An Introduction to Programming in Java

Last revised: Tue Jan 15 23:31:47 2002

1. Preface

To build an object-oriented program is to construct a simulation. This is the singular, guiding concept of this text.

Goals

The fundamental concepts of programming are few; the hours of practice required to become a competent programmer are many. This text attempts to elucidate that most basic set of concepts required to start practicing the craft of programming. We intend to teach the students a world view. We wish to show them a way of seeing that will enable them to effectively and efficiently build simulations of the world they are trying to model. To this end, we want them to become comfortable using existing abstractions, and designing simple ones of their own. While this book *uses* the Java programming language as a vehicle, it is not about learning the details of that language. Rather, it hopes to teach a deeper language -- a language for problem solving, design, and systems understanding.

Method

We believe that the world view that we are trying to teach is already tacitly understood by many, if not all students. We hold this belief, because we find that the object-oriented paradigm is deeply rooted in our cultural traditions. The dominant trunk of the Western intellectual tradition holds that we inhabit a world of objects. These objects have qualities and are composed of other objects. We may efficiently classify groups of objects based on their qualities. Furthermore, we can hierarchicalize our classifications by abstracting the shared qualities of related sub-classifications.

Whether or not we agree with this particular ontology and epistemology, we cannot deny that it represents the dominant manner by which computer scientists in particular, and Western society in general, views the world, organizes systems of information, understands nature, and so on. Because it is so deeply ingrained in our cultural tradition, we feel that the instructor's job is to *show* their students that they are already quite practiced at using the object-oriented world view. In other words, most students are already quite good at thinking about the world in an object-oriented way, they just don't know it. To this end, we appeal to the student's intuitions about how they understand and talk about the world.

We believe in stimulating not just the symbolic - but also the enactive and iconic - reasoning abilities of the student. Like crafts such as carpentry, we find that the activity of programming is highly enactive and pattern-driven. To this end, we encourage a learning-by-doing approach, immersed in rich example domains and supported by powerful patterns. We engage the student's iconic reasoning abilities by encouraging them to draw pictures. Drawing pictures of the relationships between things is to object-oriented programming what procedural decomposition and flow-charting was to imperative programming. It is the meat and potatoes of understanding, designing, and implementing a system.

Finally, we avoid showing students the wrong way to solve a problem, to the extent possible. Experience has shown us that beginning programmers are easily *imprinted* with less-than-ideal approaches to problem solving and design. We prefer students to learn things the right way, rather than unlearn things learned the wrong way.

Differences

This book differs from most other texts in various ways. First and foremost, it is short. Keeping it short means that our words weigh more; it is easier to emphasize a point when it is not lost in a sea of text. A short book is also a light book; a student is more likely to bring it on the bus, take it to bed, or read it over coffee.

Second, the book emphasizes a data-driven approach to program design rather than the control-driven one that has traditionally dominated introductory texts. The latter approach was and is appropriate when the teaching languages are fundamentally imperative, such as Algol, Ada, Pascal, or C. However, large-scale object oriented systems, and indeed, object-oriented analysis and design shift the focus to understanding and designing the relationships amongst objects in an approriate manner. Consequently, the emphasis on algorithms and their implementation is necessarily diminished. Instructors trained in a technology of a prior generation (and this includes the authors) must accept that algorithmic thinking is no longer the only valid approach to problem solving and implementation. In fact, it could be argued, that such a paradigm is in fact at odds with generating good object-oriented designs.

Third, the book is not detail, rule, or syntax obsessed. It does not attempt to be a language reference for the Java language. There is a place for rules and details, but a concept-driven introductory text is not one of them.

Fourth, the book focuses minimally on graphics and GUI programming. Somehow these language features seem to have entered the canon of introductory texts, especially those that use Java as their teaching language. While we agree that there are (a few) important concepts in this kind of programming, we do not feel that they belong in the core of an introductory text.

Finally, the text is largely decoupled from specific, nonstandard Java classes. While many texts are written in support or in tandem with a set of Java classes used to simplify common programming tasks such as graphics programming or input/output, we have made an effort to stay within the realm of standard Java classes to the extent possible. We wish to decouple the principles and concepts from particular implementations thereof. Course-specific classes are certainly a good idea in most settings, but their details are best left to an appendix or lab manual.

What's missing

The lessons at the tail end of the book have yet to be written (including File I/O and possibly Inheritance), or are in rough form (as is the case with the section on Exceptions). Eventually, a lesson on Interfaces will be developed, and probably introduced near the middle of the book. We've found the concept of *interface* to be a critical one (especially if we avoid introducing/using inheritance early in the course), and will ultimately need to dedicate a lesson to it. Finally, a lesson on programming tools, including editors, compilers, and debuggers needs to be added. The details or suggestions regarding the use of specific tools (eg. BlueJ) would be best left to an appendix.

---

2. Introduction

This primer strives to teach the reader a *way of being*. The philosopher Ludwig Wittgenstein described a world consisting of many languages. These languages - called *language games* - differ markedly from the traditional conception of languages, which views them merely as a means of transmitting information. Wittgenstein saw languages primarily as a way of getting things done. He said that over the course of our lives we become masters of many language games. There are language games for bankers, carpenters, supermarkets, classrooms, doctor's offices, the family, and so on. Each of these languages has it's own rules and is useful for surviving and being successful in a special context: at work, at play, at love, and so on. To play a language game successfully is to *understand*. Put another way, to understand is to do the right thing under the rules of the game that is being played. And to play successfully implies that one knows much more than just the rules (grammar) and words of a given language, it means literally to express a certain *way of being*. He said:

> "To imagine a language is to imagine a form of life."

Inspired by this quote we begin our journey in which we attempt to learn the language game of computer programming, and therein a new way of seeing the world -- a new way of being.

Programming requires learning a handful of fundamental concepts. These concepts can be expressed clearly, we believe, in short form. Many texts manage to make programming seem incredibly complicated. On some level, they are certainly correct. Becoming fluent in real programming languages such as Java, C++, or Smalltalk means becoming a master of many rules, details, and vocabulary words. However, at their core, all programming languages share just a few fundamental principles. Helping the student to learn and to deeply understand these principles should be the ultimate job of any text. We attempt to expose them with a minimum of clutter and distraction.

Furthermore, we believe that learning to program is done only in small parts by reading textbooks. Learning to program well perhaps shares more in common with learning to be a good writer or a good carpenter. Contrary to popular opinion, programming is more craft than science. This standard view of programming perhaps explains the mismatch between the typical programming text and the reality of programming. As computer scientists, we attempt to explain programming the way we might explain other concepts in the field. It is as if a linguist were to attempt to teach a child a language using the language of a linguist. The child, not versed in the linguist's language game, would have no common ground for communicating with her teacher and hence would hear only non-sense.

Becoming a skilled programmer takes years of experience. Teaching programmers would perhaps be better done in a master-apprentice oriented approach similar to that used in other skilled crafts. In a short introductory course, however, we can at best hope to demonstrate and encourage the reader to practice the fundamental skills and techniques, identify the essential patterns, and inspire them to adopt a critical, inquisitive, experimental approach to learning that will assist the student in their future studies as well as life in general.

By viewing programming as craft, we are also inspired to teach by example, appeal to intuition, and to reinforce lessons through practice. Hence, this book suggests spending a lot more time doing than reading. Learning-by-doing involves more parts of the brain and the body in the learning process. Programming - as a skill - involves not only strong symbolic reasoning abilities, but a powerful imagination, good design intuition, deep experience, and even an eye for beauty. These latter qualities cannot simply be explained in a textbook. They must be experienced, witnessed, and lived before they can become part of the makeup of any skilled programmer.

Programming does not exist in isolation from the rest of the world. Programming is a profoundly social activity. Only through an understanding the social aspects related to building large-scale software systems are we be able to build truly great, usable software. Programming technologies, such as the languages we use, are deeply rooted in our cultural tradition. By understanding the underlying history, philosophy, and ideology of the technologies that we use, develop, and deploy we can become more sensitive to their cultural impact. It is never too early for a technologist to begin learning about these kinds of issues, which have been traditionally relegated at worst to other, more marginalized departments on the university campus, or at best to the dusty corners of the course syllabus. This work attempts to integrate the social and cultural context of our technologies into the material it presents.

This book is organized in a few dozen short "lessons." Each lesson seeks to demonstrate, teach, or explain a key point or concept of programming. Each lesson, like a short story, attempts to digestible in a single sitting. The lessons do not attempt to explain language features, syntax, rules, and so on in all of their gory detail. No lesson should be read without doing the associated exercises, or experimenting with the presented examples. We guarantee that you will learn next to nothing if you only read this book. Only by typing in, experimenting, imagining, drawing pictures, and learning-by-doing will you get value out of this text. Good luck.

---

3. Programming as Communication

Key concepts

1. Programming as communication
2. Programs as communication artifacts
3. The difference between "reading" and "understanding"
4. The three world views: functional, operational, object-oriented simulation

Story

Programming is all about communication. First and most obviously, it is a precise way of communicating with a machine. To write a program is to tell a machine what to do. Second - and just as importantly - it is a way of communicating with other human beings. Why is the human communication element so important? First, very few programs are ever completely correct. That is, once they are put into production, errors are discovered. The program may not behave as expected. The person who fixes an existing piece of code may or may not be the person who originally wrote it. Second, programming projects, especially large ones, will suffer turnover of employees over the years. As new programmers come on board, they will inherit code that is years or even decades old, written by programmers who have long since moved on to other jobs. The new programmers will be charged with the task of reading and understanding this text. Finally, software evolves over time. The problem that a particular piece of software set out to solve a few years ago might be different today. For this reason, existing pieces of code often need to be modified to meet the new requirements of a different era.

Humans are incredibly good at "understanding" language, but very bad at "reading". For example, most people would have trouble reading the following:

I s ou r C hi l d r en le a r n i n g?

However, once you stare at the text long enough to realize that it asks, "Is our children learning?" you have no difficulty in understanding its meaning. Interestingly, despite the fact that it contains a grammar error, you still understand its meaning. Computers, on the other hand, are very good at reading. A machine would have no difficulty in picking out the sequence of letters in the above question, no matter how poorly they might be formatted. However, computers are extremely sensitive to grammatical errors, and would have great difficulty "understanding" the meaning of the above question. To communicate with a machine, we need to be incredibly precise in "meaning", but can be messy in our "writing." However, since humans will read the code as well, we need to write clearly too.

We'll rely heavily on three important metaphors when interpreting programs: a program as a precise set of instructions, programs as executable math, programs as simulation.

Operational world view

Traditionally, language has been viewed merely as a means of transferring information between two or more communicating parties. While language is certainly employed to transfer information, it has other uses too. Another view considers language as a means of effecting actions. Think about the following statement:

> "Get me an gallon of milk (please)."

This statement brings about a whole chain of events, including causing you to get up, walk to the store, purchase a gallon of milk, and so on.

This action-oriented view of language is very applicable to computer programs. Programs are not merely about transferring information, rather, they command, instruct, cause things to happen. The program is a sequence of utterances that cause the computer to undertake one action or another.

Perhaps the best analogy is that of giving directions. Directions provide you with a sequence of steps, which, if you follow them, will get you to your destination. We call such a set of instructions an *algorithm*. Of course, the directions have to have a sufficient level of detail and precision to be meaningful to you. The sort of precision and logical thinking required to give a good set of instructions are skills you will cultivate over the course of your study.

Consider giving directions to the campus post office to someone familiar with the campus geography, such as a veteran student. Compare this to giving directions to a tourist. To the student, you might say:

> "Walk just past the Union building and you'll see it on your right."

These instructions are of little use to the tourist however, because they assume a certain knowledge of the environment. For the tourist, directions such as the following are more appropriate:

> "Walk under that arch and make a right. After about two hundred yards you'll see a red brick building on your left. That's the Union building. After you pass the Union, you'll see an ugly concrete building on your right, which is the post office."

The student understands a more expressive language for direction-giving than does the tourist. Computer scientists would say that the student is operating at a higher *level of abstraction* than the tourist. *Abstraction*, while it sounds complicated, just means focusing on the *what* rather than the *how*. It is a way of thinking about something that disregards the details of that things internal makeup or function.

When talking to the student, you can leave out many details, which you would have to perhaps explain to the tourist. Programming languages also allow us to express concepts at a variety of levels of abstraction. In fact, a large part of writing great programs consists of either leveraging existing abstractions or implementing your own abstractions appropriate to the task at hand. By implementing your own abstractions, you are in a way introducing new words (and their meanings) into the language, thereby allowing you to express instructions as a higher level of abstraction. Programming languages themselves may differ in the fundamental abstractions they present to the programmer. Java, for instance, tends to provide the programmer with higher-level abstractions than does, say, C.

Functional world view

Another powerful way to think about programs or pieces of programs is as a set of mathematical expressions that are evaluated to generate or calculate some sort of interesting result. For instance the following equation:

Area = PI x Radius^2

expresses the area of a circle given its radius. We will see shortly that we can employ such expressions in computer programs. Almost all of our intuitions and experiences with mathematics can be applied to computer programs.

Object-oriented simulation world view

This view holds that programs are simulations of our world. They "model" the world around us. Hence, just like our world, they consist of animated *objects* that know things about themselves, can answer questions, and perform actions for us. Furthermore, we can construct simulations of "imaginary" worlds too. For instance, many video games suspend or modify the laws of physics and create a world unlike our own. And while spreadsheets resemble a piece of gridded paper, they are active in ways no piece of real paper will ever be.

In a way, this ability to construct simulations of imaginary worlds is one of the great things about writing computer programs. In many situations, it is much easier to build (and tear down, and experiment with) a simulation than it is to build the real thing. Programs are only plastic to an extent, however. Eventually, a program itself may become as complicated and hard to manipulate as real-world entities, because of the huge number of interactions between various pieces of the system. Remodeling the program becomes no simpler no simpler and no less without consequence than say, razing whole neighborhoods to build freeways. Finally, large-scale software systems do not and can not exist independently of the people who design, produce, test, and maintain them. Gross changes to the program structure will inevitably impact the social structure which has been erected to support it.

---

4. A World of Objects

Key concepts

1. Objects: the things that inhabit our world
2. Naming objects
3. Creating objects
4. Sending messages to objects
5. The interpreter: our first tool

Introduction

Java provides us with a language that allows us to express an object-oriented view of the world. It allows us to build models or simulations of real-world - or even non-real-world - entities and environments. The fundamental unit of modeling is called the *object*.

What's an object? The answer is simple, really. Objects in Java are very similar to the things we think of as objects: chairs, tables, cars, people, students, files, desks, drawers, money, bank accounts, and so on. Many people argue that object-oriented programming is more difficult for the beginning programmer to master, because students must understand the supposedly difficult concepts of the "object." However, we take the opposite view. Thinking about objects in computer programs is as natural as thinking about objects in the real world. It is something that we as individuals have done since a very young age. Furthermore, thinking about the world as consisting of objects is deeply rooted in the Western cultural tradition. Many of the ideas first explored by Plato and Aristotle over two thousand years ago have manifested themselves today in the technology of object-oriented programming.

Objects often (usually) are made up of other objects. In fact, very few objects are *primitive* -- most objects consist of other objects. Humans have organs; organs have tissues; tissues have cells; cells have DNA, proteins, mitochondria, and so on. Offices have desks; desks have drawers; drawers can have contents inside of them. *Composition* of objects is a central theme that we'll revisit throughout this study.

Let's take a detour and enrich our understanding of language by looking at how we often use it to describe the world around us. While this discussion may seem as if it couldn't be further from learning how to program a computer, it turns out to provide the fundamental basis for nearly everything we'll talk about for the rest of our study. Why? Because building an object-oriented computer program is all about using a simple language (such as Java, which is much simpler and less powerful than our own, natural language) to define a system of things and their relationships. We shall see that it is not at all radically different from the way we use language every day.

Things/Objects

> Imagine that I have a pet cat, named "Henry". Henry is a real, physical being, with certain qualities, such as age, weight, size, and so on. Henry is composed of other things, such as muscles, organs, and blood. Henry is also an active entity: he can consume food,

make sounds, and walk about. My neighbor may also have a pet cat, named "Natasha". Natasha and Henry are both cats, but they are distinct entities. It seems somehow degrading to say that both Henry and Natasha are objects, but we can view them as such.

Names and Meaning

The name "Henry", at least in the context of my household, refers to the being or thing that inhabits my home, that walks on four legs, meows when it is hungry and so on. We say that the *meaning* of the word "Henry" is the object that it refers to, namely that being which inhabits my home. The word or name "Natasha" refers, on the other hand, to a similar being that inhabits my neighbor's home. The meaning of a word is the object it refers to. In language, we may have multiple words that refer to the same object. For instance, the words "George" and "President" are words that refer to the same object -- the president of the United States at the time of this writing.

Concepts

Now let's think about the word "cat" for a moment. What is the meaning of that word? If our view of meaning is that the meaning of a word is the thing that it refers to, then we must ask, What does the word "cat" refer to? It certainly doesn't refer (when used in the general sense) to any particular cat. Instead, it seems to refer to the set of qualities that all cats have in common. It refers to the collection of knowlege that we have about Natasha, Henry, and all other cats. The word "cat" refers to the *concept* of cat, rather than any particular cat. A concept is a definition or a description of a set of objects. Concepts exist only in our minds -- they do not have a real, physical presence.

Must objects be physical things?

At first glance, it appears that all Objects (as opposed to Concepts) have real, physical qualities. Think about chairs, cars, tables, people. These are all objects that inhabit our world, and they are all physical things. Does this statement hold true for all objects? To answer this question, think about the number seven. Is it an object, concept, or something else altogether? Ask yourself what does the word "seven" mean? What sort of thing does it refer to? It refers to a property, a quality and nothing more. It refers to a property that is shared by Snow White's Dwarves, the days of the week, and the Deadly Sins. The number seven refers to an object as surely as the name "Henry" refers to the object that is my cat. It just so happens, however, that this particular object is a pure abstraction. It exists only in our minds. Seven is a quantity, a number. It is a particular instantiation of the concept "Number" or "Quantity". Put another way, the relationship between "Henry" and "Cat" is exactly the same as that between "Seven" and "Number". Henry and Seven are names that we have given to particular instances of a given concept. The important implication is that objects do not necessarily have to be concrete, physical things. While it is easiest for us to conceive of objects that have real, physical properities we must accept that there are objects that exist only in our minds, such as numbers.

Are Concepts Objects?

The answer to this question must be yes, because concepts can be named, talked about, described, and manipulated in (almost) the same way that real, physical objects can. Of course, we can't measure how much the concept "Cat" weighs, or see it's color, but you and I can have a conversation about cats (in general), and we can refine our understanding of them (perhaps by distinguishing between house cats and wild cats). This ability to create, learn, and manipulate concepts in our minds is a faculty that makes us distinctly human.

It is true that there is a difference between concepts and objects -- concepts are special sorts of objects that have descriptive and generative powers. They describe a set of objects that all share a certain makeup and qualities. They can also be used to generate new objects. It's easiest to see this with respect to Numbers. Understanding the concept of Number allows us, for instance, to generate all kinds of numbers, like 7, 42, or 3.1415. These differences do not mean that concepts themselves are not objects.

A Brain Teaser

If concepts are objects, and all objects are instances of a particular concept, then what concept are concepts themselves instances of?

From Language to Java

Let's briefly review what we know about how we think about and talk about the world around us. Objects are the things that inhabit our world. They have qualities and often consist of other objects. A car, for example, consists of body parts, an engine, wheels, and so on. It also has qualities such as color. An engine has qualities such as size and power output. We may also think of Objects as being able to perform certain kinds of activities. I, for example, can write, speak, drive a car, take a shower, and so on. A dog, for example, can bark, eat, roll over, and sleep. All Objects are particular instances of some Concept. Concepts are special kinds of objects that have descriptive and generative power. They describe a set of objects that all share the same makeup; and, particularly in the case of abstract entities such as numbers, they can be used to generate new instances of themselves.

Java is a language which is far less descriptive, yet much more formal than our own language. That is, because we use Java to communicate with a machine, we must be totally explicit in what we say, and we must follow certain rules for how we say things. Java allows us, above all, to define Concepts, create Objects that are instances of those Concepts, and manipulate those Objects.

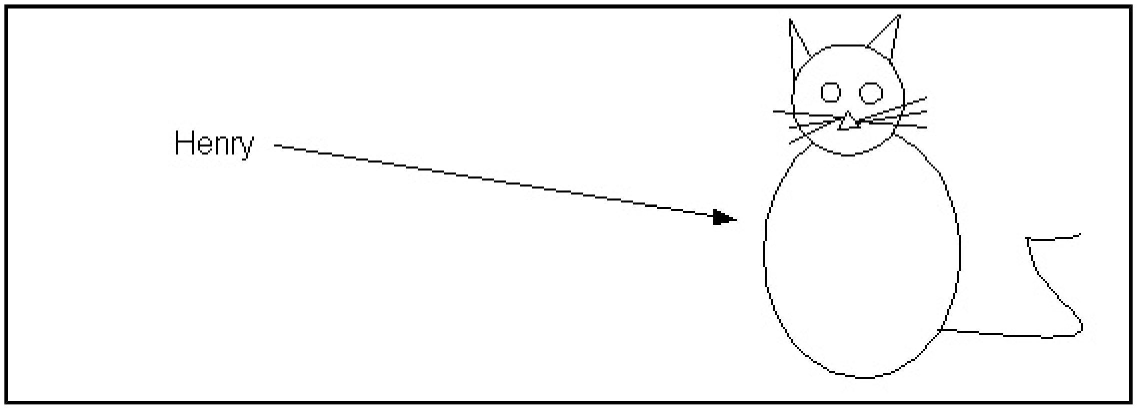
The terminology used in Java differs slightly from that which we have introduced above. The term *object* is identical; it is a thing. In Java, we use the term *class* instead of the word *concept* though. It's easy to remember this word if you remember that a concept describes a class (or set) of things. Our concept for "car" describes a set of things that are all cars. We will also use the word *type* as a synonym for *class*. There is actually a technical difference between the two terms, but for now we'll treat them as the same. Finally, just as we think of many real-world objects as being able to perform activities, Java objects can be "active" as well. In Java, if an object knows how to perform a certain activity (for instance, to drive a car) we say that it has a *method*. We can ask an object to perform an activity for us by *sending it a message*.

Note that in Java, *all* objects can have methods, that is, they may know how to perform certain actions for us. You should think of every object in Java as being "animated" or "alive". This ability to animate objects is what gives Java a special power: we can use it to define classes of objects, create objects of those classes, and finally ask those objects to do things for us. In Java everything is make-believe. We may animate everything. In animating all of our objects, we give them life. Part of being a good Java programmer is to imagine a world in which all objects are "smart", in which all objects can perform tasks on our behalf. Leaving behind the constraints of real-world limitations, and entering this make-believe world of animated objects allows us to create powerful systems of intercommunicating, interacting objects.

