We are going to build our models out of objects. Just as you’ve used models already in the real world, you have used objects as parts of these models. Objects are tangible things, like an individual Lego piece or a plant that is part of a garden (even the garden is an object). Objects are conceptual things, like the date or a meeting between two people. Objects can also be processes, like the writing of a book or the grading of an assignment. Software (or programming) objects use some concepts that you may not have associated with the real-world objects mentioned before.

In particular, our model of objects focuses on their capabilities and properties. To identify capabilities, ask yourself “What can this object do?” An object’s capabilities define how an object operates, so they are often called operations. An object’s properties describe the composition of the object, things that can differentiate one object from another, and parts of an object that can change over time.

Generally objects will have properties that differentiate one from another. Different people have different names, different balls might have different colors, etc. In addition, many properties of an object can change. At one time, a cup may be full, and at another time, empty; a person’s hair can go gray with age. We call a snapshot of an object’s properties at a given time its state at that time; change the value of a property, and the state of the object changes as well.

When we use the term “capabilities,” we are anthropomorphizing the objects we create. Like traditional cartoon characters, objects can behave in ways that make sense in the context of the model but that might not make literal sense in the real world. For example, we might say that a door has the capability of opening and closing. These are capabilities that a door has — we could not describe a door without mentioning these operations. This may give the door too much credit for its role in these operations, as after all the door is usually being acted upon. This approach is sometimes called making smart objects; it is a concept we will return to many times in these chapters.

Most everything can be modeled as a (slightly anthropomorphic) object. Houses can be lived in. Hammers can hit nails. Meetings can hold them-
selves. In this way, objects help enable the creation of a concrete mapping
between the problem domain and the solution domain. As we will see, a
good technique for determining the system’s objects, their properties and
capabilities, is to highlight key words and phrases within the problem defi-
nition. This is a simple form of analysis we can use to answer the two basic
questions of object design: what are the object’s capabilities, and how can
the object change?

■ CONCEPTS

Let’s consider these questions in a real-world example. Figure 2.1 shows a
picture of a camel (you’ll find that often animals make good examples for
object modeling). What are the capabilities of this camel? How can it
change? Study the picture to see if you can determine features that might be
important to model.

Figure 2.1
Picture of a camel

First, though, let’s look a bit more at the basic aspects of objects. Objects
in a software system are like the parts of a camel (contrived? yes). Each
object in a software system has clearly defined capabilities and properties.

Object Capabilities

An object’s capabilities are its operations — that is, what the object is able to
do. A camel’s hump, for example, stores water for the camel to use later.
Most object-oriented programmers refer to capabilities as what the object
“knows” how to do. In this case, we can say that a hump “knows” how to
store water. A hump may also know, for example, how to release some
water from storage for the camel to use. Both of these capabilities are ac-
vated in response to the appropriate signal or message from the camel’s brain
(another object). Objects send messages to other objects to activate each other’s
capabilities to work together to complete a task.

How is it that we can determine what an object can do? There are two pri-
mary concerns when thinking about an object’s capabilities: what does it

© 1997-2000 by Brown University Computer Science Department
make sense for this object to do; and what role does this object play in our model?

Clearly it is silly to make an object capable of doing something that does not make sense, but sometimes that very thing happens when people focus only on answering the second question. Even more problematic is that it is often difficult to determine what it makes sense for an object to do. It is easy to say that a camel’s hump can “store water,” “release water,” or “hold a tourist.” Similarly, it is easy to rule out capabilities like “read a book,” “play cards,” or “sing opera.” However, every capability is not so easy to determine: can the hump “feed the camel;” can it “throw the tourist off;” or even “drink the water?” Often, capabilities that may involve interacting with other objects are hard to classify. These questions are so subtle, and depend so much on what we are modeling, that we will be discussing them throughout these chapters.

