Object-Oriented Programming

School of Computer Science
University of KwaZulu-Natal

February 5, 2007

Object Oriented Programming using Java

Notes for the Computer Science Module *Object Oriented Programming* COMP200

Adapted from
Introduction to Programming Using Java
Version 5.0, December 2006
by David J. Eck
http://math.hws.edu/javanotes/

Adapted by Anban Pillay
School of Computer Science
University of KwaZulu-Natal
Durban
February 2007

Contents

Preface

These notes are intended for a Second course in Object-Oriented Programming with Java. It is assumed that students have taken a first year course in Programming and are familiar with basic (procedural) programming and introductory object-based programming in Java. The student should be familiar with the various control constucts, Arrays (one and two dimensional), the concepts of class and object, input/output and the concept of classes and objects.

Theses notes are, in the most part, taken from David J. Eck's online book INTRODUCTION TO PROGRAMMING USING JAVA, VERSION 5.0, DECEMBER 2006. The online book is available at *http://math.hws.edu/javanotes/*.

We've refactored and used in whole or parts of Chapters 4, 5, 6, 8, 9, 10, and 11. Subsections of some these chapters were ommitted, minor editing changes were made and a few subsections were added. A notable change has been the use of the `Scanner` class and the `printf` method for input and output.

Some sections were also taken from the notes of Prof Wayne Goddard of Clemson University.

The sections on UML (chapter 6) were adapted from the user manual of the UML tool: **Umbrello** `(http://docs.kde.org/stable/en_GB/kdesdk/umbrello/)`. The definitions of various software engineering terms and concepts were adapted from wikipedia (http://wikipedia.org/).

The LaTeX source for these notes are available on request.

Chapter 1

Introduction to Objects

Contents

OBJECT-ORIENTED PROGRAMMING (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects – entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

An object-oriented programming language such as JAVA includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to "orient" your thinking correctly.

1.1 What is Object Oriented Programming?

OBJECT-ORIENTATION [1] is a set of tools and methods that enable software engineers to build reliable, user friendly, maintainable, well documented, reusable software

[1] This discussion is based on Chapter 2 of An Introduction to Object-Oriented Programming by Timothy Budd.

systems that fulfills the requirements of its users. It is claimed that object-orientation provides software developers with new mind tools to use in solving a wide variety of problems. Object-orientation provides a new view of computation. A software system is seen as a community of *objects* that cooperate with with each other by passing *messages* in solving a problem.

An object-oriented programming laguage provides support for the following object-oriented concepts:

Objects and Classes

Inheritance

Polymophism and Dynamic binding

1.1.1 Programming Paradigms

Object-oriented programming is one of several programming *paradigms*. Other programming paradigms include the imperative programming paradigm (as exemplified by languages such as Pascal or C), the logic programming paradigm (Prolog), and the functional programming paradigm (exemplified by languages such as ML, Haskell or Lisp). Logic and functional languages are said to be *declarative* languages.

We use the word *paradigm* to mean “any example or model”.

This usage of the word was popularised by the science historian *Thomas Kuhn*. He used the term to describe a set of theories, standards and methods that together represent a way of organising knowledge—a way of viewing the world.

Thus a programming paradigm is a

> *. . . way of conceptualising what it means to perform computation and how tasks to be carried out on a computer should be structured and organised.*

We can distinguish between two types of programming languages: *Imperative* languages and *declarative* languages. Imperative knowledge describes *how-to* knowledge while declarative knowledge is *what-is* knowledge.

A program is ”declarative” if it describes what something is like, rather than how to create it. This is a different approach from traditional imperative programming languages such as Fortran, and C, which require the programmer to specify an algorithm to be run. In short, imperative programs make the algorithm explicit and leave the goal implicit, while declarative programs make the goal explicit and leave the algorithm implicit.

Imperative languages require you to write down a step-by-step recipe specifing *how* something is to be done. For example to calculate the factorial function in an imperative language we would write something like:

```
public int factorial(int n) {
   int ans=1;
   for (int i = 2; i <= n; i++){
       ans = ans * i;
   }
   return ans;
}
```

Here, we give a procedure (a set of steps) that when followed will produce the answer.

Functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions. Functional programming emphasizes the definition of functions, in contrast to procedural programming, which emphasizes the execution of sequential commands.

The following is the factorial function written in a functional language called *Lisp*:

```
(defun factorial (n)
   (if (<= n 1) 1 (* n (factorial (- n 1))))
)
```

Notice that it defines the factorial function rather than give the steps to calculate it. The factorial of n is defined as 1 if $n <= 1$ else it is $n * factorial(n-1)$

Logic Programming

Prolog (PROgramming in LOGic) [2] is the most widely available language in the logic programming paradigm. It is based on the mathematical ideas of relations and logical inference. Prolog is a declarative language meaning that rather than describing how to compute a solution, a program consists of a data base of facts and logical relationships (rules) which describe the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question. When asked a question, the run time system searches through the data base of facts and rules to determine (by logical deduction) the answer.