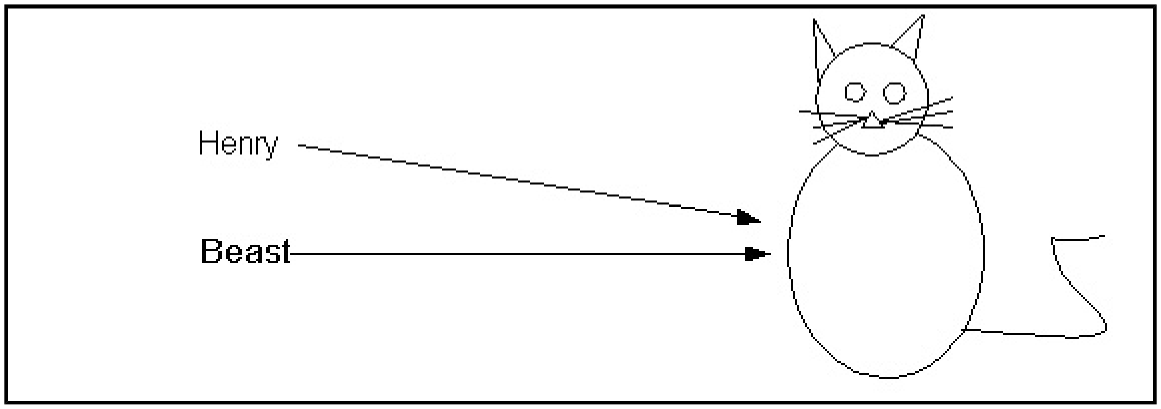
We will now proceed to re-introduce the fundamental concepts we've discussed above, but in the context of the Java language. At each step, it's important to think about the ideas we've discussed above. Think about how you consider objects, concepts, names, and so on in the real world, using real language. If you can do this, you will win most of the battle of becomming a good programmer. To be a good programmer is to know what you want to say. Once you know that, figuring out how to say it is almost mechanical. Many beginning programmers become obsessed or overwhelmed with how to say certain things in Java, when they haven't really even figured out what they want to be saying. You are already good at seeing the world in terms of objects, names, activities, and concepts. Leverage that skill as you step into a new way of expressing your ideas.

Naming Things

In our language, we've seen the use of words to *name* things. The word "Henry" names my cat. We say that the name "Henry" *refers* to the animal (a cat) who is my pet. In pictures, we might draw the relationship between the word "Henry" and the pet as follows:



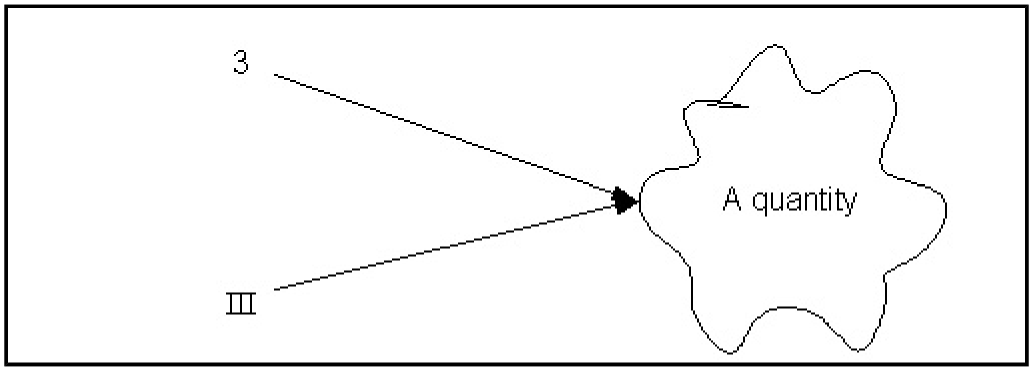
The *meaning* of a word is the thing that it refers to. It's OK to have two names that mean the same thing. For instance, I might also call my cat "Beast" sometimes, which would result in this picture:



Notice that in this particular context, we have two names, "Henry" and "Beast" that both refer to the same object, my cat.

Simple objects

Simple objects are those that do not consist of other objects. Because nearly every program manipulates numbers in some way, Java provides us with a few different kinds of numbers, which will be our first example of Java objects. You should think of numbers in Java the same way we think of numbers in real life. A number is a quantity. It is an abstract and immutable thing. For instance, the quantity of wheels on a bicycle is the same as the quantity of wheels on a motorcycle. We have a special name for that quantity: "two." It's easy to confuse the name (in this case "two") with the actual quantity. Over the course of history, we have devised numbering systems, which allow us to conveniently name quantities. Mostly we are familiar with decimal numerals, but there are many other systems for naming quantities, including Roman numerals, binary, hexadecimal, and so on. We might draw the situation as follows:

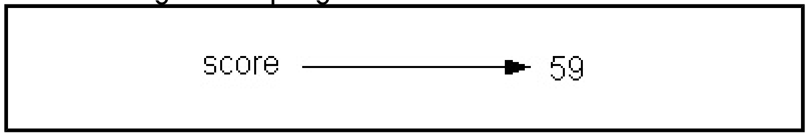


Notice that both the names indicate the same object, which is a quantity. Pretty much all of your intuitions and understanding of numbers apply equally in the Java world. In Java we may express numbers literally and give interesting names to them:

```
int    score = 59;
double todaysTemperature = 67.4;
```

We'll use two of the different kinds of numbers provided by the Java language throughout this text: we'll use an `int` when we want to represent integral values; we'll use a `double` to represent rational values.

Throughout this study, we'll draw pictures to help understand and express the relationships between the things in our programs. Let's draw one:



The above picture shows the name "score" as referencing or denoting (with the arrow) a thing, which in this case is a quantity. Above, we represented a quantity as a cloud (or a blob), to differentiate it from its name (in that example, "3" or "III"). We could have used a blob in this picture as well, but they're not terribly descriptive. So we'll just represent quantities by their common names -- in this case, 59. The way to read a picture like the above is: "The name 'score' refers to the quantity (or number) 59."

The above example also introduces us to our first pattern, which is a pattern for naming things. Sometimes, we'll also call this *binding* a name to a value.

```
<The kind of thing> <the name> = <the thing we're naming>;
```

Notice that Java is sensitive to case when we write names. In other words, the name `score` and `Score` are not the same name in Java. Furthermore, Java imposes some restrictions on the legal letters in our names, which we also call *identifiers* sometimes. The rule is essentially that names must begin with either a '_' or a lower or upper case 'a' through 'z' character. The rest of the letters can consist of lower and upper case 'a' through 'z', '0' through '9' and '_'.

Another useful primitive object is the `char`, which is used to represent individual characters, such as 'b'. Here are a few examples of characters in use:

```
char letterGrade = 'A';
char middleInitial = 'R';
char section = 'b';
```

Finally, Java also provides us with a kind of thing called `boolean`, which we can use to talk about the truth or falsehood, for instance:

```
boolean rainingInSeattle = true;
boolean rainingInLosAngeles = false;
```

A boolean can have just one of two values, true or false. We can name these values directly in Java, just like we can name integer values directly.

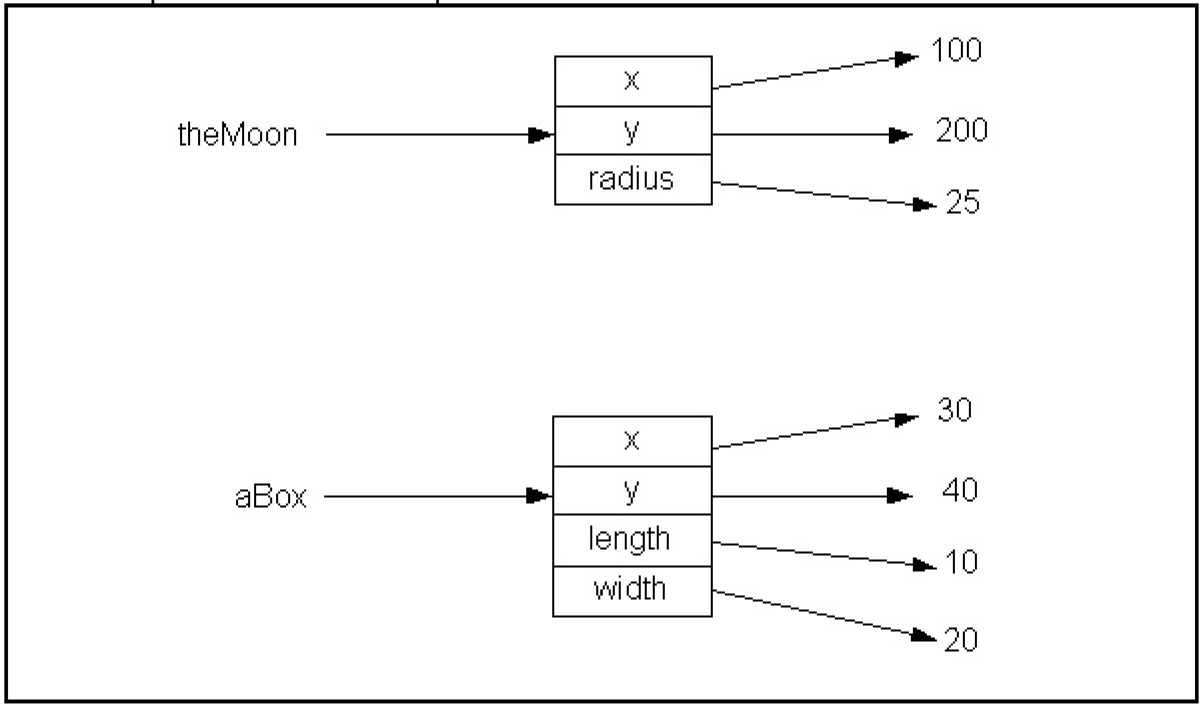
Some shapes.

Suppose we want to represent some shapes in Java. Let's imagine for the time being that someone has provided us with classes that we can use to represent shapes such as rectangles and circles. Suppose that the properties of a circle are the position of its center and a radius. Suppose that the properties of a rectangle are the position of its upper left hand corner and a width and a height. We might create and name some shapes as follows:

```
Circle theMoon = new Circle(100, 200, 25);
Rectangle aBox = new Rectangle(30, 40, 10, 20);
```

Note the special word "new" used above. This is the way we ask the machine to create a new instance of an object for us. Why don't we need to say `new` when we use numbers? In a way, we never need to create new numbers: imagine that the Java world is inhabited by numbers in much the same way as our own and we may just name them and use them at will.

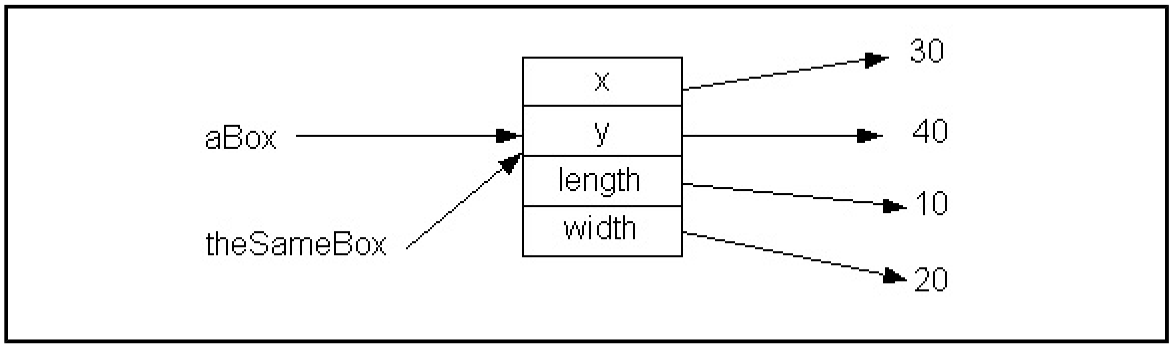
Lets draw some pictures of our new objects:



Let's look at naming in a little more detail. We say that a name *refers* to an object. For this reason, we can have multiple names that refer to the same object. For instance, the names "Mom" and "Ellen" may both refer to the same person, in this case, my mother. Let's look at a Java example:

```
Rectangle aBox = new Rectangle(30, 40, 10, 20);
Rectangle theSameBox = aBox;
```

Here's the resulting picture we might draw:



Finally, we can also send some *messages* to these objects:

```
int area = theMoon.area();
aBox.move(10, 15);
```

The first statement is asking the moon to compute its area, and giving the returned value a name. The second statement is asking the box to move a certain distance in the X and Y directions. We will often speak of objects as having *methods*. A method is a message that the object understands. For instance, we might say that a rectangle object has the `move` method. The basic pattern for sending a message to an object is the following:

```
<object-name>.<message-name>(<some information>);
```

We sometimes call the object that is receiving the message the *receiver*. It's useful to step back into real language for a moment to get a handle on this particular bit of syntax. Look at the following English sentence:

```
Bob eats pizza.
```

At some point, our language teachers totured us with the exercise of diagramming sentences, that is, breaking them down into their constituent parts. Let's do that to the above sentence:

```
Bob [subject] eats [verb] pizza [object]
```

Pretty easy. This pattern is as fundamental to sentences in the languages that humans speak as it is fundamental to Java. In Java, the receiver object is just the "subject" of the sentence. The message we send to the object is the verb or verb clause. In fact, you'll notice that method names are almost always verb clauses. Finally, the information that we send along with the message is the "object" of the sentence. Java just happens to use a different syntax (a different way of writing) to express these kinds of sentences. We could imagine rewriting the above sentence in a more Java-oriented way:

```
Bob.eat(pizza);
```

Text Strings

Most programs manipulate text in one form or another. Because of this, Java provides us with a class that we can use to represent and manipulate text. It's called a `String`.

```
String name = "Billy";
```

We surround the literal representation of the string in double quotes to distinguish it as an object we wish to use. Note that we use double quotes to write Strings but single quotes for characters. You can also send Strings messages, for instance:

```
int lengthOfName = name.length();
String upperCaseName = name.toUpperCase();
String newName = name.replace('B', 'W');
```

A Collection.

Java also provides us with kinds of objects that can maintain a collection of other objects. In much the same way as a drawer might hold a collection of pens, pencils, or clothing, a *collection* in Java can contain other objects. One important type is called the `ArrayList`. Here's an example:

```
ArrayList classList = new ArrayList();
classList.add("Bob");
classList.add("Ellen");
classList.add("Johanna");
```

The above collection now contains three objects, which are all Strings, used to represent the name of three students. How do we figure out who has the longest name? We don't know how to do this in Java yet, and won't for a few more lessons, but you already know how to do this. Given a real class list, think about how you might give a well-defined set of instructions, or algorithm, for finding the longest name.

The Interpreter

Now we'll introduce the interpreter, an important tool for experimenting with the Java language. An interpreter is like a real world language interpreter. Suppose that you don't know how to speak Spanish. If you needed to have a conversation with a Spanish speaker, an interpreter might assist you by listening to each question and answer and translating them into the appropriate language for the benefit of the other party. A Java interpreter does just that. You type a phrase at it, and it translates it into another representation that can be understood by the machine (computer), asks the machine to evaluate what you have typed, and prints the response. An example interaction may look like this (phrases typed by the user are in **`bold`**):

```
prompt> String name = "John";
prompt> name
"John"
prompt> String upperName = name.toUpperCase();
prompt> upperName
"JOHN"
prompt> name.length() + 10
14
```

Patterns we've learned

1. Naming
2. Sending messages
3. Creating new objects

5. Expressions, Types, Messages

Key concepts

1. Expressions and evaluation
2. Statements, sequences of statements.
3. Arithmetic operators
4. Kinds (types) of things
5. Errors: syntax, semantic, pragmatic
6. Argument evaluation
7. Input and Output

Story

Let's quickly review what we know. We know how to create new objects. We know how to name objects. We know how to send simple messages to objects. In this lesson, we'll do more of the same, but we'll scale it up a little bit.

Expressions and arithmetic operators

```
Circle theBall = new Circle(10, 20, 30);
```

The stuff on the left of the '=' is the kind of thing we are naming and the name we have chosen. But what's that stuff on the right? What happens to it? Before we give the thing a name we need to figure out what it really is. To figure out what something really is, we need to *evaluate* it. Evaluation just means that the stuff we write (which we call an *expression*) is allowed to express its true self. What kinds of expressions can we write? An expression can be:

1. a literal representation of an object
2. the generation of a new object (using `new`)
3. the name of an object
4. the result of sending a message to an object
5. combinations of the above

Here are some example expressions:

```
1
"hello"
aSquare
aSquare.width()
new Square(10, 20)
```

The first two items are literal representations of objects; the third is a name of an object; the fourth is a message send; and the final line shows a request to have a new object created.

We can now slightly formalize our naming pattern:

```
<The kind of thing> <the name> = <expression>;
```

Arithmetic operators

Java provides us with mathematical operators that allow us to write mathematical expressions. The basic operators are:

Symbol	Meaning	Example Expression	Value if y is 11
+	add	y + 5	16
-	subtract	y - 5	6
*	multiply	y * 5	55
/	divide	y / 5	2
%	remainder	y % 5	1

We sometimes call the operators in the above table *binary operators* because they operate upon two subexpressions. What if I want to negate a value? In Java you can use the minus symbol in a *unary* manner to do this, just like in regular math:

```
int q = x * - y;
```

Mostly, these operators work just like they work in normal math, with the standard precedence rules. If you're unsure about precedence, you can always use parenthesis to make yourself sure. Here are some examples:

```
int y = 4;
int x = 2;
int m = x + y * 8;       // Q: What's the value of m? A: 34
int n = (x + y) * 8;     // Q: What's the value of n? A: 48
```

Wait, what's the weird stuff after the "//" on the right? If you think it doesn't really look like Java, you're right. Java lets us add *comments* to programs which it dutifully ignores. A comment serves to make a program more readable to another human. Everything from the "//" to the end of the line is ignored by Java.

Note that division between integers behaves somewhat differently than you might expect. If I ask you, "What is 5 divided by 2?", you'll correctly answer "Two and a half". But now, let's look at the following Java fragment:

```
int x = 5;
int y = x / 2;    // Q: What's the value of y?  A: 2
```

In the above example, we're in a little bit of a pickle because y is an integer. It can only represent whole numbers. "Two and a half" is not a valid Java `int` any more than it is a valid real world integer. So what happens? Java will truncate (ie. drop) any remainder that results from a division between integers. Hence, the result that `y` will be bound to is two.

What if we want to know the remainder? Java provides us with an operator that can answer this question for us. It's called the *remainder* or sometimes the *modulus* (or just *mod* for short) operator. Here's an example:

```
int x = 1113;
int y = x / 200;              // Q: What's the value of y? A: 5
int theRemainder = x % 200;   // The remainder will be 113
```

In the above example, 200 goes into 1113 five times, with a remainder of 113. In general, the remainder is the number that when added to the product of the quotient and the divisor, will equal the dividend. Here are a few more examples:

Dividend	**Divisor**	**Quotient (Dividend / Divisor)**	**Remainder (Dividend % Divisor)**
5	2	2	1
13	10	1	3
13	13	1	0
0	13	0	0

Depending on the scenario, it may be appropriate to use a different kind of number that can represent fractions, such as the `double`.

```
double x = 5;
double y = x / 2;  // Q: What's the value of y?  A: 2.5
```

What happens if we write the following?

```
int x = 5;
double y = x / 2;  // Q: What's the value of y?
```

Try it and see. Is the answer what you expected? Why or why not?

Types: Kinds of things

Let's look at a few expressions again. Notice that you can tell me what "kind" of thing each of these expressions evaluates to.

```
"hello"
aSquare
aSquare.width()
aSquare.length() * 23
```

In Java, when we give a thing a name, we have to be sure that we also say what kind of thing we're naming. That's why we're always saying something like this during naming:

```
String greeting = "hello";
```

The word `String` is there to state what the kind of thing it is that we're naming.

It doesn't make much sense to say something like: "Her age is green." In that sentence there's a mismatch between "green" and "age". What is it? When we think of age, we think it should be a quantity, a number. But "green" is a color. They're not the same "kind" of thing, so it seems like non-sense. From now on, we'll use the term *type* to mean "kind of thing." Java is very picky about this kind of non-sense (we'll call them *type mismatches*) and will catch us on it every time. For instance, Java will gripe about the following:

```
String greeting = 7;
int someNumber = "hello";
```

Although it may seem annoying at first, this is actually a good thing. For one thing, imagine what would happen if Java let you add Strings and numbers. What would the meaning of that be? Since it is never the case that a programmer intends to mismatch types, the Java language rightly forbids it in practice.

Errors: Syntactic and Semantic

Let's enrich our taxonomy of errors by looking at some excerpts harvested from that most fertile field of mistaken speech, the words of President George W. Bush.

1. Is our children learning?
2. We ought to make the pie higher.
3. He can't take the high horse and then claim the low road.

Each of them fail to convey meaning, but for different reasons. The first Bushism clearly contains a "grammar" mistake. In Java we call this a *syntactic error*. Here are some examples:

```
int x 7;
int velocity = distance / ;
```

```
String foo = "hello" 56;
```

The second Bushism contains a *semantic error*: we think of pies getting bigger and smaller, but we don't typically think of pies getting higher any more than we think of plants screaming, or pigs flying. Semantic errors in Java are usually type mismatches. Here are a few:

```
String myName = 7;
aSquare.move("hello");  // assume the move method wants an integer!
int area = length * "width";
```

The third Bushism is more subtle, and we may call it a *pragmatic error*. It reflects an unconventional usage of the English language. An equivalent in Java would be poor coding style. Incomprehensible variable names or poor formatting might make your program difficult or impossible to read, even though it's legal by the rules of the Java language.

Sending information with messages

Recall our basic pattern for sending a message:

```
<object-name>.<message-name>(<some information>);
```

When we send messages to objects, we often pass along some other objects that are required for it to answer in an interesting manner. For instance, if we want to move a square, the square wants to know how far to move. If we add something to a collection, the collection needs to know what to add. The objects we pass along with the message are called *arguments* or *parameters*. Note that messages are often really picky about what kind and number of arguments are acceptable. For example, if we ask a rectangle to move, passing along two numbers seems reasonable. For instance:

```
aRectangle.move(10, 20);
```

What happens when we pass along another rectangle or a String?

```
aRectangle.move("hello", 20);
aRectangle.move(35);
```

Neither of the above statements make sense to us, and they don't make sense in Java either. The first makes no sense because the move message expects two integers, but we have provided a string and an integer. This is an example of a semantic error. The second makes no sense because we have only provided one parameter, in a situation where two were expected. This is also an example of a semantic error.

While we can often intuit what kind of arguments are acceptable, to really figure out what kinds of arguments are acceptable, we need to look at the *interface* (or sometimes *signature*)of the method. An interface to a method is a description of the behavior of the method. It defines the the name of the method itself, the number and types of arguments required by the method, and the type of object returned by the method, if any. Often, the interface will also consist of some commentary which describes the behavior or *semantics* of the method. Here is an example, for the move and area methods for Rectangles:

```
// Change the position of the shape by the given deltas.
//  parameters:
//    deltaX: the X distance to move
//    deltaY: the Y distance to move
void move(int deltaX, int deltaY);

// Answer the area of the shape.
int area();
```

The keyword `void` means that the move method does not return or answer with any value. The interface shows us the pattern that we must follow if we are to call the method correctly. In particular, for the move method it states that we must provide two arguments, both integers. Furthermore, it defines the meaning of those two parameters. The interface to the area method, tells us that it takes no parameters but returns a value, an integer, which is the area of the rectangle.