Of course there are many things any given object can do. In order to limit the sheer number of these capabilities, it is important to think about why you are modeling this object. It doesn’t help to know that a camel’s hump can “hold a tourist” if all we want to know about the camel is how many humps it has. If we are modeling how far the camel can walk, it may be important to know how much water the hump can hold, but not the process for releasing the water to the camel.

Often, specific capabilities that you will need to model are described in the definition of your model. Thus, a popular technique for finding an object’s capabilities is to examine the verbs in the model definition (this would be part of the analysis stage of model development). For example, if you were asked to describe how a camel stores and retrieves water, those operations are clearly important, whereas how the camel holds a tourist is irrelevant in this case. This technique, combined with determining how objects interact, will be the primary way we develop examples in these chapters.

Analyzing verbs is not the only way to determine an object’s capabilities, but it is an important first step. It would be nice to give you a neat set of rules that you can memorize to help the process, but none yet exist — this is both the difficulty and fun of programming in an object-oriented language.

**Object Properties**

Next, we need to describe the properties of our objects. In English, we model an object with properties and attributes. The properties may be something that belongs to the object, like a door belonging to a house. It is also usually something that all objects of the same type may have, e.g., all people have noses. And an attribute would be something that distinguishes one object from another, say a person’s hair color.

In programming, “properties” is an all-encompassing word, generally modeling three key parts of an object: attributes, components, and associations.
Attributes once again are a way of describing an object to differentiate it from other objects of the same type. One common attribute, for example, is the age of an object. Thus, we could differentiate two camels by pointing out that they had different ages. Attributes generally change over time. Therefore, while an object will always have a certain attribute, the value of that attribute (e.g., the age of the camel) might be different at different times.

Beyond simply describing the attributes of an object, properties can also represent the components (or sub-objects) that together make up the whole object we are modeling. Returning to the camel, we can model it as being composed of a head, torso, legs, and a tail. Of course these properties, like the other properties we have discussed, are objects with their own capabilities and properties. For example, the torso may contain a hump (or two) and the internal organs. By having other objects as properties, objects are able to communicate with one another — an object cannot send a message directly to another object that it does not know about.

Finally an object may have associations with peer objects with which it can communicate, which are not direct components or attributes of the original object. This type of property will play a larger role in chapter 4.

Again, we can get a lot of help in choosing the properties of our objects by examining the model definition.

- Attributes of an object are often expressed as adjectives (e.g., the ball is red).
- An object generally “has” or is “composed of” its components (e.g., a car has a steering wheel).
- Objects usually “know about” their peer objects without containing them (e.g., the professor knows about the TAs).

While these exact words may not be used in the definition, the ideas will be present. We will discuss this examination technique as we build our examples in later chapters.

**Classes and Instances**

In object-oriented programming, we have terminology that lets us describe an object’s capabilities and properties as a single unit. This unit is called a class because it allows us to classify a set of objects by their similar capabilities and properties.

There are probably millions of camels in the world. When referring to a specific one, we mean an “instance” of the general class of camels (e.g., a camel named Zephyr in the Middle East). We can also talk about camels in general. In object-oriented programming, an object class is a type or category of object and an object instance is a particular example of an object class. An object class might be “camel.” An object instance might be identified as “Zephyr.” (From now on, we use “class” to mean “object class” and “instance” to mean “object instance.”) Each instance thus has its own iden-
tity, which means that we can tell one instance to perform an operation without affecting any other instance. We will refer to different instances with different identifiers, a name for something in a programming language (like “Zephyr”). We sometimes say that each instance is a unique instantiation of a class. Because of this, we call the class of an object its type and, as we will see, an object’s type helps the compiler verify the correctness of your program. It is important to note that all instances of a class have the same type, the same capabilities, and the same set of properties.

A rubber stamp, for example, knows “how to be created” and “how to make an imprint”. It also provides each imprint with a color attribute determined by what ink was used with the stamp. We might define “position” and “color” as properties. Each instance will have not only its own location but also its own color, as in Figure 2.2 — one could be black, and two could be light gray. Thus, the class definition (the stamp) describes the existence of properties, but for each instance they might have different values.

