only want it to appear when necessary. This can be done using a Frame Form, just like in "Creating a GUI Application" on page 334, with the following changes.

1. Delete the `main()` method (this is not an Application and no one will ever send it a `main()` message.
2. Add `show()` to the constructor (or you'll never see it).
3. Change `System.exit(1);` to `setVisible(false);` in `exitForm()` see Code Example 5

Appendix Code Example 5

```
1 public class EditFrame extends java.awt.Frame {
2
3     /** Creates new form EditFrame */
4     public EditFrame(Account theAccount) {
5         initComponents();
6         setBounds(100,100,300,300);
7         show();
8     }
9
10    ...
11
12    /** Exit the Application */
13    private void exitForm(java.awt.event.WindowEvent evt) {
14        setVisible(false);
15    }
16
```

Showing the Frame, and avoiding ending the program when it closes.

Line 7: Until you send `show()` to a Component, it is invisible. Again, if you forget this, you will *never* see your Frame.

Line 14: `exitForm()` is sent when the user closes the Frame; Netbeans writes it to send `System.exit()`, which exits the program. This way the frame is simply rendered invisible.

Netbeans 3.6 Appendix G: Adding, connecting and testing a TextField

This is just like adding the Button (see previous Appendix). Open the Form editor (you can do this by double-clicking the icon to the left of the Applet in the FileSystems pane). Make sure the AWT tab is selected in the Palette pane. Click the TextField icon (just to the right of the Button icon), then click in the Applet rectangle (as the upper left corner of where you want it to appear (you can move it by clicking and dragging it). A TextField should appear. Double-click it. You will be switched to the Source Editor, ready to add code that will be executed when the user hits enter in the Textfield.

Appendix Code Example 6

```
1 private void textField1ActionPerformed(java.awt.event.ActionEvent evt) {
2     // TODO add your handling code here:
3 }
4
5 private void button1ActionPerformed(java.awt.event.ActionEvent evt) {
6     System.out.println("they pushed the button!!");
7}
```

Applet fragment with a Button and a TextField
What Netbeans writes for you after you add a TextField and double-click it in the Form Editor

Replace the comment at line 2 with the code:
`System.out.println("TextField contains: " + textField1.getText());`
Execute the Applet; try typing in the TextField and then hitting enter.

Netbeans 3.6 Appendix H: Adding a TextArea

Adding a TextArea is exactly like adding any other Component. Click the TextArea icon in the AWT pane of Palette window in the Form Editor view of the Applet. Then click where you want it, resize it and rename it. If you always call your TextAreas the same thing, like theTA, you won’t have to remember what they are called later when you want to access them.

Netbeans 3.6 Appendix I: Adding and using a Choice

Like other AWT Components, choose the AWT pane of Palette window in the Form Editor view, click on the Choice, then in the Form Editor (yes, choose Null Layout to get control of the size). Now:

1. Change its name (maybe to theChoice).
2. Select Events in the Properties pane (theChoice must be selected for this to work)
3. Select ItemStateChanged and click the ellipses box - a Handlers pane will open, click "Add..." in it.
4. Type a name for the method that will be invoked when the user selects a new item in the Choice; `newItemSelected()` is my first thought. Hit enter, then "okay". Netbeans will write the code and flip you to it.
5. Put some items in the Choice’s list of choices, right after`initComponents()` in the constructor. Like this:
```
        public BankDBMS() {
            initComponents();
            theChoice.addItem("this");
            theChoice.addItem("that");
            theChoice.addItem("the other thing");
```
6. To test the Choice write the following code in `newItemSelected()` --
```
            String item = theChoice.getSelectedItem();
            System.out.println("new choice from the choice: " + item);
```
7. Execute the program to make sure it’s working. Then the item variable can be used as a parameter to whatever method you want `newItemSelected()` to invoke.

Netbeans 3.6 Appendix J: Changing the label on a Button

The default label on Buttons is button1, button2, etc. To change the label of a Button:

1. Click on it.
2. In the Properties pane (on the right, under the Inspector), select (double-click) button1 next to Label (sixth line down).
3. Type the new label myFirstButton, or whatever (hit enter -- if you do not hit enter, *nothing will happen!*).
4. Resize the button so you can read the whole label (click and drag its border).