Logic programming was an attempt to make a programming language that enabled the expression of logic instead of carefully specified instructions on the computer.

In the logic programming language *Prolog* you supply a database of facts and rules; you can then perform queries on the database.

This is also an example of a declarative style of programming where we state or define what we know.

In the following example, we declare facts about some domain. We can then query these facts—we can ask, for example, are sally and tom siblings?

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
parent(X,Y)  :- father(X,Y).
parent(X,Y) :- mother(X,Y).
mother(trude, sally).
father(tom, sally).
father(tom, erica).
father(mike, tom).
```

The factorial function is written in prolog as two rules. Again, notice the declarative nature of the program.

```
fac(0,1).
fac(N,F) :- N > 0,
            M is N - 1,
            fac(M,Fm),
            F is N * Fm.
```

To summarize:

- In *procedural* languages, everything is a procedure.

[2] (see http://cs.wwc.edu/KU/PR/Prolog.html)

- In *functional* languages, everything is a function.
- In *logic* programming languages, everything is a logical expression (predicate).
- In *object-oriented* languages, everything is an object.

1.1.2 Object Orientation as a New Paradigm: The Big Picture

It is claimed that the problem-solving techniques used in object-oriented programming more closely models the way humans solve day-to-day problems.[3]

So lets consider how we solve an everyday problem: Suppose you wanted to send flowers to a friend named Robin who lives in another city.To solve this problem you simply walk to your nearest florist run by, lets say, Fred. You tell Fred the kinds of flowers to send and the address to which they should be delivered. You can be assured that the flowers will be delivered.

Now, lets examine the mechanisms used to solve your problem.

- You first found an appropriate **agent** (Fred, in this case) and you passed to this agent a **message** containing a request.
- It is the **responsibility** of Fred to satisfy the request.
- There is some **method** (an algorithm or set of operations) used by Fred to do this.
- You do not need to know the particular methods used to satisfy the request—such information is **hidden** from view.

Off course, you do not want to know the details, but on investigation you may find that Fred delivered a slightly different message to another florist in the city where your friend Robin lives. That florist then passes another message to a subordinate who makes the floral arrangement.The flowers, along with yet another message, is passed onto a delivery person and so on. The florists also has interactions with wholesalers who, in turn, had interactions with flower growers and so on.

This leads to our first conceptual picture of object-oriented programming:

> *An object-oriented program is structured as* ***community*** *of interacting agents called* ***objects****. Each object has a role to play. Each object provides a* ***service*** *or performs an action that is used by other members of the community.*

Messages and Responsibilities

Members of an object-oriented community make requests of each other. The next important principle explains the use of messages to initiate action:

> *Action is initiated in object-oriented programming by the transmission of a* ***message*** *to an agent (an* ***object****) responsible for the actions. The message encodes the request for an action and is accompanied by any additional information (arguments/parameters) needed to carry out the request. The* ***receiver*** *is the object to whom the message is sent. If the receiver accepts*

[3] This discussion is based on Chapter 2 of An Introduction to Object-Oriented Programming by Timothy Budd.

> *the message, it accepts **responsibility** to carry out the indicated action. In response to a message, the receiver will perform some **method** to satisfy the request.*

There are some important issues to point out here:

- The client sending the request need not know the means by which the request is carried out. In this we see the principle of *information hiding*.
- Another principle implicit in message passing is the idea of finding someone else to do the work i.e. reusing components that may have been written by someone else.
- The interpretation of the message is determined by the receiver and can vary with different receivers. For example, if you sent the message "deliver flowers" to a friend, she will probably have understood what was required and flowers would still have been delivered but the method she used would have been very different from that used by the florist.
- In object-oriented programming, behaviour is described in terms of *responsibilities*.
- Client's requests for actions only indicates the desired outcome. The receivers are free to pursue any technique that achieves the desired outcomes.
- Thinking in this way allows greater *independence* between objects.
- Thus, objects have responsibilities that they are willing to fulfill on request. The collection of reponsibilities associated with an object is often called a *protocol*.

Classes and Instances

The next important principle of object-oriented programming is

> *All objects are instances of a **class**. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.*

Fred is an instance of a category or class of people i.e. Fred is an instance of a class of florists. The term florist represents a class or category of all florists. Fred is an object or instance of a class.

We interact with instances of a class but the class determines the behaviour of instances. We can tell a lot about how Fred will behave by understanding how Florists behave. We know, for example, that Fred, like all florists can arrange and deliver flowers.

In the real world there is this distinction between classes and objects. Real-world objects share two characteristics: They all have *state* and *behavior*. For example, dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Students have *state* (name, student number, courses they are registered for, gender) and *behavior* (take tests, attend courses, write tests, party).

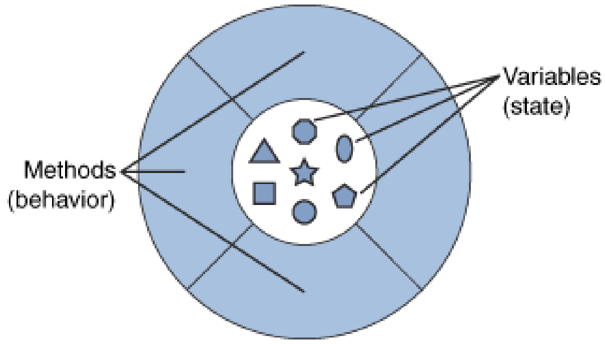


Figure 1.1: An Object

1.2 Fundamentals of Objects and Classes

We move now from the conceptual picture of objects and classes to a discussion of software classes and objects.[4]

Objects are closely related to classes. A class can contain variables and methods. If an object is also a collection of variables and methods, how do they differ from classes?

1.2.1 Objects and Classes

Objects

In object-oriented programming we create software objects that model real world objects. Software objects are modeled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more *variables*. A variable is an item of data named by an *identifier*. A software object implements its behavior with *methods*. A method is a function associated with an object.

Definition: An object is a software bundle of variables and related methods.