Properties are often called instance variables because they are associated with each instance of an object and they may vary over time. A class defines the types of properties each instance will have statically. Thus the types of properties are fixed for the life of the program, but their values may vary.

**Figure 2.2**

A class and four instances — having different values for instance variable “color”

Rubber stamp (class)

Imprints (instances)

Properties are often called *instance variables* because they are associated with each instance of an object and they may vary over time. A class defines the *types* of properties each instance will have statically. Thus the types of properties are fixed for the life of the program, but their *values* may vary.

---

**MECHANICS**

Now that we understand the concepts of classes, instances, capabilities, and properties, let’s take a look at how to express these concepts in Java. To illustrate this, we’ll try to model a ridiculously simplified camel. To make this example work we need to come up with a good problem definition.

Let’s try to define what we want to model with our program. To start we need to have a camel. We want the camel to be able to store water for later use. Let’s also have the camel be able to spit, perhaps at a bothersome tourist.

With a basic definition, let’s try to create a more detailed specification for our model. Let’s first look at the capabilities that our camel needs to have.
We want the camel to be able to store water, and it will probably need to walk to wherever it needs to go. It also needs to be able to spit.

The camel will also have some essential properties. The camel needs to have a hump to store water. In order to get water into the hump, and to be able to spit, a camel needs to have a mouth. And finally, to get to the water and to move around in general, a camel needs to have four legs.

We have now analyzed the definition that was given to us, and we can create our detailed program specification:

*This class should model a simplified camel. The camel should be composed of a hump, a mouth, and four legs. The camel should be able to store water in its hump by drinking from a water source, and should be able to spit at a tourist. The camel should also be able to walk to a destination.*

Note that you aren’t modeling a real camel at all, just the “salient features” for this example.

With this specification we now know what we need to write. Let’s see how to do it in Java!

**Declaring and Defining a Class**

The first thing to do for our model is *declare* the camel class. By this we are telling the compiler that what we are writing is a class. We start with two Java keywords, `public class`, followed by the name of the class we wish to declare. In the case of the camel, we would declare our class like this:

```java
public class Camel
```

The keyword `public` is a *modifier* that means that this class is available to the public, i.e., any other class in the program can know about it. We will discuss modifiers in more detail in later chapters; for right now, though, simply take it as a rule that all classes should be “public” and their definitions preceded by the Java keyword `public`.

Now we are ready to actually *define* the class. This will tell the compiler what the properties of the class actually are and what the class is supposed to be able to do. We follow what was previously written with a “{”, a left curly brace to begin the definition. We end the definition with a “}”, a right curly brace. Within these braces, we will later put the Java *code* that define the properties and capabilities of the class.
Here then, is the simplest class that can be defined (with our additions in bold\(^1\)):

```java
public class Camel {

}
```

This class can be compiled by the Java compiler, i.e., it is syntactically correct, but it has no semantics, i.e., it cannot do anything. The only part of this definition not strictly specified by the compiler is the class' name; that can be any set of *alphanumeric* characters, i.e., letters of the alphabet, numbers, or some special characters like underscores, “_,” that are not used for any other purpose in a Java program.

That means that, since it is not otherwise specified, the names (called identifiers) you choose for your classes (and for their capabilities and properties) are very important because it makes your code easier to understand by you as well as other people (like your TAs). Like the layout of your programs, naming conventions are part of the programming style you adopt when you program. A good consistent style can make your programs much easier to read and fix later on.

Since Java is a programming language used on the World-Wide Web, object names in Java can include letters from just about any language with an alphabet you can think of — you just need a text editor that can type in that language to let you specify Java names in that language. So if you’re more comfortable modeling in Arabic or Russian or some other language, you can do it. The only restrictions are that your name begin with a letter or an underscore, and that it contain no spaces.

This basic definition is the basis of all Java programs. In fact, a Java program is just a collection of these class definitions. Every Java statement that can be executed occurs within the curly braces that denote the “body” of the class.