Before the message is sent, the expressions that represent the arguments are evaluated. Look at the following example:

```
aRectangle.move(10, aRectangle.width() / 2);
```

Both arguments are evaluated before the message is sent. Of course, the first expression isn't very interesting, and it just evaluates to the number 10. The second evaluates to half the width of the rectangle before being sent. We can now formalize our message sending pattern:

```
<object-name>.<message-name>(<expressions>);

where
  <expressions> is zero or more expressions depending on the
  interface of the method.
```

Doing a series of things

We often need to do things that require more than one step: baking a cake, washing our hair, changing the oil in the car. Java allows us to express this notion by stringing together statements. Think of each statement as a command to the machine. Each statement is executed, and when it's finished executing, the machine executes the next one. Let's string together some statements that get something interesting done.

```
int width = 20;
Rectangle square = new Rectangle(width, width);
square.move(35, 10);
```

Notice that we separate statements by using semi-colons. The semi-colon is similar to the full-stop "." in written English. It is used to separate sentences. Java isn't picky about how we write our statements, but its considered bad style to put more than one statement on a single line. Why? Look at the above example rewritten:

```
int z=20;Rectangle square=new Rectangle(z,z);square.move(35, 10);
```

Notice that we have not only placed all three statements on one line, but we also not chosen very good names. We've managed to make it very difficult to understand what this fragment is supposed to do.

Input and Output

Nearly every interesting computer system allows the user to interact with it in some manner. When we use an automatic teller machine (ATM for short), we press buttons on a keypad to request a withdrawal and enter an amount. The ATM then may inform us that we have insufficient funds in our account, by displaying a message onto a screen. We call the consumption of information by a program *input* and the production and display of information *output*

In today's computer systems, input typically comes from the keyboard and/or a pointing device called a *mouse*, and output goes to the computer screen. However, there are other unseen sources of input and destinations for output, such as files that hold data on disk drives, network devices that allow our computers to communicate with each other, and printers.

We'll hide many of the details of doing input for now, and imagine that we have a simple interface for performing input. Here is the interface for reading an integer. Reading other simple values occurs through a similar interface.

```
// Prompt the user and read and return an integer
// If they do not provide an integer, they are re-prompted.
// parameters:
//  prompt:  a String to display
int readInt(String prompt);
```

Here's a sample interaction (in the interpreter), using this interface.

```
prompt> Input input = new Input();
prompt> int width = input.readInt("What is the width of the box?");
What is the width of the box?
8
prompt> int height = input.readInt("What is the height?");
What is the height?
hello
Not a valid integer, please try again.
4
prompt> int area = width * height;
```

Performing simple, textual output in Java is pretty simple, although the syntax may seem a little strange for now. The system provides us with a well-known object, called `System.out` that has the following simple interface.

```
// Print a textual representation of the given object,
// followed by a newline (carriage return)
// parameters:
//  o: any object to display
void println(Object o);

// Print a textual representation of the given object,
// without a trailing newline
// parameters:
//  o: any object
void print(Object o);
```

Here's another sample interaction (in the interpreter), using this interface.

```
prompt> Rectangle r = new Rectangle(10, 20, 30, 40);
prompt> System.out.println(r)
[Rectangle: x=10, y=20, width=30, height=40]
```

Patterns we've refined

1. Naming
2. Sending messages
3. Creating new objects

6. Defining Simple Objects

Key concepts

1. Objects as records: aggregates of data
2. Classes vs. Objects
3. Initializing objects
4. Assignment

Introduction

At this point, we know how to create objects, how to name them, and how to ask them to do things. In this lesson, we'll learn how to define our own kinds of objects -- our own classes. By defining our own classes, we put down in writing a definition, or a concept. Once we have stated the definition of our class, we can use it to generate new instances (objects) that are of that type.

Objects as Records

Let's start with a simple example -- a bank account. What are the essential qualities or properties of any bank account? At the very least, every bank account has a balance (the amount of money in the account), an owner, and an account number. Real accounts have lots of other qualities, such as interest rates, monthly fees, personal identification numbers, and so on. For now, however, we'll just worry about defining a simple version of a bank account.

```
class BankAccount {
  int number;
  double balance;
  String name;
}
```

That's it for now. We've just defined our first *class*. A good way to think about a class is that it is a cookie cutter, template, or pattern for manufacturing objects of a certain kind.

Remember, it's important to distinguish between classes and objects. When we analyze the world around us, we can identify concrete things, such as a desk, a dog, or a car. These things are concrete -- that is, they have properties that we can directly sense. However, there are less concrete things that we can conceive of as well, such as the *concept* of car. It's important to distinguish the concept of a car as being in some way different from a particular instance of a car (such as the car in your driveway). The concept of car seems to exist only in our mind, while an actual car exists as a physical object in the real world. Aristotle called the concept a *universal* and the actual physical thing a *particular*. Roughly speaking, this distinction between a universal and a particular is just the difference between class and object in Java. A class is a *description* of the qualities, behavior and implementation that a given set of objects have in common. Objects are instances of classes. They have concrete representations and behavior.

Here is the most basic pattern for a class definition:

```
class <class-name> {
  <instance variable declarations>
}
```

In our example, we've defined a class called `BankAccount`. Each instance of that class will have three *variables* (names) associated with it: number, balance, and name. Notice the types that we have chosen for our instance variables. The balance is a `double`, the account number is an `int` , and the name is a `String`. These types were not chosen by accident, but because they seem reasonable. This is not to say that the types we have chosen are necessarily the best or the only types that we could have chosen. We'll discuss possible variations later, but for now we'll stick with these.

Just as my real-world bank account will have a different balance than yours (withdrawals from my account change my balance, not yours), two BankAccount objects we create in Java will have different balances. Let's create a BankAccount object:

```
BankAccount account = new BankAccount();
```

What are the values of its pieces? Java guarantees that all names are bound to well-defined zero values by default, but usually these values aren't interesting, and it's arguably poor style to depend on them, so let's create an account and set the values of its pieces:

```
BankAccount account = new BankAccount();
account.balance = 100.25;
account.number = 1234;
account.name = "Bob";
```

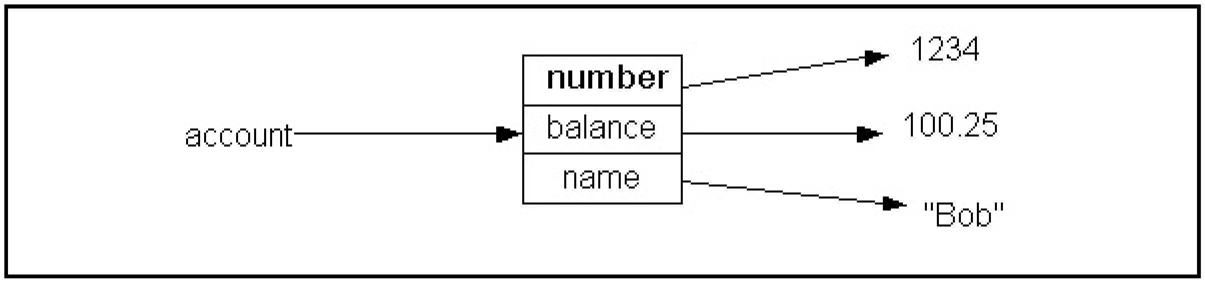
Notice that when we created a new BankAccount, and named it `account`, it is as if three more names were introduced: balance, number, and name. These names belong to the new account, so to use them, we have to use the "dot" syntax (".") to access them. The dot syntax allows us to get access to names that belong to a given object. Those names may be method names (as we've seen already) or variable names (as we're seeing here).

Notice also that these statements *assign* new values to the names of the account -- they change the state of the bank account. In a way, we are rebinding the names to new values. In general, the pattern for an assignment looks like this:

```
<name> = <expression>
```

First the expression on the right hand side of the '=' is evaluated, and then the name is bound to that value.

Let's draw a picture of the account at this point:



How would we deposit 15 dollars into our account? We can do so as follows:

```
double oldBalance = account.balance;
account.balance = oldBalance + 15.0;
```

This works, although it is more convenient to express it as follows:

```
account.balance = account.balance + 15.0;
```

Remember, in an assignment, the expression on the right hand side of the '=' is first evaluated, *then* bound to the name on the left hand side.

---

7. Objects that Do Things

Key concepts

1. Simple methods
2. Methods with parameters
3. Methods that return values

Introduction

So far, we only know how to make "passive" objects: objects that have some important qualities that intuitively belong together. For instance, we are using the BankAccount object to aggregate data that belongs to an account -- its account number, balance, and so on. While this can be useful, we'd like to make our objects be able to do interesting things (respond to messages), like the objects we saw earlier. We'd also like to be able to hide the internal details of a BankAccount object (its representation) behind a veneer of useful methods.

Simple Methods

Let's suppose that we want to assess a checking fee onto a given BankAccount object. Assuming the fee is three dollars, we can do this as follows:

```
BankAccount account = new BankAccount();
// set the values of its pieces:
account.balance = 100.25;
account.number = 1234;
account.name = "Bob";
account.balance = account.balance - 3.0;
```

This works, but not as well as we would like? Why? At this point imagine two different people. One person is the *implementor* of the BankAccount class. She is responsible for the design, implementation, and maintinence of of the BankAccount class. The other person is a *client* of the BankAccount class. They *use* BankAccount objects as an existing abstraction in a system they are building. In small programs, it is at times difficult to distinguish between these two personalities, because the same physical person is often both implementor and client of a given class. However, in large-scale projects consisting of millions of lines of code, there are often sharp divisions between the clients and implementors of given classes. Whatever the size of the project, it is helpful to imagine this split personality whenever you design and build a new class.

With this division in mind, we don't necessarily want to show the client the inside of the BankAccount object. Why not? Because it violates the abstraction we want to develop. We don't want clients to have to know too much about the internal structure of a BankAccount. We don't want them to know not because it's a trade secret, but because not having to know about the internal structure frees them to think about BankAccounts at the appropriate level -- the abstract level. Maybe it's better if we give the responsibility for assessing the monthly fee to the BankAccount object itself. In this view, a BankAccount literally knows how to deduct a fee from its own balance.

Let's modify the BankAccount class by adding a single method to its definition:

```
class BankAccount {
  int number;
  double balance;
  String name;

  void assessCheckingFee() {
    this.balance = this.balance - 3.0;
  }
}
```

We've just written a *method* for the BankAccountclass. There are many things going on in our first method definition, so lets take it apart slowly. The basic pattern for a method definition is the following:

```
<return-type> <method-name> (<parameter-list>) {
  <body-statements>
}
```

Since every method definition must be contained inside of a class definition, we can refine our pattern for a class definition now:

```
class <class-name> {
  <instance variable declarations>
  <method definitions>
}
```

In the above example, we've defined just about the simplest method possible. It is a method that takes no arguments, and hence has an empty parameter list. Furthermore, it returns no value, and hence has a return type called `void`, which is a special name in Java used for just that purpose -- methods that do not return any values. We'll see examples of methods that actually return values and take arguments shortly, but let's use the method we just defined:

```
BankAccount account = new BankAccount();
// set the values of its pieces:
account.balance = 100.25;
account.number = 1234;
account.name = "Bob";

// now assess the checking fee:
account.assessCheckingFee();
```

After we send the `assessCheckingFee` message to the account, the balance in the account is modified -- its state is changed. After the message is sent, the balance should be $97.25. Notice also how much more readable this program fragment is: we are sending the account object a nicely named message, which makes it clear exactly what is happening to the object: a fee is being assessed. This is much more meaningful than a seemingly random assignment statement that is subtracting three dollars from the previous balance.

At this point, only one real mystery about our assessCheckingFee method remains: what about that mysterious word, `this`? The word `this` is a special, implicitly defined name that refers to the object that is receiving the message. We use it to access, using the "dot" notation, the names that belong to the object that receives the message.

Methods with Parameters

At this point, our BankAccount class is still not very useful. We can create new BankAccount objects and assess fees on the account, but not much else. Clearly, only bankers would be happy using these kinds of bank accounts! To have a truly useful bank account, we need to be able to make deposits into the account. Again, we can currently do this simply by setting certain values, as we've done above:

```
account.balance = account.balance + 15.0;
```

This is a less than wonderful solution because it again forces the "user" of the bank account to manipulate the internal pieces of the account. We'd like to be able to say something like this:

```
account.deposit(15.0);
```

In other words, we'd like to send a message called `deposit` that takes a single argument, an amount to deposit. This method should "grow" the current balance of the account by that amount. Let's extend our BankAccount class by adding a new method:

```
class BankAccount {
  int number;
  double balance;
  String name;

  void assessCheckingFee() {
    this.balance = this.balance - 3.0;
  }

  void deposit(double amount) {
    this.balance = this.balance + amount;
  }
}
```

Notice the similarities between our new method, deposit, and the assessCheckingFee. Both methods have a return type of void -- they do not return any values. Both methods manipulate the state of the object -- they change the value of balance. The only real difference is that the deposit method changes the value of balance by an amount specified by the parameter. In the parentheses after the method name, we've added a single parameter declaration, which is very similar to name declarations that we've seen before: it contains a type and a name of our choosing. When the method is invoked (the message is sent) that name will be bound to the value of the argument.

Methods that return values

Suppose we want to figure out how much our account would be worth if we moved it to Europe. Of course, the currency of the European Union is not dollars, but rather the Euro. We'd like to be able to ask our account what its balance would be in Euros. In other words, we'd like to send a message of the following form:

```
double euros = account.balanceInEuros();
```

We can further modify our class to support this kind of operation:

```
class BankAccount {
  int number;
  double balance;
  String name;

  double balanceInEuros() {
    return this.balance * 1.13;
  }

  // other method definitions here
}
```

We've just added a method that takes no arguments and returns a value of type double. We've specified the return type, by adding the word `double` before the method name. We have also introduced the Java reserved word `return` that evaluates its expression and causes the value to be returned to the place the method was invoked.

There is of course a problem with our method. Currency conversion rates change on a frequent basis, and we have hard-coded the conversion rate between Dollars and Euros into our method. A better solution is to further abstract our method, and parameterize it by the conversion rate. This also makes the method useful for converting into any currency, given that we know the conversion rate. Let's replace our old method with an improved one:

```
class BankAccount {
  int number;
  double balance;
  String name;

  double convertBalance(double foreignUnitsPerDollar) {
    return this.balance * foreignUnitsPerDollar;
  }
  // other method definitions here
}
```

We can now calculate our balance in Euros, or in another currency, such as Baht, the currency of Thailand:

```
double eurosPerDollar = 1.17;
double bahtPerDollar = 45.39;
double euros = account.convertBalance(eurosPerDollar);
double baht = account.convertBalance(bahtPerDollar);
```

8. Compiling Java Classes

Key concepts

1. Tools of the trade: editor, compiler, debugger

We know just about enough to write a real program. A program seems like a complicated thing, but most programs really just do the following: get some information from somewhere, do something with that information, and maybe produce some more information and put it somewhere.

So far, we've seen one tool of the trade, which we've called the *interpreter*. This works well for testing out small chunks of code or experimenting with expressions. Many interpreters are unable to interpret class definitions, so we require a slightly different methodology. The programmer (you) type your class definition into a text file, using a *text editor*. At some point, you give your *source code* to another program called a *compiler*. The compiler reads your source code and translates it into a representation that is understandable by a machine. Notice the difference between the interpreter we've been using and a compiler. The interpreter is just like a real language interpreter, who might interpret, sentence-by-sentence a conversation you are having with a speaker of another language. The compiler is more like a translator. Given a document (in our case, containing Java source code), the translator translates it from one language to another (in our case, a kind of *machine language*).

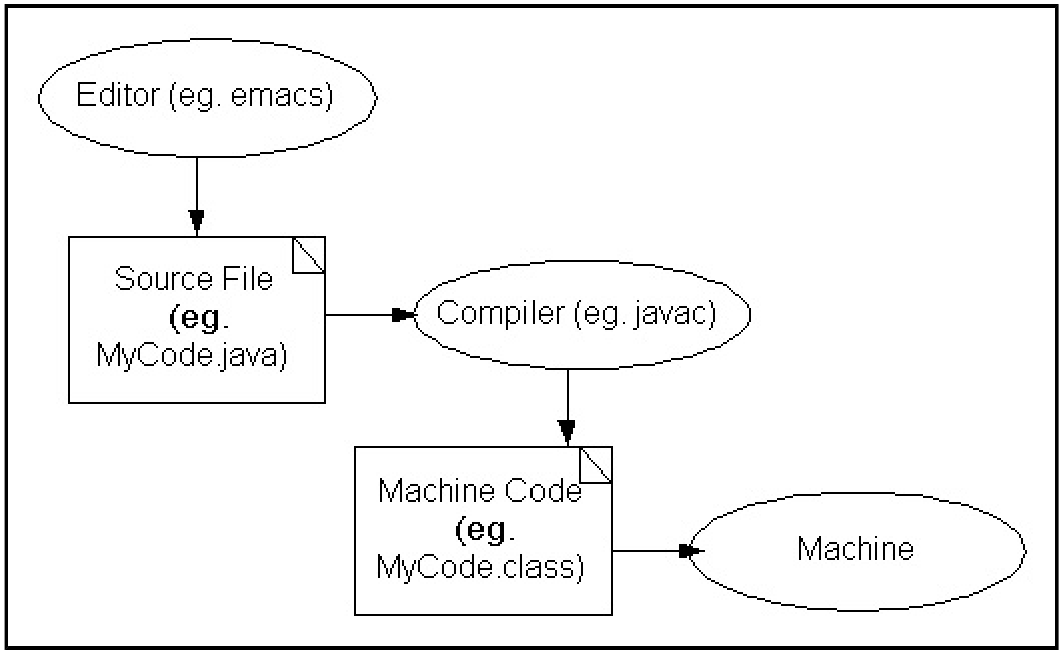
There are many fancy software development environments on the market. Many of them are great at what they do, and we're not particularly religious at using a certain one. However, we do believe that they tend to make the fundamental steps of writing a program seem much more complicated than they really are. A programmer really does the following (at a minimum):

1. Edits her source code (using an editor)
2. Compiles her source code (using a compiler)
3. Possibly repeats steps 1 and 2 if there are syntax/semantic errors
4. Runs her program
5. Possibly (probably) goes back to step 1 to improve and/or fix the program.

Let's summarize the tools:

Tool	Input	Output
Editor (e.g. Emacs)	Keystrokes and mouse input	A Java source file (e.g. MyCode.java)
Compiler (e.g. javac)	A Java source file (e.g. MyCode.java)	A Java class file (e.g. MyCode.class)
A Machine (e.g. java)	A Java class file (e.g. MyCode.class)	The result of your program!

The following image shows the relationship between the tools and the files they consume and produce:


Editor (eg. emacs)
Source File (eg. MyCode.java)
Compiler (eg. javac)
Machine Code (eg. MyCode.class)
Machine

Ok, so what does a Java source file look like? Generally, a Java source file contains just a class definition. Let's take a look at an example class definition for BankAccount. Notice that this particular version of BankAccount only understands two methods: assessCheckingFee and deposit.

```
class BankAccount {
  int number;
  double balance
  String name;

  void assessCheckingFee() {
    this.balance = this.balance + 3.0;
  }

  void deposit(double amount) {
    this.balance = this.balance + amount;
  }
}
```

Try using a text editor to type in the above code into a file called BankAccount.java. Notice that we've given the file the same name as the class definition it contains. Next try to compile it. If your file looks like the one above, it shouldn't compile. It won't compile because it contains a syntax error. The compiler will complain and helpfully tell us where it discovered the syntax error, with a message like this:

```
BankAccount.java:3: ';' expected
  double balance
                ^
1 error
```

We can fix the error by editing the file, and trying to compile it again. When it compiles, try running it. An easy way to run it is to start the interpreter, create a BankAccount object and then send it a message, as follows:

```
prompt> BankAccount myAccount = new BankAccount();
prompt> myAccount.deposit(100.0);
prompt> myAccount.assessCheckingFee();
prompt> System.out.println(myAccount.balance);
```

When you execute the above statements, is the result what you expected? Do you notice that something seems wrong with the balance of the account? This is a kind of error that the Java compiler cannot catch. It's a *logic error* in your program. It's like giving someone directions and telling them to go right instead of left at a critical intersection. The directions are perfectly understandable, but they don't get you to where you want to go. Programmers call this a "bug" in the program. In this case, the bug is that the assessCheckingFee method *adds* three dollars to the account rather than subtracting that amount. Clearly - from the banker's perspective, at least - this does not yield the intended result.

Let's fix the bug. We do so by again making the required changes to the code (in this case, just changing the '+' to a '-', and re-saving the file. We then re-compile the file, re-start the interpreter, and test the class as we did above.

9. Object Wrapup

Key concepts

1. Interface
2. Initializing Objects: the Constructor
3. Information hiding (a little)

Introduction

At this point, we're pretty close to having a complete and reasonably useful BankAccount class. What's left to do? It's worth thinking about the operations that we really need to make our BankAccount objects really useful. What are they? Mostly the answer to this question comes from thinking about the things we do with our real life bank accounts.

1. opening a new account
2. withdrawing money
3. depositing money
4. checking our balance
5. transferring a balance to a new account

While there are surely more operations that we might want to perform on an account, this set seems like a reasonable place to start. We will often talk about the set of operations supported by an object as the object's *interface*. An interface is just a set of method signatures and commentary about the behavior of those methods that allows us to reason about the behavior of an object, without having to worry about the internal details (representation) of the object.

Constructors

Let's go ahead and wrestle with just the first operation, the opening of a new account. Currently, when we create a new account, we must initialize all of its pieces to reasonable initial values, as follows:

```
BankAccount account = new BankAccount();
account.balance = 100.25;
account.number = 1234;
account.name = "Bob";
```

This four step operation is clearly less than conventient. Fortunately, Java provides us with a way to set the values of the pieces of a new BankAccount object at creation time. It's called the *constructor*. A constructor is like a special method whose only purpose is to initialize the instance variables of a new object. Here's an example:

```
class BankAccount {
  int number;
  double balance;
  String name;

  BankAccount(int acctNumber, double initBalance, String acctName) {
    this.number = acctNumber;
    this.balance = initBalance;
    this.name = acctName;
  }
}
```

Notice that the name of the constructor is the same as the name of the class, and that the constructor does not specifiy a return type. In a sense, the return type of the constructor is the type of the class itself. Let's use our new constructor:

```
BankAccount account = new BankAccount(1234, 100.25, "Bob");
```

That's it! Notice that the constructor is only used as part of a "new" expression. When the new object is created, it is invoked to set the values of the instance variables of the new object. What happens if we try this now?

```
BankAccount anAccount = new BankAccount();
```

The compiler will not like this. The reason is that when we define a class, Java provides us with a "default" constructor (with no arguments). However, if we go on and define a constructor that takes arguments, then Java no longer lets us use the default constructor. (At this point, if want an argument-less constructor then we must explicitly provide one. We'll see how in a later lesson.) This seems annoying, but why might it actually be a good thing?