Netbeans 3.6 Appendix K: Renaming Components

When you create Components, Netbeans gives them names like button1, button2, and etc. To change the name of a Button

1. click on it
2. In the Inspector pane (on the right) its name should be highlighted, click on that, once, and wait a beat for it to select the current name
3. When button1 (or whatever) is selected; type the name you wish to give it instead. Hint: use the class of the component in its name, like testButton. Again, *HIT ENTER*, or you have not renamed it.

Netbeans 3.6 Appendix L: Creating a class

1. First, have the project you are working on open.
2. Create a new file (File/Add New)
3. Open Java Classes (click the triangle on its left), select Java class, click Next
4. Type the name of the class and hit Enter

Netbeans 3.6 Appendix M: Creating a class with a test driver

1. First, have the project you are working on open.
2. Create a new file File/Add New
3. Open Java Classes (click the triangle on its left), select Main, click Next
4. Type the name of the new class, Account or whatever, click Finish. It should create the class and display it, like this:.

Appendix Code Example 7

```
/*
 * Account.java
 *
 * Created on June 25, 2004, 2:13 PM
 */

/**
 *
 * @author  levenick
 */
public class Account {

    /** Creates a new instance of Account */
    public Account() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

An Account class shell with a main method

Netbeans 3.6 Appendix N: Changing the size of an Applet

1. Assuming your Applet class is called TestApplet; Open the TestApplet.html file (in the FileSystems window it is right under the other TestApplet with a different icon).
2. Change the 300 in Code Example 8 line 10 to change the height of the Applet (don't

Appendix Code Example 8

```
1  <HTML>
2  <HEAD>
3  <TITLE>Applet HTML Page</TITLE>
4  </HEAD>
5  <BODY>
6
7  <H3><HR WIDTH="100%">Applet HTML Page<HR WIDTH="100%"></H3>
8
9  <P>
10 <APPLET code="TestApplet.class" width=350 height=300></APPLET>
11 </P>
12
13 <HR WIDTH="100%"><FONT SIZE=-1><I>Generated by NetBeans IDE</I></FONT>
14 </BODY>
15 </HTML>
```

TestApplet.html

This is the HTML code NetBeans generates automatically when you make an Applet called TestApplet.

Line 10: This is the line you could put in any HTML file to start TestApplet (assuming TestApplet.class was in the same directory as the HTML file). To change the size of the Applet change the values of width and/or height.

forget to *save* it - File/Save, or ^s).

Netbeans 3.6 Appendix O: Using the Color Editor

In the Form Editor, select a button, then in Properties click the ... to the right of background, click RGB and slide the sliders

Netbeans 3.6 Appendix P: Catching mouse clicks

You can catch mouse click in a Frame as follows. Assuming you have a GUI Frame Form (an Applet works the same way), in the Form Editor:

1. Select the Frame
2. Select Events in the Properties pane
3. Select mousePressed and click the ellipses box - a Handlers pane will open, click "Add..." in it.
4. Type a name for the method that will be invoked when the user presses the mouse in the Frame; `mousePressed()` is descriptive. Hit enter, then "okay". Netbeans will write the code and flip you to it.
5. Write diagnostic code like Code Example 9 so that you can be sure that it is catching

Appendix Code Example 9