An object is also known as an *instance*. An instance refers to a particular object. For e.g. Karuna's bicycle is an instance of a bicycle—It refers to a particular bicycle. Sandile Zuma is an instance of a Student.

The variables of an object are formally known as *instance variables* because they contain the state for a particular object or instance. In a running program, there may be many instances of an object. For e.g. there may be many `Student` objects. Each of these objects will have their own instance variables and each object may have different values stored in their instance variables. For e.g. each Student object will have a different number stored in its `StudentNumber` variable.

Encapsulation

Object diagrams show that an object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called *encapsulation*.

[4] This discussion is based on the "Object-oriented Programming Concepts" section of the Java Tutorial by Sun MicroSystems.

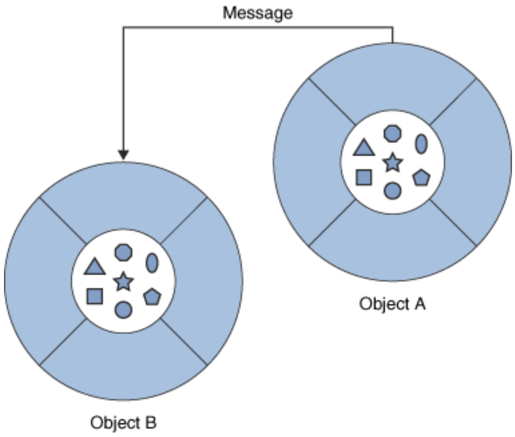


Figure 1.2: A Message

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two benefits to software developers:

- *Modularity:* The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
- *Information-hiding:* An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting other objects that depend on it.

Messages

Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B

There are *three* parts of a message: The three parts for the message System.out.println{‘‘Hello World′′}; are:

- The *object* to which the message is addressed (System.out)
- The *name* of the method to perform (println)
- Any *parameters* needed by the method ("Hello World!")

Classes

In object-oriented software, it's possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. A class is a software blueprint for objects. A class is used to manufacture or create objects.

The class declares the instance variables necessary to contain the state of every object. The class would also declare and provide implementations for the instance methods necessary to operate on the state of the object.

> *Definition: A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.*

After you've created the class, you can create any number of objects from that class.

A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and methods the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Types

JAVA, like most programming languages classifies values and expressions into *types*. For e.g. `String`'s and **`int`**'s are types. A type basically specifies the allowed values and allowed operations on values of that type.

> *Definition: A type is a set of values together with one or more operations that can be applied uniformly to all these values.*

A type system basically gives meaning to collections of bits. Because any value simply consists of a set of bits in a computer, the hardware makes no distinction between memory addresses, instruction code, characters, integers and floating-point numbers. Types inform programs and programmers how they should treat those bits.

For example the integers are a type with values in the range $-2,147,483,648\ to\ +2,147,483,647$ and various allowed operations that include addition, subtraction, modulus etc.

The use of types by a programming language has several advantages:

- Safety. Use of types may allow a compiler to detect meaningless or invalid code. For example, we can identify an expression "Hello, World" / 3 as invalid because one cannot divide a string literal by an integer. Strong typing offers more safety.

- Optimization. Static type-checking may provide useful information to a compiler. The compiler may then be able to generate more efficient code.

- Documentation. Types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be a subtype of integers – but if a programmer declares a method as returning a timestamp rather than merely an integer, this documents part of the meaning of the method.

- Abstraction. Types allow programmers to think about programs at a higher level, not bothering with low-level implementation. For example, programmers can think of strings as values instead of as a mere array of bytes.

There are fundamentally two types in JAVA: primitive types and objects types i.e. any variable you declare are either declared to be one of the primitive types or an object type. **`int`**, **`double`** and **`char`** are the built-in, primitive types in JAVA.

The primitive types can be used in various combinations to create other, composite types. Every time we define a class, we are actually defining a new type. For example, the `Student` class defined above introduces a new type. We can now use this type like any other type: we can declare variables to be of this type and we can use it as a type for parameters of methods.

Before a variable can be used, it must be declared. A declaration gives a variable a name, a type and an initial value for e.g. **int** x = 8 declares x to be of type **int**. All objects that we declare also have to be of a specified type—the type of an object is the class from which it is created. Thus, when we declare objects we state the type like so: Student st = **new** Student();. This statement declares the variable st to be of type Student. This statement creates a new object of the specified type and runs the Student constructor. The constructor's job is to properly initialize the object.

The String type is another example of an object type. Student and String are composite types and give us the same advantages as the built-in types. The ability to create our own types is a very powerful idea in modern languages.

When declaring variables, we can assign initial values. If you do not specify initial values, the compiler automatically assigns one: Instance variables of numerical type (**int**, **double**, etc.) are automatically initialized to zero; **boolean** variables are initialized to **false**; and **char** variables, to the Unicode character with code number zero. The default initial value of object types is **null**.

Introduction to Enums

JAVA comes with eight built-in primitive types and a large set of types that are defined by classes, such as String. But even this large collection of types is not sufficient to cover all the possible situations that a programmer might have to deal with. So, an essential part of JAVA, just like almost any other programming language, is the ability to create **new** types. For the most part, this is done by defining new classes. But we will look here at one particular case: the ability to define enums (short for enumerated types). Enums are a recent addition to JAVA. They were only added in Version 5.0. Many programming languages have something similar.

Technically, an enum is considered to be a special kind of class. In this section, we will look at enums in a simplified form. In practice, most uses of enums will only need the simplified form that is presented here.

> *An* enum *is a type that has a fixed list of possible values, which is specified when the* enum *is created.*

In some ways, an enum is similar to the **boolean** data type, which has **true** and **false** as its only possible values. However, **boolean** is a primitive type, while an enum is not.

The definition of an enum types has the (simplified) form:

```
enum enum−type−name { list−of−enum−values };
```

This definition cannot be inside a method. You can place it **outside** the main() method of the program. The enum−type−name can be any simple identifier. This identifier becomes the name of the enum type, in the same way that "**boolean**" is the name of the **boolean** type and "String" is the name of the String type. Each value in the list−of−enum−values must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named Season whose values are the names of the four seasons of the year:

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER };
```

By convention, enum values are given names that are made up of upper case letters, but that is a style guideline and not a syntax rule. Enum values are not variables. Each value is a constant that always has the same value. In fact, the possible values of an enum type are usually referred to as enum constants.

Note that the enum constants of type Season are considered to be "contained in" Season, which means–following the convention that compound identifiers are used for things that are contained in other things–the names that you actually use in your program to refer to them are Season.SPRING, Season.SUMMER, Season.AUTUMN, and Season.WINTER.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named vacation of type Season with the statement:

```
Season vacation;
```

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type Season. Remember to use the full name of the constant, including "Season"! For example: vacation = Season.SUMMER;.

You can print an enum value with the statement: System.out.print(vacation). The output value will be the name of the enum constant (without the "Season."). In this case, the output would be "SUMMER".

Because an enum is technically a class, the enum values are technically objects. As objects, they can contain methods. One of the methods in every enum value is ordinal(). When used with an enum value it returns the ordinal number of the value in the list of values of the enum. The ordinal number simply tells the position of the value in the list. That is, Season.SPRING.ordinal() is the int value 0, Season.SUMMER.ordinal() is 1, while 2 is Season.AUTUMN.ordinal(), and 3 is Season.WINTER.ordinal() is. You can use the ordinal() method with a variable of type Season, such as vacation.ordinal() in our example.

You should appreciate enums as the first example of an important concept: creating new types. Here is an example that shows enums being used in a complete program:

```
public class EnumDemo {
   // Define two enum types—definitions go OUTSIDE The main() routine!
   enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
   enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }

   public static void main(String[] args) {
      Day tgif;      // Declare a variable of type Day.
      Month libra;  // Declare a variable of type Month.
      tgif = Day.FRIDAY;     // Assign a value of type Day to tgif.
      libra = Month.OCT;     // Assign a value of type Month to libra.

      System.out.print("My sign is libra, since I was born in ");
      System.out.println(libra);    // Output value will be: OCT
      System.out.print("That's the ");
      System.out.print( libra.ordinal() );
      System.out.println("-th month of the year.");
      System.out.println("   (Counting from 0, of course!)");
      System.out.print("Isn't it nice to get to ");
      System.out.println(tgif);    // Output value will be: FRIDAY
      System.out.println( tgif + " is the " + tgif.ordinal()
            + "-th day of the week."); //Can concatenate enum values onto Strings!
   }
}
```

Enums and for-each Loops

Java 5.0 introduces a new "enhanced" form of the **for** loop that is designed to be convenient for processing data structures. A data structure is a collection of data items, considered as a unit. For example, a list is a data structure that consists simply of a sequence of items. The enhanced **for** loop makes it easy to apply the same processing to every element of a list or other data structure. However, one of the applications of the enhanced **for** loop is to enum types, and so we consider it briefly here.