Rounding out the interface

Let's go ahead and add the rest of the operations to our class definition. Many of these are similar to the ones we explored in the previous lesson.

```
class BankAccount {
  int number;
  double balance;
  String name;

  BankAccount(int acctNumber, double initBalance, String acctName) {
    this.number = acctNumber;
    this.balance = initBalance;
    this.name = acctName;
  }

  void deposit(double amount) {
    this.balance = this.balance + amount;
  }
  double getBalance() {
    return balance;
  }

  void withdraw(double amount) {
    this.balance = this.balance - amount;
  }

  void transferFrom(BankAccount other, double amount) {
    other.withdraw(amount);
    this.deposit(amount);
  }
}
```

The first two methods don't contain any real mysteries. Can you see a potential problem with the third method (withdraw)? Would a real bank be happy with this kind of account? We need for our BankAccount object to do something reasonable in the case that there is not enough money in the account. We'll present some solutions for managing situations such as these in the next lesson.

Finally, the transferFrom method shows us an example of a method that relies on other methods we have written. This method takes two arguments: another BankAccount object and an amount to transfer. It first withdraws the specified amount from the other account, and then deposits that amount into the receiver account. Now let's look at some BankAccount objects in action:

```
BankAccount account1 = new BankAccount(1234, 225.34, "Bill");
BankAccount account2 = new BankAccount(23455, 125.0, "George");
account1.transferFrom(account2, 75.0);
double newBalance = account2.getBalance();
```

What are the new balances of the two accounts after the above statements are evaluated?

Information Hiding

Notice that we're hiding the *representation* of our BankAccount objects behind an interface that consists of a set of operations that we can perform on such objects. This is generally regarded as being a good engineering principle, because it gives the implementor of the BankAccount class considerable flexibility in terms of how she might want to represent individual bank account objects.

As a trivial example, consider the different ways we could represent the balance of a BankAccount object:

1. total number of pennies (an integer)
2. number of dollars (a rational number)
3. an integer number of dollars and integer number of pennies

Each approach may have its benefits and drawbacks, but the decision of which representation to use should be internal to the BankAccount class. It should have no impact on *clients* of that class. In other words, software they write that *uses* BankAccount objects should not need to worry about how the balance of an account is represented internally.

We can enforce the visibility rules in Java by modifying our declarations with the keywords `public` and `private`. As a rule of thumb, we'll want to make all instance variables private and most of our methods public:

```
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  public BankAccount(int acctNumber, double initBalance, String acctName) {
    this.number = acctNumber;
    this.balance = initBalance;
    this.name = acctName;
  }

  public void deposit(double amount) {
    this.balance = this.balance + amount;
  }

  // And so on with our other methods...
}
```

Now let's see what happens when we try to directly access instance variables of a BankAccount object:

```
BankAccount account = new BankAccount(1234, 225.34, "Bill");

account.balance = 1000.0;      // ERROR!!
```

The compiler or interpreter will not allow this kind of access now, and will flag the error as such. Notice also that we also labeled the class itself as being public. This allows our class to be used anywhere within the universe of Java objects.

Documenting the Interface

Finally, each method should be preceded by some commentary describing *what* the method does; the meaning and allowable values of its parameters; the meaning of its return value, if any; and any assumptions it makes about the world. We're going to introduce a convention for writing comments that we'll use throughout the rest of the text.

```
/**
  The BankAccount class implements a simple bank account.
  @author Ben Dugan
  @version 11/11/2001
*/
```

```
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  /**
    Create a new BankAccount object, with the given initial values.
    @param acctNumber an account number to use
    @param initBalance an initial balance
    @param acctName the name of the account owner
  */
  public BankAccount(int acctNumber, double initBalance, String acctName) {
    this.number = acctNumber;
    this.balance = initBalance;
    this.name = acctName;
  }

  /**
    Increase the balance by the given amount.
    @param amount the amount to deposit
  */
  public void deposit(double amount) {
    this.balance = this.balance + amount;
  }

  /**
    Answer the current balance.
    @return the current balance
  */
  public double getBalance() {
    return balance;
  }

  /**
    Decrease the balance by the given amount.  This method does not
    guard against overdrawing the account.
    @param amount the amount to withdraw
  */
  public void withdraw(double amount) {
    this.balance = this.balance - amount;
  }

  /**
    Transfer money between accounts.
    @param other a BankAccount to tranfer from
    @param amount an amount to transfer
  */
  public void transferFrom(BankAccount other, double amount) {
    other.withdraw(amount);
    this.deposit(amount);
  }
}
```

Notice that we are using a new style of writing comments. Any text between /* and */ is ignored by Java compilers or interpreters. Furthermore, by starting our comments with an additional asterisk (/**) we are saying that we want these comments to be consumed by a documentation generating tool that can consume our program text file and produce nicely formatted, human readable documentation about our class. We are also making use of special *tags*, such as @author, @version, @param and @return, which tell the documentation tool that we are expressing information regarding the author, the version number of the class, a parameter, or the return value, respectively. The documentation tool will lay out and format these pieces of documentation differently.

10. Understanding How Methods Work

Key concepts

1. Mental models: Substitution and Control Flow
2. Drawing Pictures of Method Invocations

In this lesson, we'll look at a couple of mental models that are helpful in understanding the mechanics of a method call: substitution and control flow. Throughout this section, we'll use the terms *message send*, *method call*, or *method invocation* equivalently.

Substitution

In this model, we can imagine that the call to the method is literally replaced with the code of its body, substituting arguments for parameters. Recall the method definition for `deposit`:

```
void deposit(double amount) {
  this.balance = this.balance + amount;
}
```

The following message send:

```
account.deposit(35.0);
```

could be rewritten as:

```
account.balance = account.balance + 35.0;
```

The rule for rewriting is to first replace the message send with the body of the method definition, replacing the parameter names (`amount`, in this case) with the corresponding values of the arguments (35.0, in this case). Also, the name `this` is everywhere replaced by the name of the object that is receiving the message (`account`, in this case).

Control Flow

We will often talk about the *flow of control* in a program. This is really just a way of thinking about the sequence of statements being executed. Remember that the machine only executes one statement at a time. In a simple piece of code, we simply execute one statement after another:

```
int x = 10;
int y = x * x;
String myName = "Bill";
```

However, in a segment of code containing message sends, there is more going on than meets the eye.

```
double amount = 15.25;                              // line 1
account.deposit(amount);                            // line 2
double euros = account.convertBalance(1.10);        // line 3
```

The first line executes by binding the name `amount` to the value 15.25. The second line calls a method that returns no value. The next statement that executes is actually inside of the method `deposit`. When this statement completes, control passes back to this fragment and the third line is executed. This line binds the name `euros` to the value returned by the `convertBalance` method. To calculate that value, control must pass to the body of that method.

Thinking about method calls in terms of change of control flow is closer to the way method calls actually take place. More formally, when a method is called, the arguments to the method are first evaluated. Next, the names of the method's parameters are bound to the values of its arguments. Finally, control is passed to the first line of the method, and execution continues inside of the body of the method until either a `return` is encountered, or there are no more statements to execute. At this point, control passes back to the *call site* -- the point where the method was originally called.

Drawing Pictures

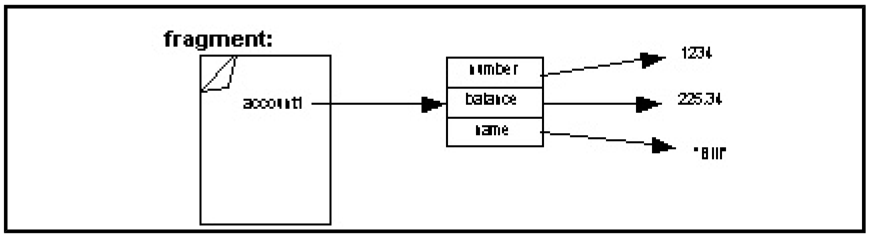
Just as it's been helpful to draw pictures of objects, and the names that refer to them, we can draw pictures to help understand method invocation. Let's use the below class definition as our example:

```
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  public void deposit(double amount) {
    this.balance = this.balance + amount;
  }

  public void withdraw(double amount) {
    this.balance = this.balance - amount;
  }

  public void transferFrom(BankAccount other, double amount) {
    other.withdraw(amount);
    this.deposit(amount);
  }

  // And some other methods...

}
```

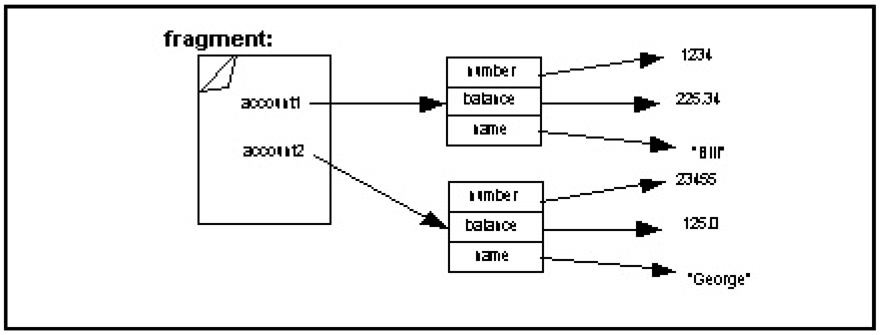
And now, let's draw pictures for what happens when the following fragment executes:

```
BankAccount account1 = new BankAccount(1234, 225.34, "Bill");
BankAccount account2 = new BankAccount(23455, 125.0, "George");
account1.transferFrom(account2, 75.0);
```

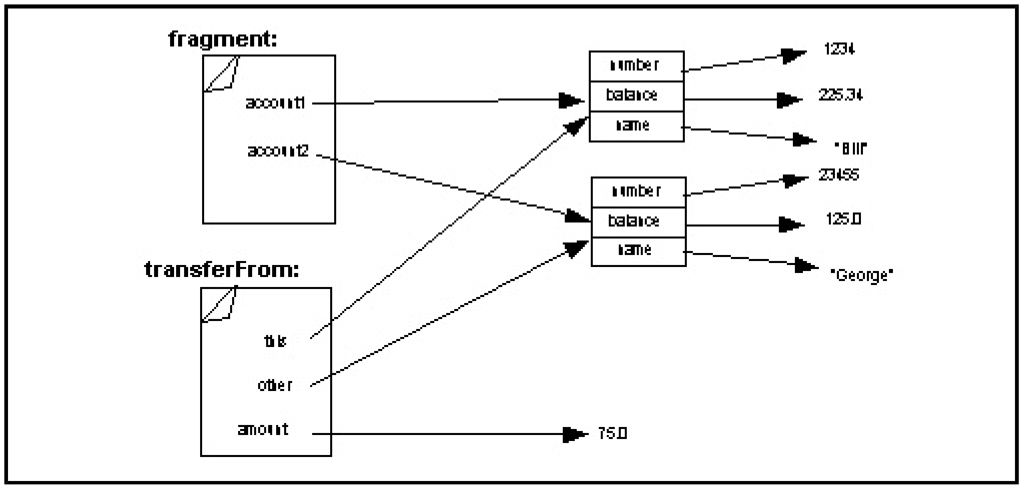
The basic idea is to imagine that the machine has a bunch of pieces of scratch paper. When it needs to execute a sequence of statements such as the above, it gets a new sheet of paper, and starts drawing. After the first line in the above fragment is executed, the following drawing results:



The box on the left is not an object, rather it is the way we draw a sheet of scratch paper. We've titled the piece of paper to differentiate it from other sheets of paper we'll be adding to this picture later on. In this case, we've chosen the arbitrary name "fragment" for our sheet of scratch paper. We modify the picture after executing the second line:

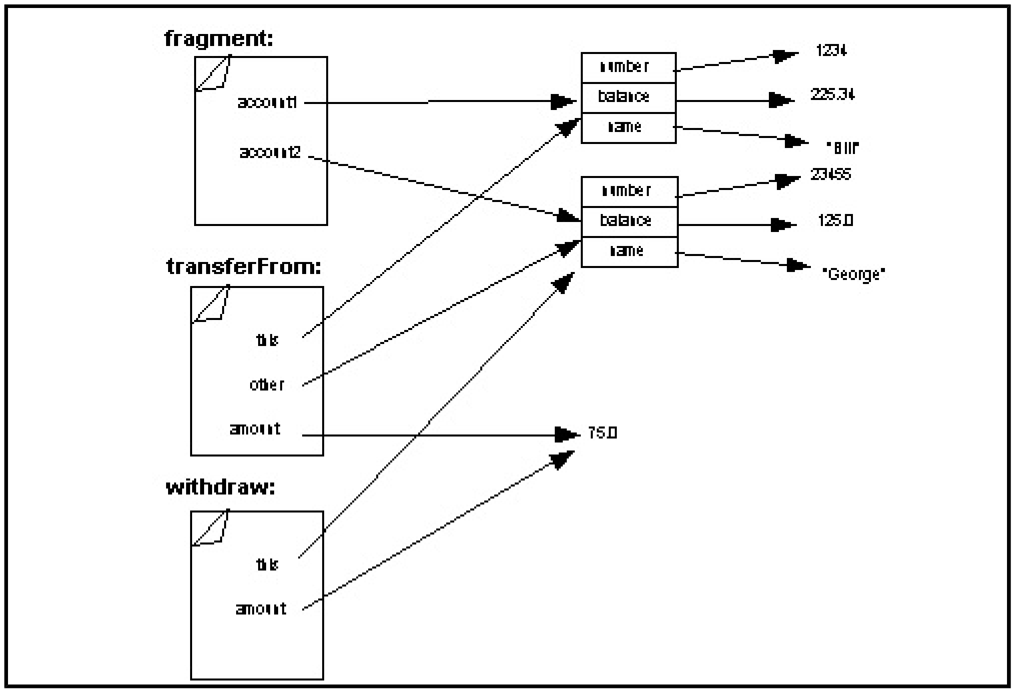


Now, things get interesting. When a message send takes place, we get a new piece of scratch paper, title it with the name of the message, and start drawing on that sheet of paper. Remember, when a method is invoked, parameter names are bound to the corresponding argument values, so we need to show those names on our paper. When control passes to the `transferFrom` method, the following picture results:



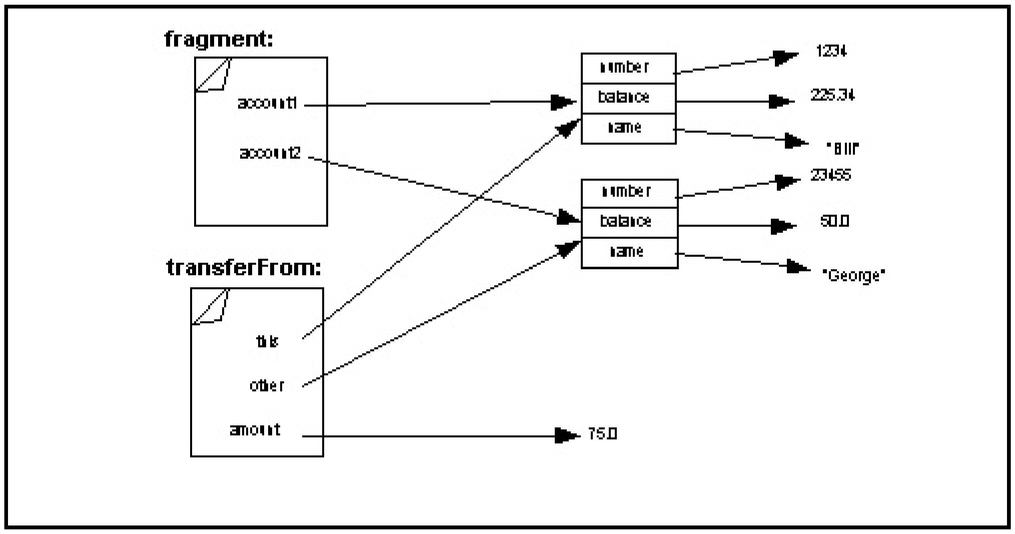
Notice that we've written three names: `this` (referring to the first bank account), `other` (referring to the second bank account), and `amount` (referring to the value 75.0). We show the binding for `this` to denote the object that is receiving the message.

Remember that the first thing that the `transferFrom` method does is call another method, `withdraw`. Again, this means adding another piece of scratch paper to our picture. When control passes to the `withdraw` method, the following picture describes the state of the world:

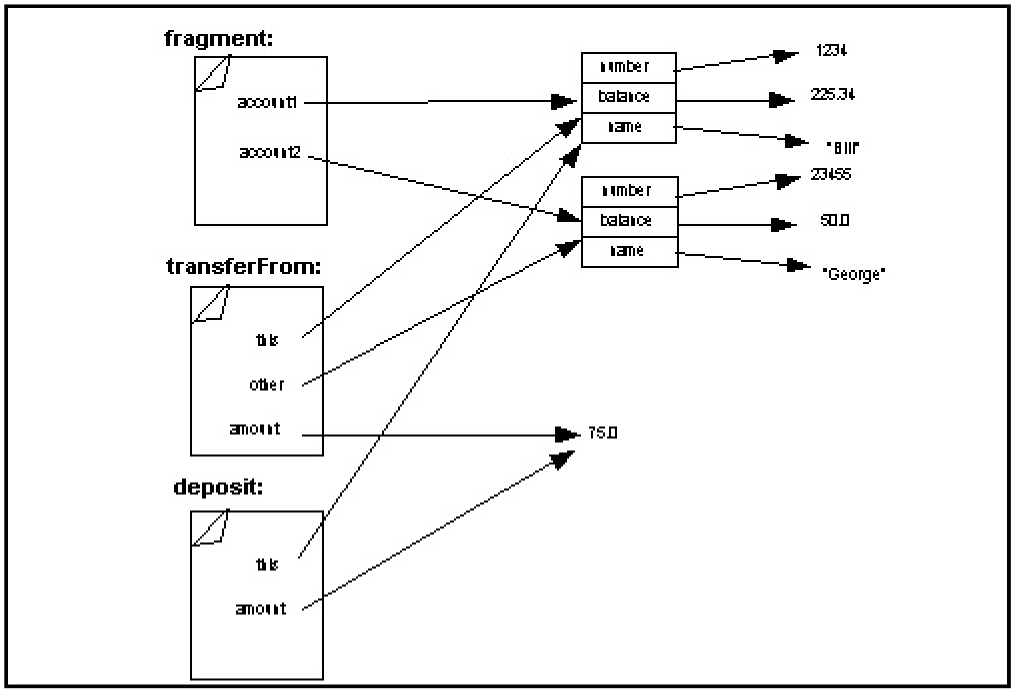


It's starting to look a bit messy, but we've really just added another piece of paper, and a couple of names to our world. Notice that on this new piece of paper, the name `this` refers to a different object than the name `this` on the piece of paper for `tranferFrom`. That makes sense, because each of those message sends have different receivers. Writing down which object `this` refers to helps us keep straight which object is handling a given message.

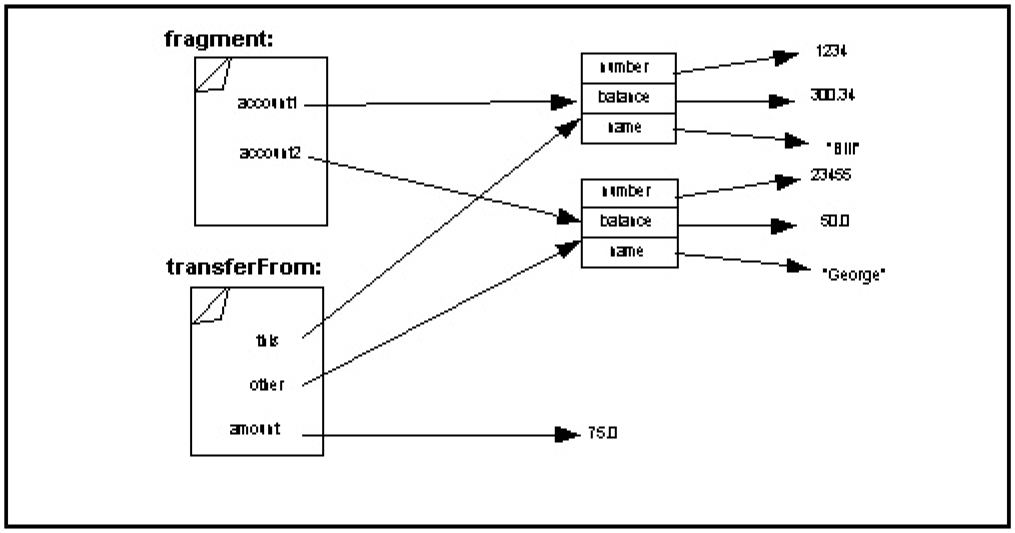
Now imagine that the `withdraw` method finishes. What's next? Well, control returns to the call site -- in the body of the `transferFrom` method. At this point, the piece of paper for `withdraw` is thrown away, and the following picture results. You've seen it before, except that the second bank account is missing some money -- due to the impact of the withdrawal we just made.



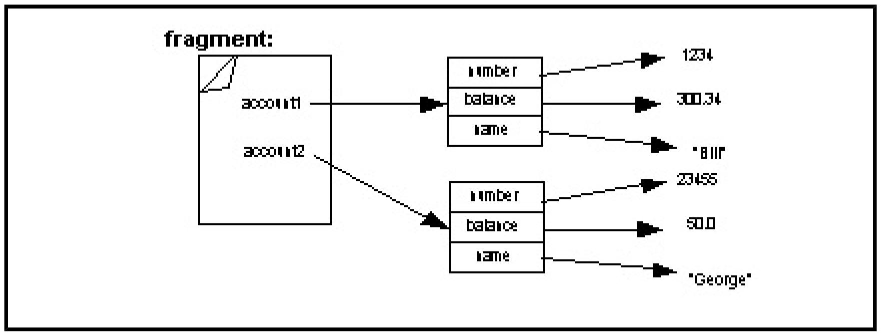
The next line of the body of `transferFrom` sends the `deposit` message to `this`. Again, we modify our picture to reflect the state of the world when that method is invoked:



Of course, the impact of the deposit method is just to increase the balance of the first account by $75.00. This happens, and the piece of scratch paper for this method is thrown away, and control returns to the `transferFrom` method again:



At this point, there's nothing left to do in the `transferFrom` method, so we just throw its piece of paper away, resulting in the final state of the world:



11. Making Choices

Key concepts

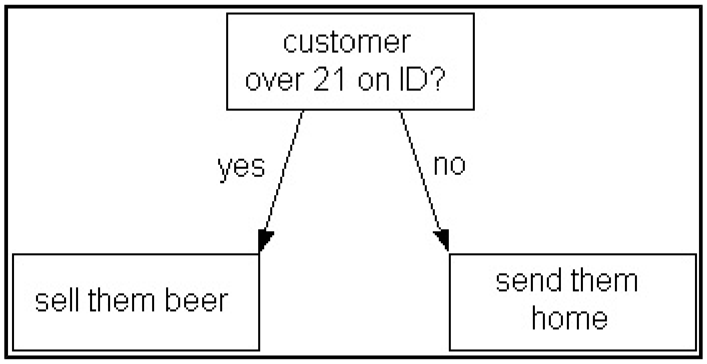
1. The simple conditional: if-then-else
2. Boolean expressions.