**Constructors**

How do we tell Java how to make an object instance (i.e., an active object) from an object class (i.e., the generic definition)? There are two answers to the question: you tell Java *when* to make an instance by using the Java keyword *new*, short for “make a new instance”; and you tell Java *how* to build the new instance by defining a *constructor* for your class that contains the actual Java instructions for what you want done when each instance is built. We call this initializing the instance. We will look at the “when” in the next section, but for now let’s focus on the “how.”

---

1. For our examples, we will put all new or changed code in bold so that you can quickly see what changes we have made.

© 1997-2000 by Brown University Computer Science Department
In some sense, constructors are just a capability that objects can have (the capability of being \emph{instantiated}). Whenever an instance is made, Java sets things up so the instance actually exists and then sends a message to the new instance which results in the constructor being activated. Remember that we previously said that an object’s capabilities are activated by sending it messages; in this case the object’s ability to create itself is activated when Java sends the constructor message. Thus, a constructor is automatically invoked whenever an instance is made.

Constructors are special for two important reasons: they cannot be invoked directly by another object, because they are executed automatically only when an object is instantiated; and they must be named the same name as the class name (this is not a convention, it is enforced by the Java compiler).

Very simply, since a constructor is only used to initialize an object instance, the message to activate it cannot be sent at any time like other messages. Instead it is invoked automatically by Java whenever an instance is created. This is important because it prevents you from accidentally constructing an object multiple times. So, the purpose of a constructor is to establish the initial state of an object, which really means that it should prepare the new instance so that another object can send it messages.

We will now define a constructor for our \texttt{Camel} class. Initially it will not do anything, but we will quickly fill it in later. As with a class definition, a constructor begins with a modifier keyword, \texttt{public} — in general, these modifiers should always match, i.e., a public class should have a public constructor. Next comes the name of the constructor, also the same as the class, followed by a set of parentheses, \texttt{()}. We complete the definition by adding open and closed curly brackets, just as with the class:

```java
public class Camel {
    public Camel (){
    }
}
```

Note that if you do not define a constructor for a class, one is typically defined for you. This constructor, often called a \emph{default constructor}, simply initializes the object’s properties with default values. However, this is rarely what we want, so it is a good idea to always define a constructor.

In fact, the default constructor looks remarkably like the one we have defined above. We will change ours as soon as we define some properties for our object to initialize.

### Defining an Object’s Properties

An object’s properties are represented in Java by \emph{instance variables}. Just like in algebra, a variable in Java holds a value. For example, in algebra we often use the letter \texttt{“x”} as a variable. So we can say \texttt{“x=5”} and know that when-
ever we say “x” we mean “5.” When programming however, we often use variables to represent the properties of an object, rather than just a number in an equation. And what are these properties? They are really other objects. So instance variables really are a way of referring to other object instances! However, do not be confused by instance variable and object instance, the “instance” in each does not mean the same thing. We will clarify this later in this chapter.

Note that when we say a variable refers to an object instance, what is really meant is that the value of the variable is that object instance. Therefore, we can change the value of the variable just like we can in algebra. If the variable “x” originally had the value 5, we can change the value to 7 and any equation that used “x” previously won’t change, but the results of any calculations using “x” in the future will change. In Java the value of an instance variable can change, but the variable itself is not affected. For example, a camel may have a color property which originally is brown. That color can change to gray, but the camel still has the color property, even though the value of that property has changed.

The final thing to remember about instance variables is that their value can be different for every instance of the class which uses them. This means that every object template defines that it has a certain number of properties of a certain type. Then each instance of that object will have those same properties, but in each instance the properties could have different values. So there might be a camel named Zephyr who’s color property is brown, and another camel named Zenith who’s color property is gray. Both camels need to have a color property because their class defines one, but their values can be different.

All we need to declare for a property is the name of the class of which we want it to be an instance (its type) and a name for it (its identifier). These declarations must also be preceded by modifiers; however, in this case we will use the special keyword private, meaning no other objects in the program can send messages to these variables.