The enhanced for loop can be used to perform the same processing on each of the enum constants that are the possible values of an enumerated type. The syntax for doing this is:

```
for ( enum–type–name  variable–name  :  enum–type–name.values() )
   statement
```

or

```
for ( enum–type–name  variable–name  :  enum–type–name.values() ) {
   statements
}
```

If MyEnum is the name of any enumerated type, then MyEnum.values() is a method call that returns a list containing all of the values of the enum. (values() is a static member method in MyEnum and of any other enum.) For this enumerated type, the **for** loop would have the form:

```
for ( MyEnum  variable–name  :  MyEnum.values() )
   statement
```

The intent of this is to execute the statement once for each of the possible values of the MyEnum type. The variable-name is the loop control variable. In the statement, it represents the enumerated type value that is currently being processed. This variable should **not** be declared before the for loop; it is essentially being declared in the loop itself.

To give a concrete example, suppose that the following enumerated type has been defined to represent the days of the week:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

Then we could write:

```
for ( Day d : Day.values() ) {
   System.out.print( d );
   System.out.print(" is day number ");
   System.out.println( d.ordinal() );
}
```

Day.values() represents the list containing the seven constants that make up the enumerated type. The first time through this loop, the value of d would be the first enumerated type value Day.MONDAY, which has ordinal number 0, so the output would be "MONDAY is day number0". The second time through the loop, the value of d would be Day.TUESDAY, and so on through Day.SUNDAY. The body of the loop is executed once for each item in the list Day.values(), with d taking on each of those values in turn. The full output from this loop would be:

```
MONDAY is day number 0
TUESDAY is day number 1
```

```
WEDNESDAY is day number 2
THURSDAY is day number 3
FRIDAY is day number 4
SATURDAY is day number 5
SUNDAY is day number 6
```

Since the intent of the enhanced for loop is to do something "for each" item in a data structure, it is often called a for-each loop. The syntax for this type of loop is unfortunate. It would be better if it were written something like "foreach Day d in Day.values()", which conveys the meaning much better and is similar to the syntax used in other programming languages for similar types of loops. It's helpful to think of the colon (:) in the loop as meaning "in."

1.2.2 Class Members and Instance Members

A class definition is made of *members* or components. A class can define *variables* (or *fields*) and methods. Variables and methods can be static or non-static i.e. they are defined with or without the keyword *static*.

e.g.

```
static double lastStudentNumber; //a static member/variable/field
double studentNumber; //a non-static variable

static void printLastNumber() {...}  //a static member/method
void printNumber() {...}  //a non-static method
```

The non-static members of a class (variables and methods) are also known as *instance* variables and methods while the non-static members are also known as class variables and class methods. Each instance of a class (each object) gets its own copy of all the instance variables defined in the class. When you create an instance of a class, the system allocates enough memory for the object and all its instance variables.

In addition to instance variables, classes can declare class variables. A class variable contains information that is shared by all instances (objects) of the class. If one object changes the variable, it changes for all other objects of that type. e.g. A Student number generator in a `NewStudent` class.

You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance. e.g. The methods in the `Math` class are static and can be invoked without creating an instance of the Math class for e.g. we can say `Math.sqrt(x)`.

Consider a simple class whose job is to group together a few static member variables for example a class could be used to store information about the person who is using the program:

```
class UserData { static String name; static int age; }
```

In programs that use this class, there is one copy each of the variables `UserData.name` and `UserData.age`. There can only be one "user," since we only have memory space to store data about one user. The class, `UserData`, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:

```
class PlayerData { String name; int age; }
```

In this case, there is no such variable as `PlayerData.name` or `PlayerData.age`, since name and age are not static members of `PlayerData`. There is nothing much in the class except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each object will have its **own** variables called name and age. There can be many "players" because we can make new objects to represent new players on demand. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new `PlayerData` object can be created to represent that player. If a player leaves the game, the `PlayerData` object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with "static" variables!

An object that belongs to a class is said to be an `instance` of that class and the variables that the object contains are called `instance variables`. The methods that the object contains are called `instance methods`.

For example, if the `PlayerData` class, is used to create an object, then that object is an instance of the `PlayerData` class, and name and age are instance variables in the object. It is important to remember that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

The source code for methods are defined in the class yet it's better to think of the instance methods as belonging to the object, not to the class. The non-static methods in the class merely specify the instance methods that every object created from the class will contain. For example a `draw()` method in two different objects do the same thing in the sense that they both draw something. But there is a real difference between the two methods—the things that they draw can be different. You might say that the method definition in the class specifies what type of behavior the objects will have, but the specific behavior can vary from object to object, depending on the values of their instance variables.

The static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class. The "static" definitions in the source code specify the things that are part of the class itself, whereas the non-static definitions in the source code specify things that will become part of every instance object that is created from the class. Static member variables and static member methods in a class are sometimes called **`class`** `variables` and **`class`** `methods`, since they belong to the class itself, rather than to instances of that class.