Introduction

We know how to define our own kinds of objects now, by defining classes and methods on those classes. However, we'd like to make our objects a little smarter -- we'd like them able to make basic decisions. We'd like them to do different things depending on circumstances or certain inputs from the user or other sources. Look at this algorithm, as used by convenience store owners:

```
Check the buyer's ID.  If the customer is 21 or over, sell them beer,
otherwise, send them home.
```

After inspecting the shopper'd ID, the owner is making a choice. We can show the choice and the two possible outcomes with a *decision tree*.



We can express this in Java as follows:

```
IDCard id = aPerson.getID();
if (id.getAge() >= 21) {
  store.sellBeerTo(aPerson);
}
else {
  store.remove(aPerson);
}
```

The if-statement in Java has the following pattern:

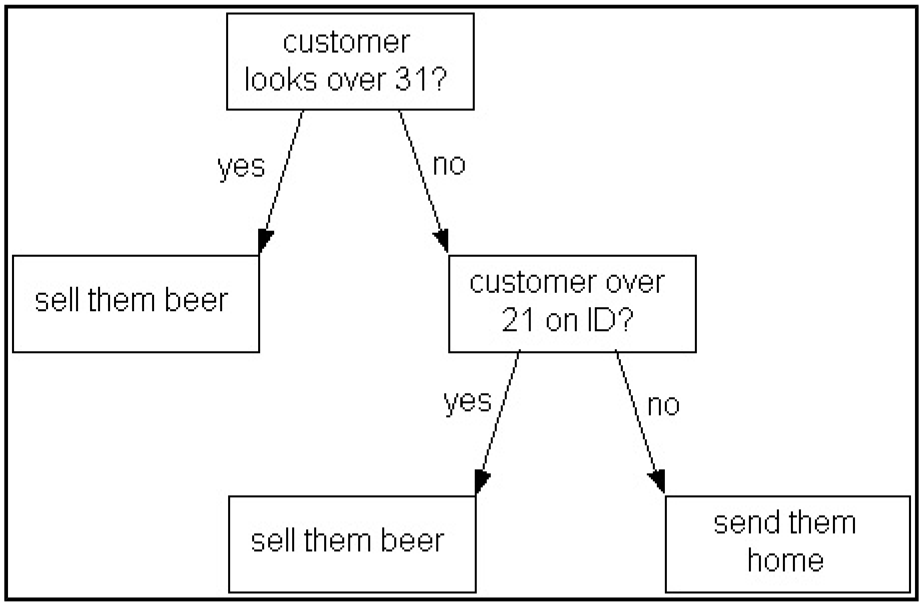
```
if (<condition>) {
  <then clause>
}
else {
  <else clause>
}
```

The semantics is as follows: The `condition` is first evaluated, and it must evaluate to a boolean value. If the expression evaluates to true, the statements of the `then-clause` are executed. Otherwise, the statements of the `else-clause` are executed.

Let's refine our algorithm, so it's more like the actual one used:

```
If the customer looks over 31, sell beer.  Otherwise, check their
ID.  If they are 21 or over, sell them beer, otherwise send them
home.
```

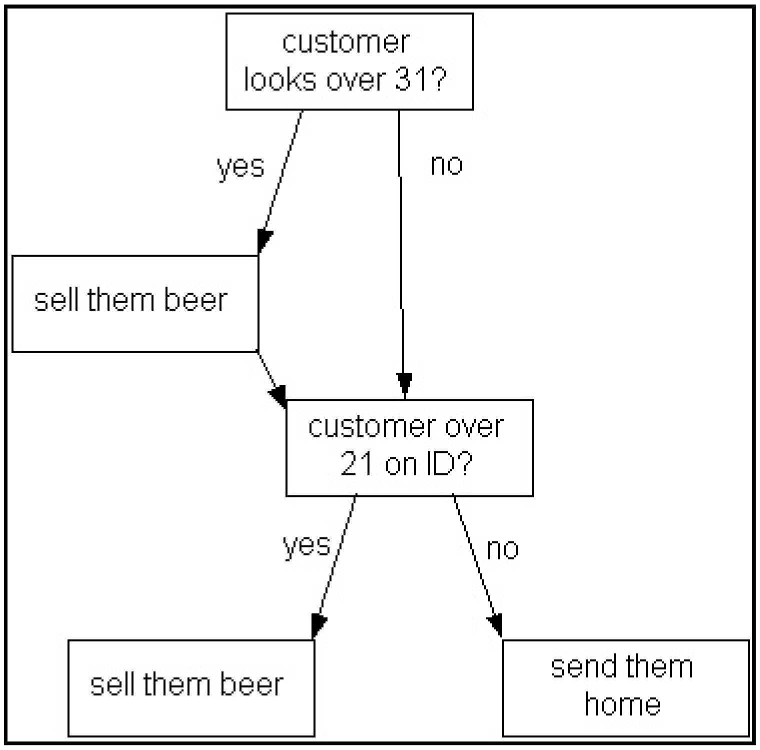
Let's draw the decision tree.



We might write this like this in Java:

```
if (aPerson.apparentAge() > 31) {
  store.sellBeerTo(aPerson);
}
if (aPerson.getID().getAge() >= 21) {
  store.sellBeerTo(aPerson);
}
else {
  store.remove(aPerson);
}
```

This almost works. What's wrong with it? Think about a 50 year-old. The mistake becomes evident when we look at the decision tree.



What we really want to say is:

```
if (aPerson.apparentAge() > 31) {
  store.sellBeerTo(aPerson);
}
else {
  if (aPerson.getID().getAge() >= 21) {
    store.sellBeerTo(aPerson);
  }
  else {
    store.remove(aPerson);
  }
}
```

When writing if-statments, we need to make decisions about exclusivity and order. In the above example, we mean for exactly one outcome to occur. The bug in the example is that we have written our if statement as a collection of two if statements. In some cases (when the person is and appears to be older than 21) they will be sold beer twice!

Java provides a set of operators, which are useful for comparing numbers. These include: <, >,

>=, <=, !=, ==. Each of these operators compares two subexpressions, evaluating to a boolean (true or false) result. != and == are special because they can also be used to compare subexpressions that are not of numeric types. We'll see examples of this later. The following table summarizes the meaning of these operators:

Symbol	**Meaning**	**Example Expression**	**Value if y is 11**
>	greater than	y > 5	true
<	less than	y < 5	false
>=	greater than or equal	y >= 11	true
<=	less than or equal	y <= 10	false
!=	not equal	y != 5	true
==	equal	y == 5	false

Java also provides a set of operators which are useful for combining or transforming boolean expressions. Suppose you want to know if someone's age is between 21 and 31. You can actually already write this:

```
if (age >= 21) {
  if (age <= 31) {
    // do something
  }
  else {
  }
}
else {
}
```

This can be expressed more directly, however, with a boolean operator:

```
if (age >= 21 && age <= 31) {
  // do something
}
else {
}
```

Note that we aren't doing anything inside of our else clause. In this case, it is allowable to simply drop the empty else clause, like so:

```
if (age >= 21 && age <= 31) {
  // do something
}
```

The following table summarizes the boolean operators:

Symbol	Meaning	Example Expression	Value if y is 11
&&	and (true when both subexpressions are true)	(y > 5) && (y < 11)	false
||	or (true when either or both subexpressions are true)	(y < 5) || (y == 11)	true
!	not (true when subexpression is false)	!((y > 5) && (y < 11))	true

Again, there are precedence rules for boolean operators (generally they have lower precedence than arithmetic or comparison operators) but you can always use parentheses to be sure and to enhance readability of your code.

Let's revisit our refined beer-selling algorithm, and use boolean operators to express it more directly. When we think about it, we see that there are really just two outcomes, selling beer and removing the customer. We sell beer when the customer looks over 31 *or* when their ID shows them to be 21 or over. We remove the customer in all other cases.

```
if (aPerson.apparentAge() > 31  || aPerson.getID().getAge() >= 21) {
  store.sellBeerTo(aPerson);
} else {
  store.remove(aPerson);
}
```

Back to the BankAccount

Recall the problems with the withdraw method of the BankAccount class. In some ways, the method is broken -- it allows clients to withdraw more money than the account contains:

```
class BankAccount {
  int number;
  double balance;
  String name;

  void withdraw(double amount) {
    this.balance = this.balance - amount;
  }
}
```

Conditionals allow us to modify the body of the withdraw method to handle this situation. What is the right action to take? At the very least, we should guard against overdrawing the account. Beyond that, there are several options. We would like the client of our class to be informed that an unexpected situation has occurred. We'll see the correct approach to notifying the client in a later lesson, but for now we'll use a strategy that relies on returning a "code" that tells the client whether their operation was successful. We'll return a boolean value: true meaning that the operation succeeded; false meaning it failed.

```
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  // All the other methods...

  public boolean withdraw(double amount) {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      return false;
    }
  }
}
```

The withdraw method now checks for sufficient funds in the account before deducting the withdrawal amount from the current balance. If there are insufficient funds, no change is made to the balance, and the value false is returned. This approach puts the onus on the client of the class to check to return value to see if the operation succeeded. A typical fragment of client code might look like this:

```
BankAccount account = new BankAccount(1234, 225.34, "Bill");
Input input = new Input();

double amount = input.readDouble("How much money do you want?");
boolean succeeded = account.withdraw(amount);

if (succeeded) {
  System.out.println("Here's your money!!");
}
else {
  System.out.println("Sorry, insufficient funds!");
}
```

12. Analysis and Design

Key concepts

1. The design process and software lifecycle
2. Extended example: The Music Collection domain

Introduction

In this lesson, we'll learn about the typical software lifecycle. How does a piece of software (a program) come into being? How does it move from a mere idea to a concrete entity, used by real people?

The Lifecycle

Very roughly speaking, most software is designed and implemented using a process similar to the following:

Stage	Artifacts Created
Analysis	Requirements
Design	Functional Specification
Implementation	Source code
Testing	Test cases, bug reports, certification
Maintenence	bug reports, feature requests

The first stage involves an analysis of the problem domain, which requires careful thinking and investigation of the problem to be solved. It usually results in a *requirements document*, which is a description of *what* problem is being solved. The design phase starts with this document and sketches out a design, a blueprint, for a system that should be able to solve the stated problem. Given a design specification, programmers write source code, during the implementation phase of a project. This source code can then be tested, to help assure that it meets the original specification. Finally, if the program is deemed to behave correctly during the test phase, it is released into the field, at which point maintenence begins. Problems may arise in practice that were not discovered in testing, or perhaps users want small features to be added to the program.

This model of the software lifecycle was originally known as the *waterfall model*, because information and workflow was seen to move in one direction only -- from analysis through to deployment and maintinence. This unidirectional flow and rigid reliance on sharp boundaries between the various phases has been shown to be unworkable and ineffective in practice, especially with large scale software projects. We'll explore some of the reasons for its failure and improvements to the process in a later lesson. For now, however, it is still worth knowing the basic stages of the software lifecycle because it gives us a language for discussing the software development process.

Introductory programming courses typically focus primarily on the implementation phase of a piece of software. Usually, a specification is provided in the form of a homework assignment ("Write a program that does this and that. It should have the following pieces. Etc.") and the end goal is to produce a program that meets the given specification. This is unfortunate because it makes programming seem like an incredibly boring and uncreative task. Further, it seems to suggest that the programmers responsibilities begin with the specification and end with the testing phase. In reality, good programmers must understand the practices and issues faced by all phases of the lifecycle, even if they don't actively participate in those phases.

An Extended Example

Suppose you've been contracted by a radio station to computerize their music collection. For simplicity's sake, assume that their music collection contains only compact discs.

Let's think about compact disks for a moment. What are their properties? Here are a few: total running time, artist, year published, and the songs themselves. Let's go a level deeper. What about songs? Each song has some properties: the music (which is some sort of audio data), the name of the song, the length of the song.

Without knowing it, we've just done an analysis of the music collection domain. If we want to model it we need to define some objects for each of the main concepts. What are the concepts? Finding the right concepts takes experience and practice, but by and large, they fall directly out of our above analysis. Look at the nouns (objects) in the analysis:

```
Music collection, CD, time, artist name, date, audio data.
```

Next, let's think about how the above objects are related to each other.

```
Song has a running time.
Song has a name.
Song has an audio data sample.

CD has a collection of Songs.
CD has a running time.
CD has a name.
CD has an artist.

MusicCollection has a collection of CDs
```

We have just completed an initial analysis of the music collection domain. Are there any problems with it? Well, maybe a recording has more than one artist, so perhaps it's better to make the artist be associated with the Song. Furthermore, CDs usually contain liner notes, a publication date, and perhaps information about label, producer, and engineering information. Ultimately, we would probably want to include these properties, but for the sake of simplicity, let's just stick with our initial analysis.

The analysis will often hint at both the representation as well as the interface of the classes we want to define. It's usually a good strategy to delay worrying about representation and implementation as long as possible. Sketching the behavior of objects and imagining their interactions is much easier and flexible than actually writing the actual code for them. Good programmers build worlds in their imaginations before building them in a program.

Let's imagine some of the operations that our Song class should support. Again, let's think about what we can do with real world songs.

1. create new songs
2. get their running time
3. get their title
4. play a song (on an audio player of some sort)

At this point, we can actually sketch out the interface of the class in Java:

```
public class Song {

  // Create a new song

  // Get the running time of this song

  // Get the title of this song

  // Play this song onto an audio player
}
```

Notice that all we have so far is the basic class structure and some comments, but it's a good starting point. In many cases, these comments will be even more formal -- they will specify to a high degree of detail the behavior of the methods we are going to want to write. The operations we wish to provide hint at the state that each Song has to maintain: some notion of song length, title, and the audio data for the song itself. This allows us to add the instance variables to our class definition.

```
public class Song {
  private int seconds;
  private Sound sound;
  private String name;

  // Create a new song

  // Get the running time of this song

  // Get the title of this song

  // Play this song onto an audio player
}
```

Now, writing at constructor and the first two methods is pretty trivial.

```
public class Song {
  private int seconds;
  private Sound sound;
  private String name;

  // Create a new song
  public Song(String title, String lengthInSeconds, Sound theSound) {
    this.seconds = lengthInSeconds;
    this.sound = theSound;
    this.name = title;
  }

  // Get the running time of this song
  public int getLength() {
    return this.seconds;
  }

  // Get the title of this song
  public String getTitle() {
    return this.name;
  }

  // Play this song onto an audio player
}
```

What about playing the song? Let's imagine that we know about a class that specializes in playing Sound objects. It might have an interface that looks something like this:

```
class AudioPlayer {
  ...
  // Play the provided sound onto the audio device.
  public void playSound(Sound aSound);
}
```

Based on this interface, we can complete our implementation of the Song object. Playing a song is just a matter of passing the song's sound to a given AudioPlayer object, yielding a complete implementation of our Song class.

```
public class Song {
  private int seconds;
  private Sound sound;
  private String name;

  // Create a new song
  public Song(String title, String lengthInSeconds, Sound theSound) {
    this.seconds = lengthInSeconds;
    this.sound = theSound;
    this.name = title;
  }

  // Get the running time of this song
  public int getLength() {
    return this.seconds;
  }

  // Get the title of this song
  public String getTitle() {
    return this.name;
  }

  // Play this song onto an audio player
  public void play(AudioPlayer player) {
    player.play(sound);
  }
}
```

We've just implemented a class that we can use to represent songs. In future chapters, we'll see how to implement some of the other classes in our system, such as MusicCollection and CompactDisc.

13. Reasoning about Programs

Key concepts

1. Program Correctness
2. Specification/Requirements (for programs and methods)
3. Testing, reasoning about programs.

Story

How do you convince yourself that a program works? How can you conclude that a method you write does the *right thing*? What is the right thing? Asking and answering these kinds of questions are all part of being a good programmer. Just like it's unacceptable to turn in a term-paper without proofreading it, it's unacceptable to write a program without convincing yourself that it works.

In the simplest terms, a working program is one that solves the problem it set out to solve. The criteria of success is based largely on the description of the original problem. For instance, if I tell you: "Write a program that does stuff." It's very easy to write this kind of a program and to convince yourself that it works. Of course, I might not be very happy with it, because it probably doesn't do what I expected. If I want something more specific, I need to be clearer in my request. So I might say: "Write a program that draws a house." Now you have a somewhat more specific problem to solve, and we can have a more reasonable discussion about whether or not your program actually solves it. Suppose you deliver a program that draws a house with no windows. I might complain, but arguably your program has solved the problem I set forth. If I'd wanted windows, I should have made that part of my request.

The technical term for this kind of request is a *specification*. Specifications should have a level of detail sufficient to solve the real-world problem they are setting out to solve. For instance, the specifications for an air-traffic control system are likely to be much more detailed than the specifications for a beginning programming assignment.

Another useful way to think about specifications is that they express a contract between a *client* and a *vendor*. Contracts often have certain *requirements* (things that the vendor expects to be true for the contract to be excuted), and make certain *guarantees* (things that the vendor insists will be true once the contract is executed). Programming by contract is a useful way to split large programming tasks into smaller components. We can apply the concept of specification at various levels of a program. Initially, we might just write a specification for the whole program, but later, we'll break it down, perhaps writing specifications for individual methods that need to be written. These specifications are often advertised as part of the interface of a method. Having specifications at various levels and degrees of detail allows us to reason more effectively about the supposed behavior of a program.

The phrase "meets the spec" is often used to mean "the program works." How can you convince yourself that the program "meets the spec"? There are several approaches -- including testing and analysis. Suppose I give you this specification:

```
Write a program that reads a number between one and a million,
divides it by two, and prints the result.
```

You could imagine trying your program on every possible value and checking the results. If they are correct for every input, then your program works -- it meets the spec. While this might work for toy programs, it's not very practical for larger pieces of software. In these cases, testers will often build suites of test cases. If the program "passes" all of the test cases, they can say it works for the given test suite. Finding a good set of test cases is an art in itself, and we'll talk about finding good test cases later on.

Specifications themselves are of course rooted in human language, so it seems that they can only be as precise as our own, somewhat imprecise language. In the extreme, the definition of terms itself seems like it leads to an infinite regress. Consider the following example:

```
Write a program that takes two numbers and prints their sum.
```

This seems simple enough, but it assumes that the terms "program", "numbers", "prints", "sum", and so forth are well-defined. We typically take the definition of these sorts of terms for granted, but imagine the dialog you could have with an inquisitive two-year old:

```
- What does 'print' mean?
- It means to show the information on the computer screen.
- What's a computer screen?
- It's like the television, but it's connected to the computer.
- What's a computer?
- It's a machine that can execute a set of instructions you give it.
- What's an instruction?
- [And so on, and so on...]
```

Clearly, I'm exaggerating the point here, but I want to show that there is a lot that we take for granted when we give and receive specifications. Our language is full of subtle shades of meaning, that can be interpreted in a variety of ways. People have attempted to make the specification process more formal, usually by creating increasingly mathematical notation. In some ways, this can make specifications more clear, but in some sense, it skirts the issue. If the problem domain itself is not a mathematical one, a highly formal notation may not assist the designer. Imagine trying to create a system to support the work of air-traffic controllers. While some aspects of the air-traffic control problem are undoubtedly mathematical in nature, studies have shown that its effectiveness is rooted in large part in the language, practices, signs, and implicit knowledge of the air-traffic controllers. Besides relying on instruments to make good and safe decisions, controllers engage in constant linguistic negotiation and interaction with pilots and other controllers both locally and at other airports. Ultimately, it seems that we need to answer the fundamental question of whether or not our own language is completely and unambiguously formalizable. The quest to formalize human language characterized many of the great philosophical projects of the 19th and 20th century. Today it is generally accepted (at least outside of the Computer Science discipline!) that they all ended in failure.

In terms of structuring programs as contracts, the same problems arise. Arguably, calling specifications "contracts" is just sugaring over the core issue. To be effective, the contracts themselves must be crafted in just the appropriate amount of exacting detail. Having a contract is not enough -- it has to be a "good" contract. Lawyers understand that contracts are incredibly difficult to write well. It is no mistake that Contracts is typically regarded as one of the most difficult courses a law student will face. In a sufficiently large software system, writing a contract can be as difficult as writing a contract in the real world. Eventually, the contractual view seems to return us to the deep philosophical issues we wrestled with above.

At this point, you may be thinking that if it's so difficult or even impossible to specify anything at a sufficient level of unambiguous detail, how can we ever write a program that "works." The solution is perhaps to redefine what we mean by "works." A working program is one that is generally useful to the people who use it in their everyday lives. It is a program that supports rather than hinders their daily practices.

Many philosophers have wrestled with the question of understanding. In other words, they attempted to answer the question: "How do humans understand language? What does it mean to understand a word or a sentence? What is meaning?" Wittgenstein gave what many people consider a rather exotic answer to this question. Traditionally, this question has been answered in a denotational manner. In other words, the meaning of a word is that which it denotes. The word "apple" denotes (refers to) a crunchy, sweet, juicy fruit that grows on trees. For you to understand the word "apple" is for you to grasp this reference. Wittgenstein argued, however, that this view of language is impoverished. Consider a one year old who is just learning to speak. She may utter the word "banana". What she really "means" by saying that word is that she wants a piece of the banana that she sees sitting on the counter. For you to "understand" her is to actually cut and mash the banana and give her some. The traditional view of language does not explain this action- and use-oriented quality of language. For you to understand is to "do the right thing," rather than simply grasp the denotation of the word "banana" to the concept of the thing you have in your head.

Stepping back into the world of computer programs now, we see that the difficulty with specifications is that they do not capture the richness of meaning in the same way that a denotational theory of language cannot express the richness of how we use language. Fundamentally, we humans use programs to do things, the same way that we use language to do things. To have a successful (that is, understood by both parties) conversation is to get the thing done that you are wanting to get done. To have a successful, working computer program is to write one that gets done what you intended. The metric of success is usefulness to the humans that use it rather than simply meeting the specification.

This redefinition of success implies a markedly different approach to designing and developing computer programs -- where designers, developers, and users work together and contribute broadly to the creation of a software artifact. By working together more closely, designers and developers can better learn and understand aspects of the user's domain that may not have been well expressed in the initial specification. Furthermore, by testing prototypes - early versions of the software - on actual users, feedback can be gathered at a time when it makes the greatest impact -- early in the project. Understanding and improving the concept of *user-centered* or *participatory* design is an active field of research at the moment.

While specifications have their weaknesses, I want to emphasize that they do play an important role in the software development process. Specifications *are* useful. While it is arguable that they are even a necessary condition for building good software, it is a mistake to believe that they are sufficient condition. Perhaps the best we can do is formalize a specification to the extent that it assists us, but not to the extent that we lose the richness of the orginal domain. When asked about rules (definitions) for using language, Wittgenstein provided a characteristically fuzzy answer. He argued that rules should be viewed more as signposts at a fork in the road. Rules are adequate, he said, if they eliminate our doubt as to which direction to take. While the signpost cannot guarantee that someone will take the correct fork in the road, they will tend to do so if they are effective in their design. The metric of their success is their usefulness, rather than a normative standard of completeness or definiteness. By answering the question this way, he underscored the importance of rules without needing to make any claims or requirements as to their form.