Let’s write the code for our instance variables. Looking back at the specification we see that we are supposed to be composed of a hump, mouth, and legs (of which we need four). Therefore we need one instance variable for each of these.

```java
public class Camel {
    private CamelHump _hump;
    private CamelMouth _mouth;
    private CamelLeg _leftFront, _leftBack, _rightFront,
        _rightBack;
    private CamelColor _color;

    public Camel() {
    }
}
```
Note each of these “instance variable declaration statements,” variable declarations for short, has the form of “private type identifier;.” As we said above, our properties are objects themselves, so the compiler must know what template to use to stamp out an instance. Note that when declaring the legs, however, we put one variable right after the other, separated by commas. Java allows this convenience for variables of the same type.

Note also that we have preceded each variable name with an underscore. This is a convention we will follow in these chapters for naming instance variables.

Unfortunately, this is not the whole story. We have simply declared our instance variables, but we haven’t given them a useful value. So, since we did not specify one, Java has automatically given them default values of null. This means that every instance of our Camel class will have three properties that represent nothing. If we were to send them a message while our program was running, we would get an error, called a NullPointerException, that would halt the program.

Clearly this is usually not what we want. To create an instance variable that we can use effectively within our model, we must give it the value of a real instance. To do this, we assign it one with the assignment operator, =. Once we have assigned an instance variable a value, we can send messages to the object referenced by that variable (the value of that variable).

A good place to assign our instance variables a value is within the constructor. Since it is invoked whenever an instance is made and before any other message can be sent to the object, it is a good place to give our properties valid values so they can be sent messages later. This is a key component of establishing the initial state of the object.

The syntax for an assignment statement can be seen below in bold:

```java
public class Camel {
    private CamelHump _hump;
    private CamelMouth _mouth;
    private CamelLeg _leftFront, _leftBack, _rightFront,
                     _rightBack;
    private CamelColor _color;

    public Camel(){
        _hump = new CamelHump();
        _mouth = new CamelMouth();
        _leftFront = new CamelLeg();
        _rightFront = new CamelLeg();
        _leftBack = new CamelLeg();
        _rightBack = new CamelLeg();
        _color = new CamelColorBrown();
    }
}
```

Each statement begins with the name of an instance variable followed by an equals sign, the assignment operator, and then the Java keyword `new`. The
name of the class whose instance we are creating follows the word `new`, and a pair of parentheses and a semicolon concludes the statement. In fact, it is not simply the name of the class, but an activation of the class’ constructor. The parentheses are what distinguish a constructor activation from just the class name.

To summarize, for each instance variable, we have told Java to create a new instance of a specific class (`new CamelHump()`), then given this instance as a value for the instance variable (using `=`). The semicolon concludes our statement. These assignment statements are like sentences in the programming language Java — each a complete “thought.” The English sentence might be read as “variable gets value,” or “variable _hump gets new instance of class CamelHump.”

You will notice that names don’t have spaces in them, and that at least one space comes between names. Other than that, the number of spaces and blank lines doesn’t much matter and you can use as many or as few as you want. Even how many statements you put on a single line is arbitrary. You’ll see that in these chapters, we tend to be consistent in spacing. We’ve found that certain ways of spacing a program out on a printed page are more readable than others, and we’re trying to keep our programs as readable as possible.

**Defining a Message and a Method**

We have given our class the appropriate properties, now we need to show somehow what capabilities our class has. In general, we assume that other classes will know what our class’ capabilities are and that they will be able to send it a message to tell it to activate that capability. Therefore it is the programmer’s responsibility to say what an instance of the class will do in response to that message. We call this response a *method*, as in “the method of response to the message.” Unlike the real world, where sometimes a message does not expect a response (like an “F.Y.I.” message), all messages in Java programs need to have a response, even if that response is to explicitly do nothing. A Java message response, or method, has several parts, including what it does, what its name is, and what it provides in return. We will discuss methods that return “stuff” in chapter 3. First, let’s look at the syntax for a method that returns nothing at all:

```java
public void spitAtTourist
```