So far, we've been talking mostly in generalities. Let's now look at a specific example to see how classes and objects work. Consider this extremely simplified version of a `Student` class, which could be used to store information about students taking a course:

```
public class Student {

  public String name; // Student's name.  public double test1 ,
  test2, test3; // Grades on three tests.

  public double getAverage() { // compute average test grade return
    (test1 + test2 + test3) / 3; }

}// end of class Student
```

None of the members of this class are declared to be **`static`**, so the class exists only for creating objects. This class definition says that any object that is an instance of the Student class will include instance variables named name, test1, test2, and test3, and it will include an instance method named getAverage(). The names and tests in different objects will generally have different values. When called for a particular student, the method getAverage() will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In JAVA, a class is a **type**, similar to the built-in types such as **int** and **boolean**. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a method. For example, a program could define a variable named std of type Student with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

> *In* JAVA*, no variable can ever hold an object. A variable can only hold a reference to an object.*

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the *heap* where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a *reference* or *pointer* to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

In a program, objects are created using an operator called **new**, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself. It is not quite true to say that the object is the "value of the variable std". It is certainly **not at all true** to say that the object is "stored in the variable std." The proper terminology is that "the variable std *refers to* the object,".

So, suppose that the variable std refers to an object belonging to the class Student. That object has instance variables name, test1, test2, and test3. These instance variables can be referred to as std.name, std.test1, std.test2, and std.test3. This follows the usual naming convention that when B is part of A, then the full name of B is A.B. For example, a program might include the lines

```
System.out.println("Hello, " + std.name + ". Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which std refers. Similarly, std can be used to call the getAverage() instance method in the object by saying std.getAverage(). To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use std.name any place where a variable of type String is legal. You can use it in expressions. You can assign a value to it. You can pass it as a parameter to method. You can even use it to call methods from the String class. For example, std.name.length() is the number of characters in the student's name.

It is possible for a variable like std, whose type is given by a class, to refer to no object at all. We say in this case that std holds a **null** reference. The null reference is written in JAVA as "**null**". You can store a null reference in the variable std by saying "std = **null**;" and you could test whether the value of "std" is null by testing "**if** (std == **null**) . . .".

If the value of a variable is **null**, then it is, of course, illegal to refer to instance variables or instance methods through that variable–since there is no object, and hence no instance variables to refer to. For example, if the value of the variable st is **null**, then it would be illegal to refer to std.test1. If your program attempts to use a null reference illegally like this, the result is an error called a *null pointer exception*.

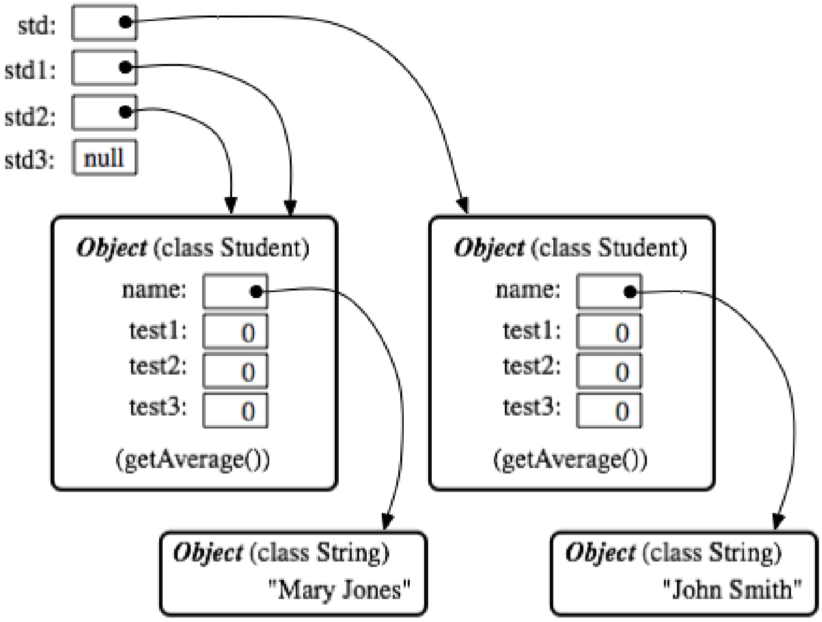
Let's look at a sequence of statements that work with objects:

```
Student std, std1,          // Declare four variables of
          std2, std3;       //   type Student.
std = new Student();        // Create a new object belonging
                            //   to the class Student, and
                            //   store a reference to that
                            //   object in the variable std.
std1 = new Student();       // Create a second Student object
                            //   and store a reference to
                            //   it in the variable std1.
std2 = std1;                // Copy the reference value in std1
                            //   into the variable std2.
std3 = null;                // Store a null reference in the
                            //   variable std3.

std.name = "John Smith";  // Set values of some instance variables.
std1.name = "Mary Jones";

     // (Other instance variables have default
     //     initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable `std3`, with a value of **`null`**, doesn't point anywhere. The arrows from `std1` and `std2` both point to the same object. This illustrates a Very Important Point:

> *When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.*

When the assignment "`std2 = std1;`" was executed, no new object was created. Instead, `std2` was set to refer to the very same object that `std1` refers to. This has some consequences that might be surprising. For example, `std1.name` and `std2.name` are two different names for the same variable, namely the instance variable in the object that both `std1` and `std2` refer to. After the string "`Mary Jones`" is assigned to the variable `std1.name`, it is also be true that the value of `std2.name` is "`Mary Jones`". There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators `==` and `!=`, but here again, the semantics are different from what you are used to. The test "**`if`** `(std1 == std2)`", tests whether the values stored in `std1` and `std2` are the same. But the values are references to objects, not objects. So, you are testing whether `std1` and `std2` refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether

```
std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3
== std2.test3 && std1.name.equals(std2.name)}
```

I've remarked previously that `Strings` are objects, and I've shown the strings "`Mary Jones`" and "`John Smith`" as objects in the above illustration. A variable of type String can only hold a reference to a string, not the string itself. It could also hold the value **null**, meaning that it does not refer to any string at all. This explains why using the == operator to test strings for equality is not a good idea.

The fact that variables hold references to objects, not objects themselves, has a couple of other consequences that you should be aware of. They follow logically, if you just keep in mind the basic fact that the object is not stored in the variable. The object is somewhere else; the variable points to it.

Suppose that a variable that refers to an object is declared to be **final**. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is **final**, not the object. It's perfectly legal to say

```
final Student stu = new Student();

stu.name = "John Doe"; // Change data in the object;
                       // The value stored in stu is not changed!
                       // It still refers to the same object.
```

Next, suppose that obj is a variable that refers to an object. Let's consider what happens when obj is passed as an actual parameter to a method. The value of obj is assigned to a formal parameter in the method, and the method is executed. The method has no power to change the value stored in the variable, obj. It only has a copy of that value. However, that value is a reference to an object. Since the method has a reference to the object, it can change the data stored in the object. After the method ends, obj still points to the same object, but the data stored **in the object** might have changed. Suppose x is a variable of type **int** and stu is a variable of type Student. Compare:

```
void dontChange(int z) {         void change(Student s) {
     z = 42;                           s.name = "Fred";
}                                }

The lines:                       The lines:

x = 17;                          stu.name = "Jane";
dontChange(x);                   change(stu);
System.out.println(x);           System.out.println(stu.name);

outputs the value 17.            outputs the value "Fred".

The value of x is not            The value of stu is not changed ,
changed by the method,           but stu.name is.
which is equivalent to           This is equivalent to

z = x;                           s = stu;
z = 42;                          s.name = "Fred";
```

1.2.3 Access Control

When writing new classes, it's a good idea to pay attention to the issue of access control. Recall that making a member of a class **public** makes it accessible from anywhere, including from other classes. On the other hand, a **private** member can only be used in the class where it is defined.

In the opinion of many programmers, almost all member variables should be declared **private**. This gives you complete control over what can be done with the variable. Even if the variable itself is private, you can allow other classes to find out what its value is by providing a **public** accessor method that returns the value of the variable. For example, if your class contains a **private** member variable, title, of type String, you can provide a method

```
public String getTitle() { return title; }
```

that returns the value of title. By convention, the name of an accessor method for a variable is obtained by capitalizing the name of variable and adding "get" in front of the name. So, for the variable title, we get an accessor method named "get" + "Title", or getTitle(). Because of this naming convention, accessor methods are more often referred to as getter methods. A getter method provides "read access" to a variable.

You might also want to allow "write access" to a **private** variable. That is, you might want to make it possible for other classes to specify a new value for the variable. This is done with a setter method. (If you don't like simple, Anglo-Saxon words, you can use the fancier term mutator method.) The name of a setter method should consist of "set" followed by a capitalized copy of the variable's name, and it should have a parameter with the same type as the variable. A setter method for the variable title could be written

```
public void setTitle( String newTitle ) { title = newTitle; }
```

It is actually very common to provide both a getter and a setter method for a private member variable. Since this allows other classes both to see and to change the value of the variable, you might wonder why not just make the variable **public**? The reason is that getters and setters are not restricted to simply reading and writing the variable's value. In fact, they can take any action at all. For example, a getter method might keep track of the number of times that the variable has been accessed:

```
public String getTitle() {
    titleAccessCount++; //Increment member variable titleAccessCount.
    return title;
}
```

and a setter method might check that the value that is being assigned to the variable is legal:

```
public void setTitle( String newTitle ) {
  if ( newTitle == null ) //Don't allow null strings as titles!
     title = "(Untitled)"; // Use an appropriate default value instead.
  else
     title = newTitle; }
```

Even if you can't think of any extra chores to do in a getter or setter method, you might change your mind in the future when you redesign and improve your class. If you've used a getter and setter from the beginning, you can make the modification to your class without affecting any of the classes that use your class. The **private** member variable is not part of the public interface of your class; only the **public** getter and setter methods are. If you **haven't** used get and set from the beginning, you'll have to contact everyone who uses your class and tell them, "Sorry guys, you'll have to track down every use that you've made of this variable and change your code."