---

14. Introduction to Collections

Key concepts

1. Collections
2. Ordered collections: ArrayList
3. Casting
4. Unordered collections: HashSet and HashMap

Introduction

The concept of a collection is deeply rooted in our everyday experience. Think about the following concepts:

1. dictionary
2. class list
3. deck of cards
4. music collection
5. library
6. bookbag

A dictionary is a collection of words and their definitions; a music collection is an assortment of compact discs, cassette tapes, and so on; a bookbag is a container for items we might use in school. All of these concepts share an important quality: they are all collections of things.

Some collections are inherently *ordered*. A deck of cards, for instance, has a first card, a last card, a seventh card, and so on. The books in a bookbag or backpack are not really ordered, on the other hand. What is the first book in the bookbag? You could answer this question a variety of ways: the alphabetically first book; or perhaps the first book you put into the bag. In any case, it seems that the answer to this question is unclear. Hence, we might say that a bookbag is an example of an *unordered* collection.

Because collections are so fundamental to the way we think about the world, every programming language provides some means to represent a collection. Java provides us with a particularly expressive set of collection classes, which we can use in our programs to build powerful abstractions. In this lesson, we'll take a look at just three examples, Arraylist, HashSet, and HashMap.

An Ordered Collection: ArrayList

ArrayLists is a standard Java collection class that provides *indexed* access. That is, they we can ask for the seventh (or in general, the i-th) element of the collection. The first element of an ArrayList is always located at index 0, rather than index 1. The final element of an ArrayList is found at index size-1, where size is the number of items in the collection. Starting to count at "zero" may seem strange or awkward to a beginning programmer because most humans start counting at "one". Let's look at a few of the important messages understood by ArrayLists:

```
public class ArrayList {

  /** Answer the size of the collection  */
  public int size();

  /** Answer the object at the given index */
  public Object get(int index);

  /** Add the given object to the end of the collection */
  public void add(Object o);

  /**
    Remove the object located at the given index from the collection,
    and shift any subsequent elements to the left.
    @return the element that was removed from the list
  */
  public Object remove(int index);
}
```

Now let's look at a simple example of using an ArrayList (in the interpreter):

```
prompt> ArrayList names = new ArrayList();
prompt> names.add("Ahab");
prompt> names.add("Quequeg");
prompt> names
ArrayList["Ahab", "Quequeg"]
prompt> names.get(0)
"Ahab"
prompt> names.size()
2
```

In the first line in the above example, we create a new ArrayList object and name it `names`. We then perform some operations on this object, namely, we add two objects to the collection -- the Strings "Ahab" and "Quequeg". We then ask the ArrayList to show itself, and it shows us that it contains the two Strings that we just added. Notice that they occupy the order in which they were added. Next, we ask the ArrayList to access the element at the zero-th (first) position. This should be the object we added first, namely "Ahab", and we see that indeed it is. Finally, we ask the ArrayList to tell us its size, and it informs us that it contains two elements, as we would expect.

Rounding out the Music domain: The CompactDisc class

Now let's revisit the Music domain we considered a few lessons ago. In that lesson, we built classes to represent individual Songs, but stopped short of implementing a class to represent the CompactDiscs that contain those songs. Recall some of the properties of a CompactDisc:

```
CompactDisc has a collection of Songs.
CompactDisc has a running time.
CompactDisc has a name.
CompactDisc has an artist.
```

Let's think also about the interface that a CompactDisc object might support, by thinking about the kinds of things that we might do with real-world CompactDiscs:

1. create new discs
2. get the title of a disc
3. get the artist of a disc
4. add a song to a disc
5. get the total running time of a disc
6. get the ith song of the disc

This simple list of uses hints at the interface a CompactDisc object might want to support. We can sketch it out like so:

```
public class CompactDisc {

  // A constructor for making new CompactDisc objects

  // Get the title

  // Get the artist

  // Get the total running time

  // Add a song

  // Get the ith song

}
```

Now we extend our implementation by giving it a representation, guided by the property analysis we performed above:

```
public class CompactDisc {
  private ArrayList  songs;          // a collection of songs
  private int        runningTime;    // total running time
  private String     artistName;     // artist name
  private String     title;          // disc title

  // A constructor for making new CompactDisc objects

  // Get the title

  // Get the artist

  // Get the total running time

  // Add a song

  // Get the ith song
}
```

Now we can go ahead and add a constructor and implement the five methods quite easily:

```
public class CompactDisc {
  private ArrayList  songs;
  private int        runningTime;
  private String     artistName;
  private String     title;

  /**
    Create a new, empty compact disc.
    @param artistName the name of the recording artist
    @param title the title of the CD
  */
  public CompactDisc(String artistName, String title) {
    this.artistName = artistName;
    this.title = title;
    this.songs = new ArrayList();
    this.runningTime = 0;
  }

  /** Answer the title of the CD. */
  public String getTitle() {
    return this.title;
  }

  /** Answer the artist of the CD. */
  public String getArtist() {
    return this.artistName;
  }

  /** Answer the running time of the CD. */
  public int getRunningTime() {
    return this.runningTime;
  }

  /** Add the given song to the end of CD. */
  public void addSong(Song aSong) {
    this.songs.add(aSong);
    this.runningTime = this.runningTime + aSong.getLength();
  }

  /** Answer the ith song.
      @param i the index of the song to retrieve
      @return the Song at the ith position
  */
  public Song getSong(int i) {
    return (Song)this.songs.get(i);
  }

}
```

The constructor just initialized the parts of the CD to reasonable values: it sets the artist name and disc title; it creates a new, empty ArrayList to hold the songs; and finally it sets the running time to be zero. The first three methods are just simple accessor methods that let the client of this class get access to some of the basic qualities of a CompactDisc, such as artist name.

The `addSong` method is a little trickier, but not totally far out. It just adds the Song object (given as a parameter to the method) to the collection of songs. Finally, it increments the running time. Notice how this operation keeps the total running time up to date: every time a new song is added to the CD, it is queried for its running time, and this value is accumulated in the total running time of the CD.

The final method, getting a song from the disc, does indeed look somewhat strange. Can you see what is different about it? It looks the way it does because of a few special rules that the Java language imposes upon us. We'll talk about these rules in the following section.

Casting

Note that the return type of the `get` method for ArrayLists is Object. For now, think of the type `Object` as Java's way of expressing any kind of object defined by a class in the language. This includes "built in" classes like String and classes that we define, such as Songs. It does not include, however, primitive built in types such as int, char, double, or boolean. Every class we define has an "is-a" relationship to the class Object. In other words, we can say that every Song "is an" Object. Notice, however, that the reverse is not true: not every Object "is a" Song. This is true of the real world too. For instance, every car is an object, but not every object is a car. There could be objects that are not cars, but instead are chairs, or stoves, or some other kind of object. This rule means that a binding such as the following is perfectly legal:

```
Object someObject = new Song(...);
```

The type of the expression on the right is Song. And since every Song is an Object, the above does not result in a type mismatch. Again, the reverse is not true. In other words, it is not the case that every Object is a Song. Let's look at the following fragment:

```
ArrayList someSongs = new ArrayList();
someSongs.add(new Song(...));
...
Song aSong = someSongs.get(0);              // type mismatch!!
```

The return type of the method get() is Object, and the compiler will complain of a type mismatch. Why? Because type of the thing being returned from the get() method is Object and in the final statement, we are saying that it is a Song. In other words, we are claiming that an Object is a Song, which is not, in general, true.

This should help explain the strange-looking method body of `getSong()` in the class CompactDisc. We know that our ArrayList `songs` contains only Song objects. A client of our class expects that when they get the i-th song of a CompactDisc, they should receive a Song, not an Object. The problem is that the `get()` method on ArrayList returns an Object. We are faced with a seemingly impossible situation now -- as implementors of the class CompactDisc, we know that the ArrayList contains only Songs, but the compiler will insist it contains only Objects. How can we make the compiler happy? The answer is to *cast* the type of the object that is returned by get() to the appropriate type. The pattern for casting is as follows:

```
(<type-name>)<expression>
```

A cast is a promise to the compiler that the expression is actually of the stated type. The cast expression does not actually change the type of the value of the expression -- if the type of the expression is not as we promise, an error will be reported at runtime. Let's look at another example:

```
ArrayList things = new ArrayList();
things.add("Hello");
Song aSong = (Song)things.get(0);       // runtime error!!
```

The above fragment will compile beautifully, but will fail spectacularly when run. Why? Because we have in essence told a great lie. We have made the claim that the object returned by the get() method is actually a Song, when it really is a String. Since it is not the case that a String is a Song, the program cannot run correctly. Since the compiler cannot, in general catch errors like these when it compiles your program, checking these kinds of errors must be deferred until the program is actually being run.

Casting is a practice that is subject to terrible abuse, and hence must be used carefully. As we know, Java is very strict about using and abusing types, and does its best to catch abuses of types at compile time. Many beginning programmers use casting to get the compiler to shut up about type errors, only to pay the price when their programs are actually run. When using a cast, ask yourself the question, "Is it really the case that this expression is of this type?" If you cannot answer in the affirmative, then you should not be using a cast.

In the `getSong` method, we can ask ourselves that question and indeed answer in the affirmative. Why? Because we are the only people who can add things to the ArrayList, and we know (because of the interface of the `addSong` method), that we'll only ever add Song objects to our collection. Hence, when we access an object in that collection, we know that it must be a Song. Therefore, we may use a cast to make this promise to the compiler, and support the desired interface to the `getSong` method, namely that it should return an object of type Song, not Object.

Notice that the CompactDisc class is a relatively "thin" class. Its primary job is to maintain a collection of objects (Songs) and provide a client a nice interface for adding and accessing songs as well as getting general information about that collection. It performs these operations by building on pre-existing classes (such as String and ArrayList). The CompactDisc class is an example of a *facade* class. It provides a client with a clean, uniform interface to a set of sub-objects that together implement some sort of object. We'll see another example of this style of programming in the next section.

An Unordered Collection: HashSet

A *set* is a collection that does not contain duplicates and does not have a specific order. In mathematics, the set {1, 2, 3} is the same as the set {3, 2, 1}. Furthermore, sets support operations such as intersection (the intersection of two sets is the set that contains elements that exist in both sets) and union (the union of two sets is the set that contains elements that exist in either or both sets). Java provides us with a class that we can use to conveniently represent sets in our programs, called the HashSet. HashSets are used less frequently than ArrayLists, but are important for understanding our next collection, the HashMap.

An Unordered Collection: HashMap

Suppose we want to further extend our music collection example. So far, we have written classes to represent songs and compact discs. What about the music collection itself? As the name suggests, it is clearly some kind of collection. Primarily, we wish to access the items in this collection by the title (or perhaps artist). Java provides us with a kind of collection that makes this easy -- the HashMap. Rather than wade through a lengthy analysis and development of the MusicCollection class, let's jump right to an example implementation.

```
public class MusicCollection {
  private HashMap discs;
  private String name;

  /** Create a new collection with the given name. */
  public MusicCollection(String collectionName) {
    this.name = collectionName;
    this.discs = new HashMap();
  }

  /** Add the given disc to the collection. */
  public void add(CompactDisc disc) {
    this.discs.put(disc.getName(), disc);
  }

  /** Answer the disc with the given title. */
  public CompactDisc get(String title) {
    return (CompactDisc)this.discs.get(title);
  }

  /** Remove the disc with the given title. */
  public void remove(String title) {
    this.discs.remove(title);
  }
}
```

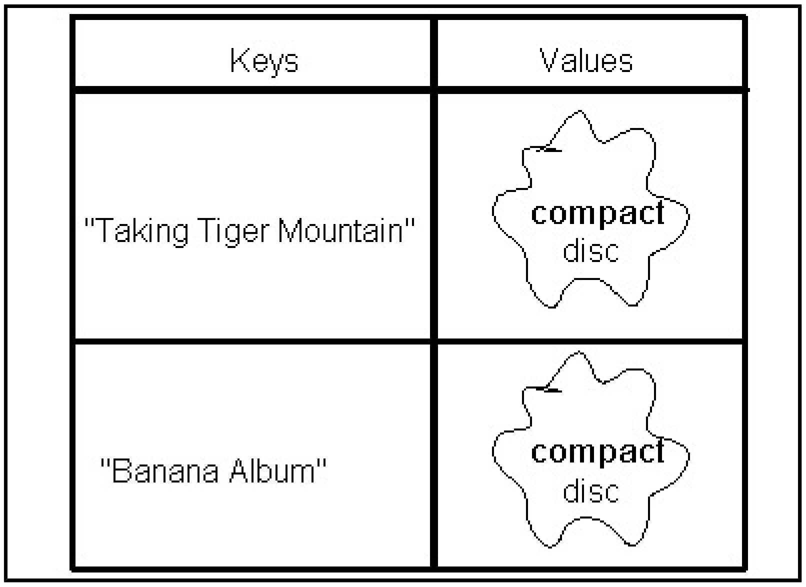
A MusicCollection object really just has two fields, a name and a collection of CompactDiscs. To represent the collection, we use the HashMap class. Here's a peek at its interface:

```
public class HashMap {

  /** Answer the value associated with the given key. */
  public Object get(Object key);

  /** Associate the given key with the given value. */
  public void put(Object key, Object value);

  /** Remove the key and its corresponding value from the table.
      @return the value if it exists, null otherwise */
  public Object remove(Object key);
}
```

A HashMap is a collection that *maps keys to values*. Let's look at a fragment that uses a HashMap:

```
HashMap songs = new HashMap();
CompactDisc disc1 = new CompactDisc("Banana Album", "Velvet Underground");
CompactDisc disc2 = new CompactDisc("Taking Tiger Mountain", "Brian Eno");
songs.put(disc1.getName(), disc1);
songs.put(disc2.getName(), disc2);
```

The above fragment associates the name "Banana Album" with the first CompactDisc object we created and the name "Taking Tiger Mountain" with the second object we created. We can visualize the contents of the table as follows:



We can retrieve items from our collection as follows:

```
CompactDisc aDisc = (CompactDisc)songs.get("Banana Album");
```

Note that we must perform a cast again, because a HashMap knows how to map key Objects to value Objects. HashMaps can map many keys to the same value, but cannot map a single key to many values. For instance:

```
HashMap songs = new HashMap();
CompactDisc disc1 = new CompactDisc("Banana Album", "Velvet Underground");
CompactDisc disc2 = new CompactDisc("Taking Tiger Mountain", "Brian Eno");
songs.put("Banana Album", disc1);
songs.put("Great Album!", disc1);
```

The above example maps the both the keys "Banana Album" and "Great Album!" to the same CompactDisc object (disc1). Extending our example, however:

```
HashMap songs = new HashMap();
CompactDisc disc1 = new CompactDisc("Banana Album", "Velvet Underground");
CompactDisc disc2 = new CompactDisc("Taking Tiger Mountain", "Brian Eno");
songs.put("Banana Album", disc1);
songs.put("Great Album!", disc1);
songs.put("Great Album!", disc2);
```

The final line *re-maps* the key "Great Album!" to disc2. A HashMap cannot by itself represent a one-to-many mapping of the sort we desire. Can you imagine combining a HashMap with an ArrayList to make this sort of mapping work?

What about the following example?

```
HashMap songs = new HashMap();
CompactDisc disc1 = new CompactDisc("Banana Album", "Velvet Underground");
CompactDisc disc2 = new CompactDisc("Taking Tiger Mountain", "Brian Eno");
songs.put(disc1.getName(), disc1);
songs.put(disc2.getName(), disc2);
CompactDisc anotherDisc = (CompactDisc)songs.get("Revolver");
```

What value is anotherDisc bound to? Remember that the songs contains only two mappings at this point, and neither of them has a key called "Revolver". Conceptually, the table should tell us that no such mapping exists at this point, and that is exactly what happens. There is a special value `null` in Java that is used to represent nothingness, and this value is returned by the get() method in this case.

We can exploit this fact to check if we have a given CompactDisc in our collection:

```
String title = input.readString("Title of the disc? ");
CompactDisc theDisc = (CompactDisc)songs.get(title);
if (theDisc != null) {
  output.println("We have it in stock!");
}
else {
  output.println("Sorry, please try later...");
}
```

Of course, by examining the interface to HashMap we see that this operation is supported directly, via the containsKey() method:

```
String title = input.readString("Title of the disc? ");
if (discs.containsKey(title)) {
  output.println("We have it in stock!");
}
else {
  output.println("Sorry, please try later...");
}
```

15. Introduction to Iteration

Key Concepts

1. Iteration
2. for loop (a counting loop)
3. while loop
4. loop equivalence

Introduction

There are many scenarios in our life where we repeat some actions for a period of time. Let's list some of them:

1. While there are still donuts in the box, eat one.
2. Send each relative that sent you money for your birthday a thank-you note.
3. Drive until you see a fork in the road.
4. Put each item of clothing on the floor in the laundry basket.
5. Give a cookie to each of your instructors.
6. Bake the roast until it has an internal temperature of 220 degrees.
7. Lather, rinse, repeat. (Buggy!)

In this lesson, we'll learn how to express algorithms like these in Java.

The for loop

Imagine that you're on contract to the weather station, and they want you to collect temperature statistics. In particular, they'd like you to find the average high temperature over a period of time. You already (sort of) know how to do this:

```
int totalTemp = 0;

int mondayHi = input.readInt("Monday's high temp?");
totalTemp = totalTemp + mondayHi;
int tuesdayHi = input.readInt("Tuesday's high temp?");
totalTemp = totalTemp + tuesdayHi;
int wednesdayHi = input.readInt("Wednesday's high temp?");
totalTemp = totalTemp + wednesdayHi;
   ...
int averageHi = totalTemp / 7;
```

This approach has some serious drawbacks. What are they? What we really want to say is something like:

```
For each day of the month (week or whatever period) get the high
temperature and accumulate it.  Afterwards, divide the total
temperature by the number of days in the period.
```

Java lets us express this algorithm almost verbatim:

```
int totalTemp = 0;
int numDays = 30;

for (int i=0; i < numDays; i=i+1) {
  int todaysHi = input.readInt("High temperature for day " + i);
  totalTemp = totalTemp + todaysHi;
}
int averageHi = totalTemp / numDays;
```

The for-loop in Java has the following pattern:

```
for (<initializer-statement>; <condition-expression>; <update-statement>) {
  <body>
}
```

The semantics is as follows: First, the `initializer statement` is executed. Then, while the `condition-expression` is true, the `body` statements, followed by the `update-statement` is executed.

Since so many for-loops contain statements like i=i+1 in their update step (indeed, incrementing by one is very common in other places too) you'll see the following shorthand used frequently:

```
i++;    // means i=i+1
```

The while loop

Now let's say we want to find the average rainfall over a period of consecutive days with rain. We might express the algorithm for doing this as follows:

```
While there is rain on a given day, accumulate the total rainfall.
Afterwards, calculate the average.
```

This problem is not naturally counting loop -- it loops simply until some condition is true. Java supports this kind of loop too:

```
int todaysRainfall = input.readInt("Today's rainfall?");
int daysWithRain = 0;

while (todaysRainfall > 0) {
  daysWithRain = daysWithRain + 1;
  totalRainfall = totalRainfall + todaysRainfall;
  todaysRainfall = input.readInt("Today's rainfall?");
}
int averageRainfall = totalRainFall / daysWithRain;
```

We can express this as a for loop, but it's not naturally a counting loop. It is worth convincing yourself, though, that every while loop can be expressed as a for loop, and vice-versa. For example, we could transform the above into a for loop:

```
int daysWithRain = 0;
for (int todaysRainfall = input.readInt("Today's rainfall?");
     todaysRainFall > 0;
     todaysRainfall = input.readInt("Today's rainfall?")) {
  daysWithRain = daysWithRain + 1;
  totalRainfall = totalRainfall + todaysRainfall;
}
int averageRainfall = totalRainFall / daysWithRain;
```

In general, it's better style to use a for loop when the loop is naturally described as a counting loop -- a loop that iterates over a sequence or range of numeric values.

16. Building a Software System

Key concepts

1. Basic program behavior: get input, do work, perform output
2. Example: A Banking Application
3. Wrapper Classes
4. Model-View-Controller Pattern (low calorie version)

Introduction

At this point, you know enough to build a pretty interesting program. Most programs - even though they may look very different on the surface - follow the following basic pattern of behavior. They get input from the user, do some work, and then produce some output. They probably repeat this cycle a number of times before the user decides to quit. We'll sometimes call a program a *system* to emphasize the point that it is in fact a system of interacting objects that work together to get some interesting job done. The art of building a robust, extensible, and reusable system is partitioning the work between the different objects in an appropriate manner. This partitioning takes experience and practice, but in some cases there are well known *patterns* that we can identify. These patterns are motifs or idioms that are shared by a great number of real-world programs. By identifying and understanding these patterns, we can integrate them into our toolbox of design strategies. In this lesson we'll introduce one classic pattern, known as the Model-View-Controller pattern, which will help us design classes that maximize flexibility and reuse.

The Banking Domain: Analysis

Let's imagine that we want to build a system that can be used by bank tellers to access and update customers' bank accounts. Let's think for a moment about some of the things that a bank teller might do in the course of his or her day:

1. Open a new account
2. Close an existing account
3. Deposit money into an account
4. Get the balance from an account
5. Withdraw money from an account
6. Print a list of all accounts

The above list of uses implies at least two important objects, a BankAccount object, and some sort of collection object that keeps track of all of the bank's accounts. We've already seen an example of a BankAccount class in a prior lesson. A BankAccount is a simple object that represents a single bank account. It aggregates the balance, owner name, and some other information about a single account. Here is an example implementation of a BankAccount class that we'll use in our system. It's identical to the implementation we built in prior lessons.

```
/**
  The BankAccount class implements a simple bank account.
  @author Ben Dugan
  @version 11/11/2001
*/
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  /**
    Create a new BankAccount object, with the given initial values.
    @param acctNumber an account number to use
    @param initBalance an initial balance
    @param acctName the name of the account owner
  */
  public BankAccount(int acctNumber, double initBalance, String acctName) {
    this.number = acctNumber;
    this.balance = initBalance;
    this.name = acctName;
  }

  /**
    Increase the balance by the given amount.
    @param amount the amount to deposit
  */
  public void deposit(double amount) {
    this.balance = this.balance + amount;
  }

  /**
    Answer the current balance.
    @return the current balance
  */
  public double getBalance() {
    return balance;
  }

  /**
    Answer the account number.
    @return the account number
  */
  public int getAccountNumber() {
    return number;
  }

  /**
    Decrease the balance by the given amount.
    @param amount the amount to deposit
    @return true if and only if there were sufficient funds for the withdrawal
  */
  public boolean withdraw(double amount) {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      return false;
    }
  }

  /**
    Transfer money between accounts.
    @param other a BankAccount to tranfer from
    @param amount an amount to transfer
  */
  public void transferFrom(BankAccount other, double amount) {
    other.withdraw(amount);
    this.deposit(amount);
  }
}
```

What about the collection of BankAccounts in the bank? Let's call this kind of object `Accounts`. What kind of object is it? The Accounts object is responsible for maintaining a collection of accounts for the bank. Is the collection ordered? Even though we probably access an account via its account number, this does not mean that there is an inherent order to the accounts. An account number is just a unique *key* used to access a single account. Whereas there may be two people named "Bill Wattie", they ought to have two separate bank accounts, each with their own unique account number. Let's use a HashMap that associates account numbers with BankAccounts as the fundamental representational component of our Accounts class:

```
public class Accounts {
  private HashMap accounts;

  /** Create a new, empty Accounts collection. */
  public Accounts() {
    this.accounts = new HashMap();
  }

  /** Add an account to the collection. */
  public void addAccount(BankAccount account) {
    int accountNumber = account.getAccountNumber();
    Integer key = new Integer(accountNumber);
    accounts.put(key, account);
  }

  /** Retrieve an account from the collection. */
  public BankAccount getAccount(int accountNumber) {
    Integer key = new Integer(accountNumber);
    return accounts.get(key);
  }
}
```

First, let's emphasize that this class looks a lot like the MusicCollection class. Its main job is to maintain a collection of things and provide a nice interface for adding and getting access to those things. There are some strange differences though. Can you spot them?