At this point, our method declaration looks just like our instance variable declaration (without the “;”). To distinguish between the two declarations, we need to add some syntax that says our method references executable code instead of an object instance. We do this using parentheses and curly braces. As with curly braces, parentheses may hold more information in the future. Right now, they are simply placeholders; we will explore what they
contain in more detail next chapter. Let’s look at the empty method definition within the context of the Camel class in which it will be defined:

```java
public class Camel {
    private CamelHump _hump;
    private CamelMouth _mouth;
    private CamelLeg _leftFront, _leftBack, _rightFront, _rightBack;
    private CamelColor _color;

    public Camel() {
        _hump = new CamelHump();
        _mouth = new CamelMouth();
        _leftFront = new CamelLeg();
        _rightFront = new CamelLeg();
        _leftBack = new CamelLeg();
        _rightBack = new CamelLeg();
        _color = new CamelColorBrown();
    }

    public void spitAtTourist() {
    }
}
```

In contrast to our instance variables, our method is `public`, meaning any object in the program can send the `spitAtTourist` message to instances of class `Camel`.

This partial definition (really just the method declaration with brackets) is the only information that other classes know about. The public parts of a class are collectively known as a class’ `interface` (analogous to how a method’s public components are known as its `signature`). This means that another object is aware of any of our class’ public properties and capabilities. In general it is good for a class to know what we are able to do, but it is bad for it to know how we do it. Therefore we generally choose to make all methods `public` and all instance variables `private`. By hiding the implementation of our class’ capabilities, a process known as encapsulation, we are making it a “black box.” The only thing that the public knows is what is shown on the outside, but the internals are completely hidden. The benefits of this will be discussed more in the next chapter.

Now we need to say what the method will do. We do this with a list of statements enclosed within the curly braces. We can use any kind of statement, such as messages being sent to other objects or assignment statements.

In order for our model `Camel` to spit at a tourist, it must use its legs to walk and then use its mouth to spit at the tourist. Obviously, a real camel does this differently, but for our model this is fine. In addition, if we were modeling this in a real program, we would probably use some tools that we
haven’t had time to discuss in this chapter. Therefore, the methods may seem to work a little magically, but the idea behind them remains the same. You will get the tools to make everything less magical in later chapters.

```java
public void spitAtTourist() {
    _leftFront.walkToTourist();
    _rightBack.walkToTourist();
    _rightFront.walkToTourist();
    _leftBack.walkToTourist();
    _mouth.spit();
}
```

This method sends five messages, one to each of its component properties. To do this, we specify the name of the instance to which we want to send the message, followed by a period, followed by the name of the message we want to send. Note we must use parentheses after each message send. Finally, we end each of the five statements in our method body with a semicolon. Sending a message is also known as calling or invoking a method. Thus in the `spitAtTourist` method the camel calls the `walkToTourist` method on four instances of `CamelLeg`.

## Diagrams

It often helps to summarize information about a class in a diagram. We will use a notation in these chapters that is fairly common in object-oriented programming.\(^2\) Now that you have seen the construction of a class, let’s look at how it might be represented in a diagram. Figure 2.3 shows a class box, which describes a camel (very simplistically) by itself. The box shows the class name in boldface at the top. Next the diagram shows the important types of properties that each instance of the class will have (and optionally how we want to refer to that property). Finally the diagram shows the important capabilities that instances of the class can perform, indicating only what an object can do, not how it does it.

Sometimes though, we want to see more information on how a class relates to the other classes in the system. For this we use a class diagram. A class diagram is composed of multiple class boxes. Instead of simply listing the important properties of an object, we now draw various forms of arrows to give more detail on the type of relationship between the objects (whether they are peer objects or components, etc.). If the property is an attribute of the object, we don’t need to draw the box for it (but remember that it is still an object instance). Figure 2.4 shows the class diagram of our camel model (we have added the Tourist property to demonstrate the peer object relation-

At this point you have learned enough Java to build a complete model. We are going to build a simple program that shows a color-changing ball bouncing around within a simple rectangle. While this is not an interactive model, it will exercise the modeling mechanisms we’ve already learned — it will be a complete, working model.