```
private void mousePressed(java.awt.event.MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    System.out.println("Pressed! x=" + x + " y=" + y);
}
```

TestApplet.html

This is the HTML code NetBeans generates automatically when you make an Applet called TestApplet.

Line 10: This is the line you could put in any HTML file to start TestApplet (assuming TestApplet.class was in the same directory as the HTML file). To change the size of the Applet change the values of width and/or height.

the mouse presses correctly.

Once you are sure you are getting the correct coordinates when the mouse goes down, you can pass x, and y to some other method to do whatever you wanted to do with that information.

Note that this code will execute when the mouse goes down. You can also handle the event when it goes up (mouseReleased), or when it goes down and up without moving (mouseClicked), or when it is moved while down (mouseDragged).

Netbeans 3.6 Appendix Q: Adding JavaDoc comments.

You can add comments to a class from NetBeans by Tools/Auto Comment. This will guide you through commenting your code. Then, after you have added the comments you want to all your classes, use Tools/Generate Javadoc to create all the files. Try it out. It’s fun! It’s easy! It looks really impressive!

When you have a file open and select Tools/Auto Comment, NetBeans opens an Auto Comment Tool window. All the methods in the current class are listed on the left, each with an icon indicating it’s status relative to JavaDoc comments. If there is no comment, a red X is displaced, if there are missing tags for elements the tool can detect, a lightning bold is displayed. This is the "autocommenting" the tool does; you, the human, must fill in the comments to make all the methods display the green check icon.

Each method needs a comment which tells what it does. This is entered in the "Java Comment Text" box at the top. Additionally, methods which have non-void return types must have an @return tag which describes the return value, and methods with parameters require @param tags which describe each parameter.

Notice the four buttons to the right of the list of methods. Autocorrect will insert the tags it can tell you need; click on each one to add its description. Best of all, push the Help button for a much more thorough description than is provided here.

access modifiers - the keywords public, private, and protected control which methods and variable can be accessed from where. If a method has no access modifier, it is "friendly" -- i.e. public to you, private to anyone else. That s okay to do for now. 28

actual parameters - parameters in the parentheses of a message (may be any expression of the appropriate type) 104

Applet - a Java program that runs in the context of a web browser. Also a class in java.applet. 20

Application - A java program that runs independently. 21

assignment operator - a single equals sign 95

BNF - Backus Naur Form. A metalanguage for describing context free grammars. Commonly used to describe the syntax of a programming language. 92

bug - An error in a program. 19

byte code - The intermediate form that the Java compiler puts into the .class file to be interpreted by the Java Virtual Machine when the program executes. 25

compile-time - during compilation, compare with execute-time 126

Component - a generic class in java.awt that includes Button, TextField and other common GUI components. 33

concatenation - to attach together end to end. If you concatenate "psycho", "the", and "rapist", you get "psychotherapist". 29

control variable - a variable that controls the execution of an iterative loop 192

default value - the value you get if you don t do anything. Instance variables are assigned zero by default when they are created. If you want to initialize them to something else you may (e.g. int x=17;). 60

echo a file - to read a file and display it on the screen 247

file I/O - file input and output 239

formal parameters - parameters defined in a method heading (each must have a type and a name) 104

GUI - graphical user interface 39

hand simulation - simulating code by hand; performing the semantics of each statement, one by one to discover how code works, or when it is broken, why it doesn t 192

HTML - hypertext mark-up language: an embedded command formatting language commonly used for web pages. 67

identifiers - Java names. Must start with a letter and be composed of only letters, digits and underscores; case matters. 24

idiom - a sequence of symbols whose meaning cannot be derived from the individual symbols, but which must be learned as a whole, by rote 23

infinite loop - a loop that executes forever 192

instance variable - a variable declared outside of any method, a copy is created for each instance 100

invariant - a condition that does not change; something which is known to always be true in some part of an algorithm 314

iteration - another word for repetition. See the next chapter for iterative constructs. 225

Java Virtual Machine - Software that creates the Java runtime environment on a particular machine. 21

main method - where execution begins when your Application runs 21

Math.random() - returns a random double in the range [0,1) 123

Package - A package is a collection of related classes and interfaces. You can import packages that other people have written into your program 21

parameter - information sent along with a message which invokes a method 28

parameter linkage - when a message is sent with parameter, the values of the actual parameters are copied to the corresponding formal parameters 106

pixels - picture elements, the smallest drawable part of the output 67

precedence - in an expression with multiple operators, which operator precedes which (* precedes +) 121

problem solving -- The behavior one engages in when one is stuck and doesn t know what to do next. 18

prototype - a simplified, preliminary version of something, in this case, a program. 26

public - An access type; means anyone can see this. 25

recursive definition -- a definition that uses the thing being defined 41

scope - the portion of a program where a construct is visible (or defined) 126

Semantics - meaning, or action. 92

shadow - to hide a variable, by being named the same thing in a more local scope; most often happens with parameters 116

signature - the type, name of a method along with the number of parameter and their types 66

Socket - A mechanism to connect one computer to another (virtually). Also a class in java.network. 21

spawned - technical term for created; used only for Threads. When a new thread of control is initiated, i.e. begins execution, it is said to be spawned. 218

static - modifier that creates class variables or methods instead of instance variables or methods 127

String - the Java class whose instances are each a literal series of characters. 29

Syntax - grammar, or form. 92

syntax - Grammar. Every programming construct has both syntax (grammar) and semantics (meaning). 21

thread of control - the sequence of statements executed when a program executes. A temporal map of where control resides during execution. 216

toggle - a two state switch which changes state each time you activate it 234

variable - memory to store one value of a particular type 43