Wrapper Objects

There's something strange about the `getAccount` and `addAccount` methods. Both methods create an Integer object to use as a key for accessing the HashMap. The reason we do this is related to the discussion of casting in a prior lesson. Recall the interface for HashMaps:

```
public class HashMap {

  /** Answer the value associated with the given key. */
  public Object get(Object key);

  /** Associate the given key with the given value. */
  public void put(Object key, Object value);

  /** Remove the key and its corresponding value from the table.
      @return the value if it exists, null otherwise */
  public Object remove(Object key);
}
```

This interface tells us that HashMaps can associate any key Object with a value Object. Recall that the primitive types int, char, and double - unlike most other classes/types in Java - do not have the "is-a" relationship with the Object class. This exception is a famous "wart" in the definition of the Java language. The reason for this exception is related to efficiency considerations that worried the original implementors of the Java language. We *want* to implement a method like `getAccount` as follows:

```
public BankAccount getAccount(int accountNumber) {
  return accounts.get(accountNumber);
}
```

However, if we do this, the compiler will refuse to compile our program. Why? Because the type of `accountNumber` in the above example is `int` and the type `int` does not have an "is-a" relationship to the type expected by the `get` method on HashMaps, which is Object. To deal with this situation, the Java language provides us a set of classes whose only job it is to "wrap" primitive types such as char, int, and double. These classes are called Character, Integer, and Double, respectively. Once we wrap a primitive thing like an int in an Integer object, that object can be used in places where Objects are expected. Here's a fragment of the Integer class interface (interfaces to Character and Double are similar):

```
public class Integer {
  /** Create a new Integer containing the given value. */
  public Integer(int value);

  /** Answer the integer value of this object. */
  public int intValue();
}
```

In our Accounts class, we use Integer objects to wrap the account number value before we associate new BankAccounts with their account numbers, and before we look up a BankAccount by its number.

The User Interface

We have now implemented two fundamental classes for our banking world: BankAccount and Accounts. A BankAccount represents a single customer's account and an Accounts object maintains a collection of those BankAccounts so we can access them at later times. While these classes will form the basis for our banking application, we still need to do more work to make the system usable by a teller. In particular, we need to figure out how to handle the input and output to the system. For instance, the teller must input the account number into the system when he or she is looking up an account. What object in our system should be responsible for handling the input, and producing the output?

As usual, we have several options for assigning these responsibilities. The first and perhaps most obvious option is to assign them to the Accounts object. We could, for instance, add a method to the Accounts class that prompts the user for an account number, then looks up that account, and prints the result to the screen. The method might look like this:

```
public void printAccount(Input in, Output out) {
  int number = in.readInt("Enter account number: ");
  BankAccount account = this.getAccount(number);
  out.println(account);
}
```

This turns out to not be a very good idea. Why? Because it violates a number of important principles of software design.

First, it violates what we might call the "Hedgehog Principle". The name "Hedgehog Principle" comes from the saying "The fox knows many things, but the hedgehog knows one great thing." The hedgehog is, above all, an expert at rolling itself into a ball to defend itself. The fox, for all of its general cunning cannnot defeat the expert hedgehog. Taken to a programming context, this principle states that classes should prefer to be excellent at doing one thing, rather than be pretty good at a variety of things. By adding a method like `printAccount` to the Account class, we're asking it to be good at both doing input and output, as well as managing a collection of BankAccounts.

Second, the above design does not typically result in maintainable software. Imagine trying to ship the above software to a country where another language is spoken. We've hardcoded the English language prompt into the definition of the Accounts class. To ship the software to Germany, for instance, we'd have to make a copy of the class (call it something like GermanAccounts), and then go change the prompts in the GermanAccounts class. While this approach would work, what happens when we discover a bug in the original Accounts class? We'd have to fix the bug in the Accounts class, as well as GermanAccounts, FrenchAccounts, so on. Doing this in practice would be a maintinence nightmare.

Finally, it does not partition the work between different programmers very well. In a real system, getting the user interface right is a big job, and we'd like to allow some programmers to focus just on building an excellent interface, while other programmers might focus on building the underlying system objects, such as BankAccount and Accounts.

These issues encourage us to create a separate class whose job it was only to be an expert at performing input and output. We'll call this class BankUI (short for BankUserInterface). It will have methods that support the following typical bank teller activities:

1. Open a new account
2. Close an existing account
3. Deposit money into an account
4. Get the balance from an account
5. Withdraw money from an account
6. Print a list of all accounts

Let's look at how the BankUI class might be implemented:

```
public class BankUI {
  private Accounts accounts;
  private Input input;
  private Output output;
  private nextAccountNumber;

  /** Create a new BankUI for the given objects. */
  public BankUI(Accounts accounts, Input in, Output out) {
    this.accounts = accounts;
    this.input = in;
    this.output = out;
    this.nextAccountNumber = 0;
  }

  /** Create a new account */
  public void createAccount() {
    int number = nextAccountNumber;
    nextAccountNumber = nextAccountNumber + 1;
    double balance = input.readDouble("Enter starting balance: ");
    String name = input.readString("Enter account name: ");
    BankAccount account = new BankAccount(number, balance, name);
    accounts.addAccount(account);
    output.println("Account# " + number + " created for " + name);
  }

  /** Get the balance from an account. */
  public void checkBalance() {
    int number = input.readInt("Enter account number: ");
    BankAccount account = accounts.getAccount(number);
    output.println("The balance is " + account.getBalance());
  }

  /** Deposit money into an account. */
  public void depositMoney() {
    int number = input.readInt("Enter account number: ");
    BankAccount account = accounts.getAccount(number);
    double amount = input.readDouble("Enter amount: ");
    account.deposit(amount);
    output.println("New balance: " + account.getBalance());
  }

  /** Withdraw money from an account. */
  public void withdrawMoney() {
    int number = input.readInt("Enter account number: ");
    BankAccount account = accounts.getAccount(number);
    double amount = input.readDouble("Enter amount: ");
    boolean success = account.withdraw(amount);
    if (success) {
      output.println("Withdrawal succeeded.");
    }
    else {
      output.println("Withdrawal failed.  Insufficient funds.");
    }
  }
}
```

Driving the interface

We now have a class that is pretty good at prompting the user (the bank teller) for the basic information (account numbers and dollar amounts) and then dispatching the work (looking up accounts, changing their balance, etc) to the objects that can handle those operations (the Accounts collection or individual BankAccounts). Let's look at how we might use the interpreter to test the system we've built so far:

```
prompt> Input in = new Input();
prompt> Output out = new Output();
prompt> Accounts accounts = new Accounts();
prompt> BankUI ui = new BankUI(accounts, in, out);
prompt> ui.createAccount();
Enter starting balance:
200.50
Enter account name:
Ahab
Account# 0 created for Ahab
prompt> ui.depositMoney();
Enter account number:
0
Enter amount:
50.0
New balance: 250.50
prompt>
```

Notice that the first four lines are setup code. We need to create our basic system objects, and then we stitch them together with a BankUI object. After that, we send messages to the BankUI object and it handles the input and output required to access and update BankAccount objects.

In real life, we wouldn't expect the teller to actually type Java code to send messages to the BankUI. We can actually internalize the calling of the BankUI methods in the class itself:

```
public class BankUI {

  // All the other methods...

  /** Print a menu of options. */
  public void printMenu() {
    output.println("Please enter one of the following options:");
    output.println("  [n]ew account");
    output.println("  [d]eposit");
    output.println("  [w]ithdraw");
    output.println("  [q]uit");
  }

  /** Prompt teller and dispatch to appopriate method.
      @return false if and only if the teller quit */
  public boolean dispatch() {
    char option = input.readChar("Enter Option: ");
    boolean quitting = false;
    if (option == 'n') {
      this.createAccount();
    }
    else if (option == 'd') {
      this.depositMoney();
    }
    else if (option == 'w') {
      this.withdrawMoney();
    }
    else if (option == 'q') {
      output.println("Goodbye!");
      quitting = true;
    }
    else {
      output.println("Invalid option.");
    }
    return quitting;
  }

  /** Run the UI until the teller decides to quit. */
  public void go() {
    boolean quitting = false;
    while (!quitting) {
      printMenu();
      quitting = dispatch();
    }
  }
}
```

We've added three methods to our BankUI class. The first prints a menu of options that are available to the bank teller. The second consumes a character from the teller (by which he or she selects an option), and calls the appropriate method. The third implements a loop, that continually prints the menu and dispatches the chosen option to the right method. Let's use this new class in our interpreter:

```
prompt> Input in = new Input();
prompt> Output out = new Output();
prompt> Accounts accounts = new Accounts();
prompt> BankUI ui = new BankUI(accounts, in, out);
prompt> ui.go();
Please enter one of the following options:
  [n]ew account
  [d]eposit
  [w]ithdraw
  [q]uit
```

```
Enter option:
n
Enter starting balance:
200.50
Enter account name:
Ahab
Account# 0 created for Ahab

Please enter one of the following options:
  [n]ew account
  [d]eposit
  [w]ithdraw
  [q]uit
Enter option:
d
Enter account number:
0
Enter amount:
50.0
New balance: 250.50

Please enter one of the following options:
  [n]ew account
  [d]eposit
  [w]ithdraw
  [q]uit
Enter option:
x
Invalid option.

Please enter one of the following options:
  [n]ew account
  [d]eposit
  [w]ithdraw
  [q]uit
Enter option:
q
Goodbye!

prompt >
```

Notice that the first four lines are identical to the original example -- we create and stitch together the system objects. After that, however, we make a single call to the `go` method on the BankUI object, and this method executes a loop which handles the interaction with the teller. In a way, the `go` method implements a simple interpreter that forever does the following three things: reads an option from the teller, executes that option, and then prints the result. This read-evaluate-print interpreter cycle is another common pattern in many real-world systems.

What's missing

The above system, consisting of three classes (Accounts, BankUI, and BankAccount), implements a simple application that could be used by a bankteller to access and update customers' bank accounts. It should be pointed out that it is far from complete, however. In particular, several issues remain unresolved:

1. When the teller quits the application, all account information is lost. We'd like the application to be able to "open" and "save" account information from a file on a disk.
2. What about non-existent accounts? Right now, we are assuming that the teller always enters a valid account number. What happens if an invalid account number is entered?
3. In the real world, several people can share one account. Does the current system handle this situation? If not, how might we modify one or more of the classes to handle this case.
4. People often forget their account numbers. We'd like a way to access accounts by a key other than account numbers. Can you think of another useful key and how we might modify the system to handle this situation?

Clearly, building a robust, user-friendly system takes a lot of work. Error checking and handling the wide range of real-world user activities are all part of the art of building usable software systems. Even though the above example only implements a fraction of the activities a real banking system should support, it should give you the flavor of what goes into building such a system.

The Model-View-Controller Pattern

In the above example, we have covered a great deal of territory. We've implemented a class to provide a simple facade for a collection of BankAccounts, the Accounts class. Implementing this class required a detour into the world of Java wrapper classes. We also implemented a class - the BankUI - that is responsible for handling the user-interface related tasks of our system. In doing so, we separated the user interface from the basic system objects. This separation is a primitive example of a fundamental design pattern known as *Model-view-controller* (MVC for short). This pattern posits three fundamental components to any system with a user interface. The *Model* is the set of objects that implements the basic model of the world. It does not perform any input or output. Rather, the input is handled by a component called the *controller*, and the output handled by a component called the *view*.

While we have combined the jobs of the view and the controller in the BankUI object, we are still faithful to the basic distinctions of the MVC pattern. By decoupling the components in this manner, we gain a great deal in terms of flexibility and reusability. For instance, assume that we need to implement a fancier version of our user interface. In this day and age, text-driven user interfaces are not very popular anymore. Most user interfaces are graphical in nature, and rely heavily on mouse inputs and menus for option selection. Given the above design, we can reuse the BankAccount and Accounts classes without modification, and simply implement a new class (perhaps called GraphicalBankUI) to manage the appropriate display objects and handle mouse inputs from the user. Furthermore, we can mix and match our user interface components in an interesting manner. We can, for instance, start our development by building a simple text-driven user interface, and then later start working on a graphical user interface. During the development and testing of this more complicated interface, we can still rely on the old text-driven interface to test-drive our system, or provide access to features that are not yet available in the nascent graphical user interface.

---

17. Iterating Over Collections

Key concepts

1. Ordered (Indexed) Collections
2. Unordered Collections
3. Iteration Abstracted
4. Common Iteration Idioms

Introduction

Let's take another look at the CompactDisc class we developed in a previous lesson. Suppose we want to add a method to the CompactDisc class that will print out the name of every song on the disc. We can say how we would do this in English:

```
Walk through the collection of songs, printing the name of each one.
```

In this lesson we'll apply the concept of iteration to the management and processing of collections.

Ordered Collections

Let's start by revisiting the way that we can access the items of a ArrayList. Suppose we have an ArrayList that we know contains three Strings and we want to print out each of them. We can do this as follows:

```
output.println(nameList.get(0));
output.println(nameList.get(1));
output.println(nameList.get(2));
```

This works, but is tedious if the list contains many items. More problematic, however, is the fact that it won't work if we don't know when we write our program how many items the list contains. The solution is to use *iteration*, which is a way to execute a variable number of statements. Since we know how to ask an ArrayList how many items it contains, we can write a counting loop that accesses each item of our ArrayList sequentially, from zero up to the number of items in the ArrayList:

```
int numNames = nameList.size();
for (int i=0; i<numNames; i++) {
  output.println(nameList.get(i));
}
```

In English, this fragment of code implements the following algorithm:

```
For each item of the list, print it.
```

Remember that the items in the ArrayList live at indices 0 to size-1. A very common error is to iterate over items 0 to size:

```
int numNames = nameList.size();
for (int i=0; i<=numNames; i++) {
  output.println(nameList.get(i));
}
```

Unordered Collections: The Easy Way

What about iterating over the items in an unordered collection? Remember, the items in an unordered collections are just that, unordered -- that is, there is no first, last, or in general, i-th element. However, if we can find a way to make an ordered collection out of the items in our unordered collection, then we can reuse the pattern we learned above. By inspecting the full interface for some of the Java collection classes, we see that we can do the following:

```
HashMap someTable = new HashMap();
// assume we add some items here...

// the keySet() method on HashMaps returns a Set of the keys of the map
Set keys = someTable.keySet();

// now we can create and initialize a new ArrayList with that collection
ArrayList list = new ArrayList(keys);

// and now we can iterate
for (int i=0; i<list.size(); i++) {
  // Access the values of the map with:  someTable.get(list.get(i))
}
```

In other words, we ask the HashMap for a Set of its keys and then create a new ArrayList (initialized by the elements of that collection) and exploit its indexability to access the values of the HashMap. While this is a roundabout method of approaching the problem, it does work, and it requires only that we understand a single pattern for iterating over collections.

Iteration Abstracted: an Iterator

It turns out that there is a more general purpose way to iterate over collections in Java, achieved by a decoupling of the concept of iteration from the collection itself. To support a uniform model of iteration over the items of any kind of collection in Java, we may ask a collection for an object that allows us to sequentially access the items in that collection. In general, such an object is called an *iterator*. Java provides a couple of different kinds of iterators, but we'll just look at one called, appropriately enough, Iterator:

```
public interface Iterator {

  /** Answer the next Object and advance the iterator. */
  public Object next();

  /** Answer true if the iterator has more elements. */
  public boolean hasNext();
}
```

We can ask a HashMap for an iterator enumeration of all of its keys as follows:

```
// Create a HashMap and add some mappings
HashMap map = new HashMap();
map.put("Willa", "Cather");
map.put("Raymond", "Carver");

// Now get the Set of its keys
Set keys = map.keySet();

// Now get an iterator over this set:
Iterator iter = keys.iterator();
```

Since an Iterator does not provide us with a notion of size, or indexed access, it is not natural to use a counting loop when processing an Iterator. A while-loop, however, will do the trick nicely:

```
// Create a HashMap and add some mappings
HashMap map = new HashMap();
map.put("Willa", "Cather");
map.put("Albert", "Camus");
map.put("Raymond", "Carver");

// Now get the iterator and iterate over it
Iterator iter = map.keySet().iterator();

while (iter.hasNext()) {
  Object aKey = iter.next();
  output.println(aKey + " " + map.get(aKey));
}
```

In English, the above fragment expresses the following algorithm:

```
First, get all of the keys in the map.
Next, while there are still keys, get the next key and print out the
  key and the associated value from the map.
```

If we ran the program, the output might look like this. Notice that the order presented here bears no relation to the order in which these things were originally inserted into the table -- the HashMap makes no guarantees about the order in which is stores its mappings.

```
Raymond Carver
Albert Camus
Willa Cather
```

Finally, by inspecting the interface for the class ArrayList, we see that ArrayLists are also willing to provide us with Iterators we can use to iterate over their contents. Here's an example:

```
ArrayList names = new ArrayList();
names.add("Bob");
names.add("Bill");
names.add("Susan");

// Now get an Iterator and iterate over it
Iterator iter = names.iterator();

while (iter.hasNext()) {
  Object item = iter.next();
  output.println(item);
}
```

The great thing about using iterators is that we learn one pattern for iterating over the items inside of a collection and can reuse that style of iteration for all of Java's collection classes.

Common Idioms

At this point, we're ready to classify common patterns of iteration over collections. It turns out that perhaps 90% of loops over collections fall into one of three categories: *simple traversal*, *reduction*, and *filtering*.

The loops we've seen so far are examples of simple traversals. They simply walk over the collection, doing something to or with each item. We can write the pattern as follows:

```
Iterator iter = <collection>.iterator();
while (iter.hasNext()) {
  Object item = iter.next();
  <do something with the current item>
}
```

A reduction is a loop that aims to reduce, or distill some piece of information out of the elements in the collection. Examples include finding the largest (or smallest) element, finding a specific element, summing the values of the elements. Suppose we have an ArrayList called `names` that contains Strings, and we wish to find the longest one:

```
String longest = null;
Iterator iter = names.iterator();
while (iter.hasNext()) {
  String name = (String)iter.next();
```

```
    if (longest == null || longest.length() < name.length()) {
      longest = name;
    }
  }
```

Notice the null check in the if statement inside of the loop. On the first iteration, we have not yet determined the current longest string, so we must check for that. Suppose we want to sum the lengths of all of the Strings in the array:

```
int totalLength = 0;
Iterator iter = names.iterator();
while (iter.hasNext()) {
  String name = (String)iter.next();
  lotalLength = totalLength + name.length();
}
```

Both of these loops "boil down" the items in the collection, reducing them to a single piece of information, be it the longest element or the sum of the sizes, or whatever we choose. Here is the pattern for a reduction loop:

```
<set up initial bindings for the essence>
Iterator iter = <collection>.iterator();
while (iter.hasNext()) {
  Object item = iter.next();
  <rebind the essence based on the current item>
}
```

A final idiom for collection iteration is one that filters out certain elements of interest. Generally this means collecting the elements that pass a test into a new collection. Suppose we want to collect all of the elements in our ArrayList of Strings that are even in length:

```
ArrayList evens = new ArrayList();
Iterator iter = someArrayList.iterator();
while (iter.hasNext()) {
  String name = (String)iter.next();
  if (name.length() % 2 == 0) {
    evens.add(name);
  }
}
```

The above loop tests every element for "even-ness". If the element is even, it is added to a second collection, called "evens". In general, filtering loops tend to look like this:

```
ArrayList passedTheTest = new ArrayList();
Iterator iter = <collection>.iterator();
while (iter.hasNext()) {
  Object item = iter.next();
  <conditionally add item to the passedTheTest collection>
}
```

When writing a loop that iterates over a collection, it's useful to ask yourself "Does the iteration fit into a pattern I already know?" Chances are that it will. Understanding and practicing these few patterns will make writing most collection iterations a breeze.

---

18. Arrays

Key concepts

1. Arrays as basic collections; array types
2. Creating Arrays
3. Indexed Access
4. Iterating over arrays

Introduction

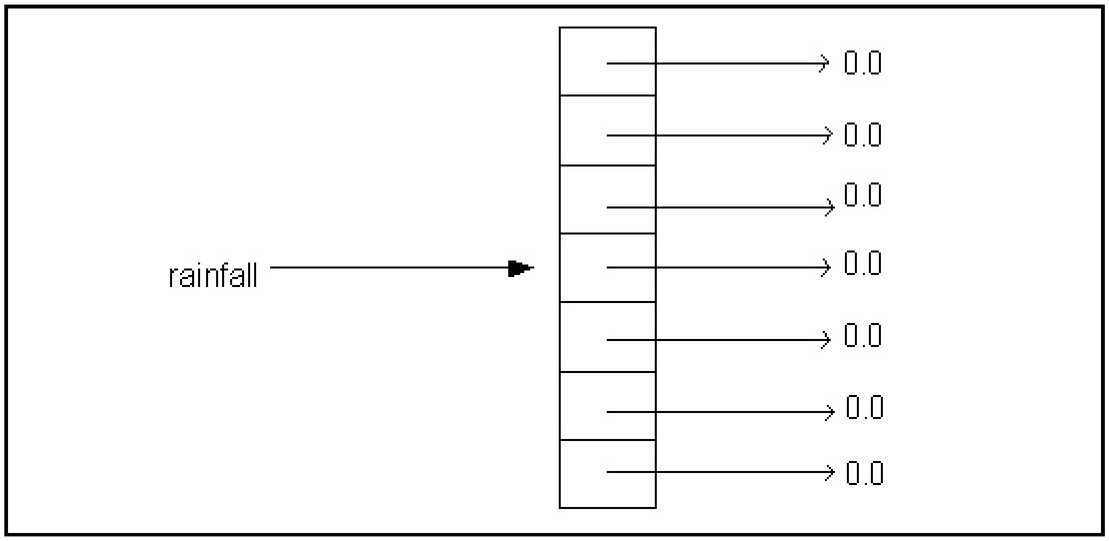
So far we've seen two powerful collection classes in Java -- the ArrayList and HashMap. In this lesson, we'll see a more primitive mechanism for representing a collection, called an *array*. In many programming languages (particularly older languages, such as C), arrays are the only mechanism for representing collections of data, and programmers typically use arrays to build higher-level collection abstractions that are more convenient and flexible to use.