Analysis

Given the basic definition of our problem, let’s see what we are going to need to solve it. Earlier in the chapter we were creating classes without any
context of where they might be used. Now, however, we want to actually use this program in real life. For now, we’re going to need some help.

Generally, a program that is ready to be used is called an application. These range from small games to large word processing programs. Applications can be built with a textual user-interface, or they can have an elaborate graphical user interface (GUI). The user-interface is what allows a user to communicate with the computer. The majority of programs created these days use a GUI to communicate with the user. To understand the “graphical” part of this, let’s take a brief look at computer graphics.

Typical computer displays are organized as a grid of individual lighted points, called pixels (short for picture elements). In fact, when you look at a computer screen you are looking at over a million tiny colored points of light. How each of those points are colored determines what you see on the screen.

Computer graphics is a field of computer science that includes techniques for drawing all kinds of shapes including common 2D geometrical objects, like lines, ovals, and rectangles, as collections of pixels. These types of shapes are needed very often, and it would be silly for a programmer to have to recreate them for every program written. Therefore, people have written graphics libraries which contain objects that already know how to draw these primitives and other important graphics.

![Figure 2.5](image)

Libraries are an important part of programming. You can think of a library as a parts catalog. Often there are certain objects (not just graphical) that need to be used over and over. Instead of writing them from scratch each time, libraries are written so that you the programmer can focus on the unique parts of your program and not have to worry about the very basic components (we sometimes refer to this as being appropriately lazy). Just as you can go through a catalog deciding what parts to order to assemble into a car or a bicycle, you can go through a class library deciding what classes can be used to build bigger and more useful classes. This type of code-reuse is one of the major benefits of object-oriented programming. Java provides some libraries that make up the core of its language. CS 15 is providing you with three more libraries, NGP (which stands for Nice Graphics Package), SP (which stands for Starter Package), and Demos (which is for lecture demos and this book). For this example we are going to be using objects from SP and Demos.

We have been talking about parts throughout this chapter. GUIs are composed of many parts (which are themselves composed of parts, and so on).
For example, look at your favorite Web browser. It probably has a number of buttons along the top, maybe an area to enter text below that, and a large display area below that. Each of these are a part of a graphical user interface. In fact, as we shall see later, it is easy to choose the objects in a graphical program because they correspond to graphical images on the computer screen. This is one of the primary reasons we have chosen “a graphical approach” to teaching object-oriented programming.

The program that you will create here is really a small GUI application. Because this type of application is common Java has provided a framework called (appropriately enough) frame. The purpose of a frame is to specifically use a GUI (and handle many of the GUI's activities for you) and display itself on your computer.

Unfortunately, the frame itself is very generic. There is no way Java could know about every possible application you would want to use a frame for. When we write programs using frames, we will write a specific type of frame that knows how to do everything the generic frame does and more — the more is what we want to model.

To do this, we will extend the generic Frame class to include specific capabilities and properties for our model. Extending classes is a means of organizing classes that is so key to object-oriented programming that we will spend an entire chapter investigating this technique. For now, all that you need to understand is that by extending a generic Frame, we can define specific properties and capabilities that a Web browser will be able to run and display.

If you look through the SP documentation (ask a TA how after you get comfortable with Java) you will see that there is an Frame provided for you. This frame does a number of things for us, but most importantly it can define the rectangular area for our ball to bounce around in. Therefore, this is what we will base our program on. The other component that we need then according to the problem definition is the bouncing ball. Looking through Demos we find that a BouncingBall already exists. This class already knows how to bounce around the screen and change colors; therefore, we are ready to use it.

**Specification**

We have now done a thorough analysis of the problem and are ready to set some specifications for ourself. We know that we are going to need an frame where everything else will be based, and we know that we will need to use a bouncing ball, which has already been written for us. Therefore, our specification is fairly simple.

*This application should model a ball that bounces around and changes color inside a rectangular frame.*

© 1997-2000 by Brown University Computer Science Department
Design

By already having done a good analysis, our design is fairly simple. We know that we have to start with the Frame, and we’ve been told that it does a good job of modeling a rectangular area. The only other component is the bouncing ball. Because the only thing that a user can interact with is the Frame, all graphical objects must be properties of the Frame (or properties of properties of the Frame, etc.). Thus, since we only have two classes, and one must be an instance variable of the other, the hierarchy is clear. The BouncingBall will be an instance variable of the Frame.

The graphics library that we use makes sure that all objects know how to show themselves on the screen once they are created. We also know from the documentation that the pre-existing BouncingBall class takes care of the bouncing and color-changing behavior. Therefore, we don’t even need to send any messages to the ball once it has been created. We will see how to define our own behaviors for shapes later.

Implementation

When it gets down to actually writing the code for our program, it is extremely simple. This is because we are operating at a high-level of abstraction, one that we will be refining to lower levels of abstraction for the rest of the book. At this point, let’s look at the implementation of the Frame class.

We have commented the code so that you will better understand what is happening. This is part of the documentation process that we discussed earlier. Java has two types of comments, in-line comments and multi-line comments. An in-line comment is a comment that goes near a single line of code to explain what that line does and why it does it. In-line comments are signified with a double-slash (/!). A multi-line comment generally gives more detail about either a section of code, or a specific class or method. These are usually more official documentation that should explain conceptually what is happening, while in-line comments are generally to help a reader or the programmer understand what a specific line of code is doing. A multi-line comment is begun with a slash with a star (/!) and then closed with a star and a slash (*/).
We have not seen the Java keyword `extends` before. This is the mechanism that Java provides for us to add on to another pre-existing class, and we will be exploring it in great detail in chapter 4.

The other new syntax is the line that says `public static void main(String[] argv)`. You will notice that this looks similar to a method declaration. Indeed, `main` is a special kind of method known as the `mainline`. The Java Virtual Machine (JavaVM) starts the program by calling this special method. As you can see, the `main` method then uses the new operator to create an instance of `MyApp`. You can think of the mainline as the prolog to the Frame’s constructor. It is almost like a method that doesn’t belong to any class. It may be a little mystifying, especially with the `static`, `void`, and `String[] argv` terms which we have not explained. For now, just accept that this mainline is required for your program to work and we will gradually reveal the significance of each of these terms in future chapters.

**Steps of Execution**

For clarity, let’s follow the steps that are taken for this program to execute. To run this program, we would type `java MyApp` on a command line (a command line is a text-based environment used on the UNIX platform that allows a user to input commands that will execute programs) This command tells the system to invoke the Java VM and run the mainline of the class `MyApp`. The main method instantiates `MyApp` and moves us into the constructor of `MyApp`.

The only thing that `MyApp` does in its constructor is instantiate the `BouncingBall` and invoke its constructor. With the `BouncingBall` off and running, the constructor of `MyApp` is complete. An program that uses a `Frame` will continue to run until the user of the program quits it. Since the `BouncingBall` knows how to bounce around the screen, it will continue to do so until the program is quit.\(^3\)
Testing

Because this program is extremely short, there is not much to test after verifying that it is syntactically correct. However, you should still run the program to verify that it meets the specifications laid out in the beginning of this section. This may seem trivial, but it serves two important purposes: as the user of the program, you should make sure it meets your understanding of the specifications; and since you did not write the program, you may test different things than the original programmer did.

This suggests that when you write programs, you should give them to your intended users to test before you call them done.

3. In our case, since we did not provide a “user-friendly” way to quit the program, to only way to end our bouncing ball program is to forcibly quit it. This is generally done by pressing Control-C.

© 1997-2000 by Brown University Computer Science Department