Creating Arrays

Arrays are a very simple kind of collection. They are fixed in size and provide indexed access. For example, let's say we want to store the rainfall data for a week's rainfall. Here's how we can create an array of seven numbers, one for each day:

```
double[] rainfall = new double[7];
```

`double[]` is the type of the array -- an array of doubles. `rainfall` is the name that we're going to use to refer to the new array object (it can be any identifier we like). `7` is the size of the array that we're going to build (it can be any expression). `new double[7]` creates the new array-of-doubles object, with the given length. We can draw the resulting picture as follows:

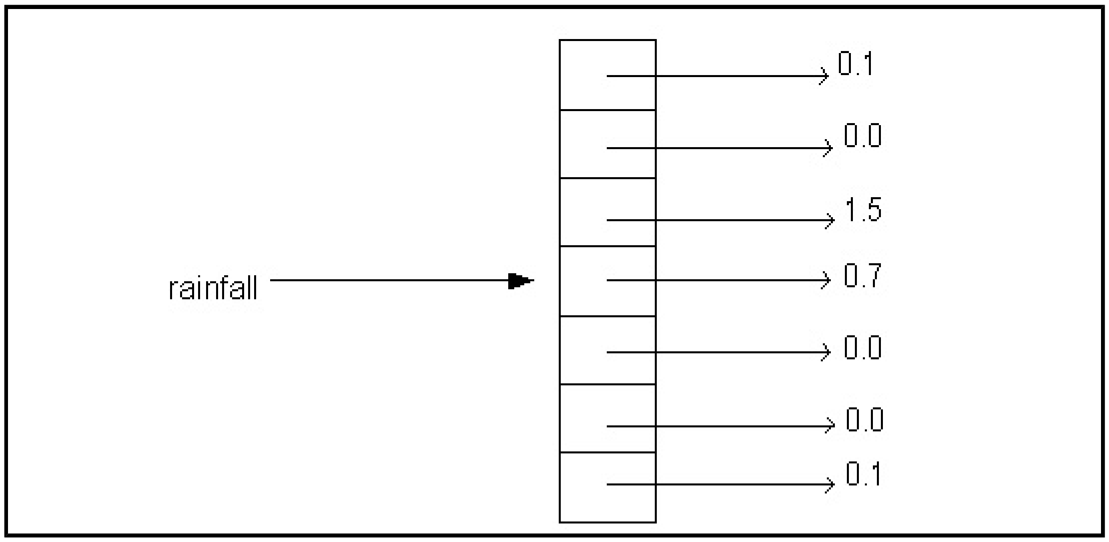


Indexed Access

Now we'd like to put the rainfall data in it. To assign to the *elements* of the array, we use *indexing*. Indexing into arrays is very similar to indexing into ArrayList objects, except that the syntax is different. Here's how we might set the rainfall data:

```
rainfall[0] = 0.1;  // Sunday's rainfall
rainfall[1] = 0.0;  // Monday's rainfall
rainfall[2] = 1.5;  // Tuesday's rainfall; a real gusher
rainfall[3] = 0.7;  // Wednesday's rainfall
rainfall[4] = 0.0;  // Thursday's rainfall
rainfall[5] = 0.0;  // Friday's rainfall
rainfall[6] = 0.1;  // Saturday's rainfall
```

The number in brackets (0 - 6) is the index. Like ArrayList objects, the first element of an array is always at index 0, and other elements follow sequentially after that. So for an array of size N, the indices of the elements are in the range 0..N-1.



We use indexing to get things back out of the array, too:

```
double mondaysRainfall = rainfall[1];
double weekend_rainfall = rainfall[0] + rainfall[6];
```

Notice that we do not need a cast operation (as we did when we accessed elements of an ArrayList), because we expressed the type of the array elements when we created the array. In general, the indices can be any expression, so we can compute which elements we want to store into or fetch from. For example:

```
rainfall[x + y - z] = rainfall[x - y * z] * 5;
```

We can make arrays of any kind of object, not just numbers. Here's an array of Rectangles:

```
Rectangle[] rectangles = new Rectangle[4];
rectangles[0] = new Rectangle(0, 0, 100, 200);
rectangles[1] = new Rectangle(200, 200, 10, 20);
rectangles[2] = new Rectangle(300, 300, 100, 40);
rectangles[3] = new Rectangle(100, 100, 10, 20);
```

Again, we use indexing to access elements of our array:

```
Rectangle aRectangle = rectangles[2];
```

In general, we use the following pattern for accessing an array:

```
<array-name>[<index-expression>]
```

Iterating Over Arrays

As with other collections, if we wish to visit every element of an array, we'll need to use iteration. For example, here's how we can compute the total rainfall for a week:

```
// assume we've created and filled an array of rainfall data

int totalRainfall = 0;

for (int day = 0; day < 7; day++) {
  // day is in the range 0..6 here
  double dayRainfall = rainfall[day];
  totalRainfall = totalRainfall + dayRainfall;
}

// totalRainfall now has the right answer
```

In the loop body, `rainfall[day]` looks up the rainfall for the particular day.The loop makes this variable go through all the elements of the array. A standard pattern for looping over an array is

```
for (<indexName>=0; <indexName> < <array-length>; <indexName>++) {
  ... array[<indexName>] ...
}
```

In this example, we knew that the rainfall data was for a week, so we used the constant 7 as the upper limit of the loop. But in the future, we might change it to be for a month, or a year, or anything else. To make our code be more adaptable to future changes to the size of the rainfall array, as well as to make the code clearer to the reader ("what does 7 mean?", they might ask), we can change it to use a final Java feature of arrays: asking their length.

The expression `a.length` evaluates to the length of the array a. We can use this expression as the upper limit of the for-loop:

```
// assume we've created and filled an array of rainfall data

int totalRainfall = 0;

for (int day = 0; day < rainfall.length; day++) {
  double dayRainfall = rainfall[day];
  totalRainfall = totalRainfall + dayRainfall;
}

// totalRainfall now has the right answer
```

This is a much better implementation of this loop, since it will automatically adapt to some kinds of program changes.

An even better idea is to anticipate that we might be wanting to calculate sums of arrays of numbers in other applications too, and so implement a function just for that purpose:

```
// in some class, e.g. ArrayFunctions
public static double sum(double[] array) {
  int total = 0;
  for (int index = 0; index < array.length; index++) {
    // index is in the range 0..(array.length-1) here
    total = total + array[index];
  }
  return total;
}
```

This function can then be called to compute the total rainfall data for a week:

```
int totalRainfall = ArrayFunctions.sum(rainfall);
```

We've now created a method that is readable and robust to future changes. Furthermore, it can be debugged once and then reused in lots of other contexts.

19. Using Arrays

Key concepts

1. When to use arrays instead of ArrayLists
2. A non-trivial application of arrays

Motivation

As we pointed out earlier, arrays are a primitive kind of collection. When we create an array, we need to know its size before we do so. Suppose for instance we want the user to input a bunch of weather data. We must allocate the array before the user starts to input the data, but how do we know how much data the user intends to input before they are finished doing so? One solution is to ask the user *before* they input any data how many items they intend to input, as follows:

```
int size = input.readInt("How many items?");
double[] rainfall = new double[size];

// now get the items:
for (int day=0; day < size; day++) {
  rainfall[day] = input.readInt("How much rainfall on day " + day);
}

// now the array is full of valid data...
```

The above works, but if we used an ArrayList, we wouldn't need to worry about "sizing" the ArrayList before adding items to it. ArrayLists resize their capacity dynamically. ArrayLists also allow us to add items to the ArrayList at any location. If we add an item to the middle or front of the ArrayList, the other elements of the ArrayList are shifted appropriately. This is not the case with arrays.

Despite the apparent limitations of arrays, they do provide us with a fundamental building block for creating higher level abstractions. Furthermore, accessing elements in an array is more efficient than the equivalent operations on an ArrayList. While these efficiency considerations should not concern us, they are of concern to programmers who must interface with the services of the system (such as the disk, network devices, audio resources, and so on).

Building our own (simple) ArrayList

To show an application of arrays, imagine that Java does not provide us with the ArrayList class. Using only arrays, we can build our own ArrayList class. We'll implement just the following portion of the interface:

```
public class SimpleArrayList {
  /** Create a new, empty SimpleArrayList. */
  public SimpleArrayList();
  /** Add the given object to the end of the list. */
  public void add(Object o);

  /** Answer the object that lives at the given index. */
  public Object get(int i);

  /** Answer the number of elements. */
  public int size();
}
```

Our strategy will be to use an array of Objects internally in our SimpleArrayList class. The array will be used to hold the items that have been added to the SimpleArrayList. For this to work we must also remember how many elements of the array have been set, so that subsequent adds will set the appropriate array element. Let's look at a first cut of an implementation:

```
public class SimpleArrayList {
  private Object[] items;      // the elements
  private int      size;     // the number of valid elements

  /** Create a new, empty SimpleArrayList. */
  public SimpleArrayList() {
    this.items = new Object[4];
    this.size = 0;
  }

  /** Add the given object to the end of the list. */
  public void add(Object objectToAdd) {
    this.items[this.size] = objectToAdd;
    this.size = this.size + 1;
  }

  /** Answer the object that lives at the given index. */
  public Object get(int index) {
    if (index >= 0 && index < this.size) {
      return this.items[index];
    }
    else {
      return null;
    }
  }

  /** Answer the number of elements. */
  public int size() {
    return this.size;
  }
}
```

The above implementation works, but only to a point. What happens after we add four elements to the SimpleArrayList? When we attempt to add a fifth element, we'll run out of room in our items array. When this occurs, we need to make more room. To do so, we'll allocate a new array which is twice as large as the original array, set the corresponding elements of the new array to those of the `items` array, and then rebind the name items to the new array. Let's look at how this works:

```
public class SimpleArrayList {
  private Object[] items;      // the elements
  private int      size;       // the number of valid elements

  /** Create a new, empty SimpleArrayList. */
  public SimpleArrayList() {
    this.items = new Object[4];
    this.size = 0;
  }

  /** Grow the array, by doubling its size */
  public boolean grow() {
    Object[] newItems = new Object[this.items.length * 2];
    for (int index=0; index < this.items.length; index++) {
      newItems[index] = this.items[index];
    }
    this.items = newItems;
  }

  /** Add the given object to the end of the list. */
  public void add(Object objectToAdd) {
    if (this.size == this.items.length) {
      grow();
    }
    this.items[this.size] = objectToAdd;
    this.size = this.size + 1;
  }

  /** Answer the object that lives at the given index. */
  public Object get(int index) {
    if (index >= 0 && index < this.size) {
      return this.items[index];
    }
    else {
      return null;
    }
  }

  /** Answer the number of elements. */
  public int size() {
    return size;
  }
}
```

Notice that we've just added a private method that handles the growing of the internal array. It allocates an array twice the size of the old one, sets the elements appropriately, and then rebinds the name `items`.

The SimpleArrayList class is far from complete. We'd at least like to be able to add an item into the middle of the collection. Imagine how you would do this. First, such a method would perhaps have to grow the items array. Second, it would also need to deal with shifting those items to the "right" (at higher indices) of the insertion point to the right.

20. Multidimensional Arrays

Key concepts

1. Two Dimensional Arrays
2. Multi-Dimensional Arrays

Introduction

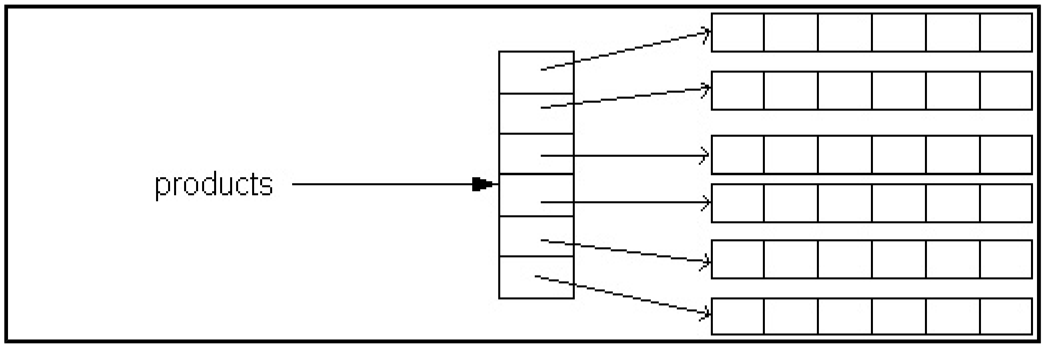
Arrays give us a linear, one-dimensional sequence of data. We may want, however, to represent to two dimensional grid of data. For instance, suppose we want to represent a multiplication table for numbers between 0 and 5. We can get this effect by using an array whose elements are themselves arrays.

Creating a 2D Array

We can create a two dimensional array as follows:

```
int[][] products = new int[6][6];
```

The type of `products` is an array of an array of integers. We can visualize it as follows:



Notice that the name `products` refers to an array. That array is an array of arrays of integers. `products[0]` refers to the first array of integers. We'll often talk about two dimensional arrays in terms of *rows* and *columns*. The first dimension selects the row -- `products[0]` selects the array that is the first row. Here are some examples of getting and setting values in the array:

```
int[][] products = new int[6][6];

int[] secondRow = products[1];          // the second row
secondRow[0] = 0;                       // set some values
secondRow[1] = 1;
secondRow[2] = 2;

// This is legal shorthand: set the 4th element of the 3rd row
```

```
products[3][4] = 12;
```

Again, we will often use loops to visit all of the elements in our array. Because each row is really another array, we will usually nest our loops in order to visit each element. The outer loop visits every row, and the inner loop visits every column element in the current row.

```
for (int row=0; row<6; row++) {       // visit each row
  for (int col=0; col<6; col++) {     // visit each column of the current row
    products[row][col] = row*col;
  }
}
```

Again, we can rewrite this loop to be more resilient in the face of changes to the dimensions of our array, by using the length field of our array:

```
for (int row=0; row < products.length; row++) {
  for (int col=0; col < products[row].length; col++) {
    products[row][col] = row*col;
  }
}
```

Finally, it's worth pointing out that there are no limits to the number of dimensions our arrays have. A weather application that models temperature in the atmosphere must be able represent three dimensions (essentially latitude, longitude, and altitude). We can create an array to represent this data as follows:

```
double[][][] temperatures[150][100][120];
```

The above fragment creates a three dimensional array which we can think of as 150 two dimensional arrays, each of which is 100 one-dimensional arrays of length 120.

21. Exceptions

Key concepts

1. Handling Errors
2. Error Codes
3. Exceptions

Introduction

Strategies for handling errors depend largely on the nature of the application. Since beginning programming courses typically deal in trivial applications, error handling is often dismissed as an unimportant subject. In real world applications, however, doing the "right thing" in the face of errors is critically important.

There are a variety of strategies for handling errors at run time. These include doing nothing, notifying the user, halting the program, and a host of other possible actions. Depending on the type of application, various of these approaches may be valid.

At one extreme, imagine a program that averages weather data over extremely large data sets. The program designers probably expect out of range or missing data, and would design their program to be silently tolerant of such errors, perhaps reporting the number of missing or invalid data items. Halting the program or issuing a warning for every invalid data item would certainly be overkill in this case.

In the middle ground, imagine an ATM application. If for instance, the user attempts to overdraw your bank account, halting the ATM program (and shutting down the ATM) is probably not the right approach. Silently ignoring the problem is probably not the right strategy either. Ideally, the program should give them a warning, and not modify their account. It may then want to give the user another opportunity to withdraw less money.

At the other extreme, there exist applications whose failure can put many lives at risk. Imagine a program that controls a nuclear reactor or assists air traffic controllers. If the program that controls a nuclear reactor encounters an input that is out of the expected range, it may mean that something very serious has happened. Silently ignoring it is certainly not an option. Notifying human operators is certainly of utmost importance. Shutting down the program is perhaps even the right strategy, depending on how the entire control system has been designed.

It's the job of a class designer and implementor to decide what to do about errors, and how to report and handle them if necessary. The conditions that are considered errors, and the behavior of classes of objects in the face of these errors are important details of a class interface.

One approach: Error codes

Recall the BankAccount class. We handled an insufficient funds condition by returning an "error code". Here's the relevant fragment of the class definition for the BankAccount:

```
public class BankAccount {
  private int number;
  private double balance;
  private String name;

  /**
    Decrease the balance by the given amount.  If insufficient funds,
    the balance is unchanged.
    @param amount the amount to withdraw
    @return true if and only if sufficient funds were available
  */
  public boolean withdraw(double amount) {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      return false;
    }
  }
}
```

Also, recall the example of how we can use the error code that is returned in order to deal with errors such as these:

```
BankAccount account = new BankAccount(1234, 225.34, "Bill");
Input input = new Input();

double amount = input.readDouble("How much money do you want?");
boolean succeeded = account.withdraw(amount);

if (succeeded) {
  System.out.println("Here's your money!!");
}
else {
  System.out.println("Sorry, insufficient funds!");
}
```

The error code places the onus of dealing with the error onto the client of the BankAccount class. This is fundamentally the right strategy, but has shortcomings: First, it means that to really do the right thing, client code must constantly capture and check the returned error code for every operation. Unfortunately, programmers are fundamentally pretty lazy people, and error codes are frequently ignored. Second, if many methods in a class return error codes, it becomes tedious to check for them, each and every time they might be meaningful.

A Better Approach: Exceptions

Exceptions allow an implementor of a class to defer the responsibility of handling an error to the client. In this way, they are similar to error codes. However, exceptions have two important advantages over error codes. First, the compiler will enforce handling of the error. This means that exceptions, unlike error codes, cannot be ignored. Second, in handling exceptions, clients may wrap up segments of potentially error generating code and handle classes of errors in one central location.

First, let's look at an example of *raising* or *throwing* an exception.

```
public class BankAccount {

  // ... other stuff ...

  /**
    Decrease the balance by the given amount.  If insufficient funds,
    throws an Exception.
    @param amount the amount to withdraw
  */
  public void withdraw(double amount) throws Exception {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      throw new Exception("Insufficient Funds");
    }
  }
}
```

The above example introduces two new keywords, `throw` and `throws`. Think of `throw` as similar to the keyword `return`. It alters the flow of control of the program. After `throw` we must provide an expression that evaluates to an exception object. In Java, exceptions are represented as objects.

The keyword `throws` modifies the signature of the method, advertising to the world that this method *may*, under some circumstances throw an exception. If the body of the method contains a throw statement, or calls a method that may throw an exception that does not get handled inside of the body of the method, then the compiler will insist that this exception throwing behavior be advertised in the method signature. By this manner, the compiler can and does enforce the handling of exceptions in client code.

Handling Exceptions

We've seen an example of throwing an exception, so now let's look at how we might handle one:

```
BankAccount account = new BankAccount(1234, 225.34, "Bill");
Input input = new Input();

double amount = input.readDouble("How much money do you want?");

try {
  account.withdraw(amount);
  System.out.println("Here's your money!!");
  // give the user their money....
}
catch (Exception anException) {
  System.out.println("Sorry, insufficient funds!");

}
```

The new bit of syntax that we're introducing here is called a `try-catch` block. We can use the try-catch block to wrap up a method call that may throw an exception. If an exception is thrown by any statement in that block, control passes to the `catch` block, where we can place code to handle the exception.

Think about what actually happens if the exception is thrown by the withdrawal operation. Do the subsequent lines in the try block get excecuted? We would hope not, because this would mean giving the user money that didn't exist in their account. This behavior might make you and me happy, but not the banker. In fact, the subsequent lines do not get executed, because the thrown exception causes the control of the execution of the program to change. Code is no longer excecuted sequentially; rather, the execution of the program is literally suspended and control is passed to the enclosing catch block. This is substantially different from the standard form of program execution. But if you think about it, not at all unreasonable: erroneous conditions should cause our program to halt normal operation and do some work to fix or report the condition.

Class Design Issues

What if we call an exception throwing method in the body of a method? We have two options. The first is to handle it inside of the method, the other is just to declare that it is thrown. In the case of the BankAccount, the latter strategy is best. We just "percolate" the exception onward, and force the client to deal with it. Here's an example of the first approach.

```
public class BankAccount {

  // ... other stuff ...

  /**
    Decrease the balance by the given amount.  If insufficient funds,
    throws an Exception.
    @param amount the amount to withdraw
  */
  public void withdraw(double amount) throws Exception {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      throw new Exception("Insufficient Funds");
    }
  }

  /**
    Transfer money between accounts.  If the account to transfer from
    has insufficient funds, an Exeption is thrown.
    @param other a BankAccount to tranfer from
    @param amt an amount to transfer
  */
  public void transferFrom(BankAccount other, double amt) throws Exception {
    other.withdraw(amt);
    this.deposit(amt);
  }
}
```

The `transferFrom` method calls the `withdraw` method. Of course, that method may throw an exception. The simple approach taken here is just to declare that fact by noting it in the method signature for `transferFrom`. This approach keeps with our general strategy for this class: force the client to deal with errors relating to insufficient funds.

Now let's look briefly at the other approach, which handles the exception within a method body.

```
public class BankAccount {

  // ... other stuff ...

  /**
    Decrease the balance by the given amount.  If insufficient funds,
    throws an Exception.
    @param amount the amount to withdraw
  */
  public void withdraw(double amount) throws Exception {
    if (amount <= this.balance) {
      this.balance = this.balance - amount;
      return true;
    }
    else {
      throw new Exception("Insufficient Funds");
    }
  }

  /**
    Transfer money between accounts.  Print a message if the origin account
    has insufficient funds.
    @param other a BankAccount to tranfer from
    @param amount an amount to transfer
  */
  public void transferFrom(BankAccount other, double amt) {
    try {
      other.withdraw(amt);
      this.deposit(amt);
    }
    catch (Exception anException) {
      System.out.println("Insufficient funds.");
    }
  }
}
```

Notice that because we are handling the exception in the body of `transferFrom`, we no longer need to modify the method signature of that method.

This approach is undesirable on at least two counts. First, the methods of the class are inconsistent in terms of their error behavior. The `withdraw` method may throw an exception if there are not enough funds, but the `transferFrom` method simply prints out a warning. Consistency in class design is a principle of utmost value, and the above design violates that principle.

Second, the above example demonstrates the potential drawback of handling errors at this level. BankAccounts are data objects, they have basic properties and behaviors, but it's hard to argue that the BankAccount should be responsible for printing an error message. BankAccounts are certainly responsible for doing the right thing in the face of insufficient funds, which is to not alter the account balance and to let the client code know that an error has occurred. Imagine a variety of banking systems that use different display technologies yet all interface with the same BankAccount objects. One system might wish to display a graphical icon if an account is overdrawn; another system may want to display a pop-up dialog; another system may want to print a message on a piece of paper; systems in Germany will want to display their messages in a different language from those in England. The display of an error message to the user is an issue for the designer and implementor of the user interface. The above design violates this partitioning of responsibility, and results in a less reusable class design.