1.2.4 Creating and Destroying Objects

Object types in JAVA are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

Initializing Instance Variables

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named PairOfDice. An object of this class will represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {

     public int die1 = 3;    // Number showing on the first die.
     public int die2 = 4;    // Number showing on the second die.

     public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
          die1 = (int)(Math.random()*6) + 1;
          die2 = (int)(Math.random()*6) + 1;
     }

} // end class PairOfDice
```

The instance variables die1 and die2 are initialized to the values 3 and 4 respectively. These initializations are executed whenever a PairOfDice object is constructed. It is important to understand when and how this happens. Many PairOfDice objects may exist. Each time one is created, it gets its own instance variables, and the assignments "die1 = 3" and "die2 = 4" are executed to fill in the values of those variables. To make this clearer, consider a variation of the PairOfDice class:

```
public class PairOfDice {

  public int die1 = (int)(Math.random()*6) + 1;
  public int die2 = (int)(Math.random()*6) + 1;

  public void roll() {
     die1 = (int)(Math.random()*6) + 1;
     die2 = (int)(Math.random()*6) + 1;
  }

} // end class PairOfDice
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a **static** variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Instance variables of numerical type (**int**, **double**, etc.) are automatically initialized to zero if you provide no other values; **boolean** variables are initialized to **false**; and **char** variables, to the Unicode character with code number zero. An instance variable can also be a variable of object type. For such variables, the default initial value is **null**. (In particular, since Strings are objects, the default initial value for String variables is **null**.)

Constructors

Objects are created with the operator, **new**. For example, a program that wants to use a PairOfDice object could say:

```
PairOfDice dice; // Declare a variable of type PairOfDice.

dice = new PairOfDice(); // Construct a new object and store a
                         // reference to it in the variable.
```

In this example, "**new** PairOfDice()" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, dice, so that after the assignment statement is executed, dice refers to the newly created object. Part of this expression, "PairOfDice()", looks like a method call, and that is no accident. It is, in fact, a call to a special type of method called a constructor. This might puzzle you, since there is no such method in the class definition. However, every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a **default** constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other method, with three differences.

1. A constructor does not have any return type (not even **void**).

2. The name of the constructor must be the same as the name of the class in which it is defined.

3. The only modifiers that can be used on a constructor definition are the access modifiers **public**, **private**, and **protected**. (In particular, a constructor can't be declared **static**.)

However, a constructor does have a method body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the PairOfDice class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

The constructor is declared as "**public** PairOfDice(**int** val1, **int** val2)...", with no return type and with the same name as the name of the class. This is how the JAVA compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "**new** PairOfDice(3,4)" would create a PairOfDice object in which the values of the instance variables die1 and die2 are initially 3 and4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice; // Declare a variable of type PairOfDice.

dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            // object that initially shows 1, 1.
```

Now that we've added a constructor to the PairOfDice class, we can no longer create an object by saying "**new** PairOfDice()"! The system provides a default constructor for a class **only** if the class definition does not already include a constructor, so there is only one constructor in the class, and it requires two actual parameters. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the PairOfDice class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
            // Constructor.  Rolls the dice, so that they initially
            // show some random values.
        roll();  // Call the roll() method to roll the dice.
    }

    public PairOfDice(int val1, int val2) {
            // Constructor.  Creates a pair of dice that
            // are initially showing the values val1 and val2.
        die1 = val1;  // Assign specified values
        die2 = val2;  // to the instance variables.
    }

    public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a PairOfDice object with "**new** PairOfDice()" or with "**new** PairOfDice(x,y)", where x and y are **int**-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "`(int)(Math.random()*6)+1`", because it's done inside the `PairOfDice` class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the `PairOfDice` class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value. This illustrates once again that you can create several instances of the same class:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();

        int countRolls;   // Counts how many times the two pairs of
                          //    dice have been rolled.

        int total1;       // Total showing on first pair of dice.
        int total2;       // Total showing on second pair of dice.

        countRolls = 0;

        do {  // Roll the two pairs of dice until totals are the same.

            firstDice.roll();    // Roll the first pair of dice.
            total1 = firstDice.die1 + firstDice.die2;   // Get total.
            System.out.println("First pair comes up  " + total1);

            secondDice.roll();    // Roll the second pair of dice.
            total2 = secondDice.die1 + secondDice.die2;   // Get total.
            System.out.println("Second pair comes up " + total2);

            countRolls++;   // Count this roll.

            System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                          + " rolls until the totals were the same.");

    } // end main()

} // end class RollTwoPairs
```

Constructors are methods, but they are methods of a special type. They are certainly not instance methods, since they don't belong to objects. Since they are responsible for creating objects, they exist before any objects have been created. They are more like **static** member methods, but they are not and cannot be declared to be **static**. In fact, according to the JAVA language specification, they are technically not members of the class at all! In particular, constructors are **not** referred to as "methods".

Unlike other methods, a constructor can only be called using the **new** operator, in an expression that has the form

```
new class−name{parameter−list}
```

where the parameter−list is possibly empty. I call this an expression because it computes and returns a value, namely a reference to the object that is constructed. Most often, you will store the returned reference in a variable, but it is also legal to use a constructor call in other ways, for example as a parameter in a method call or as part of a more complex expression. Of course, if you don't save the reference in a variable, you won't have any way of referring to the object that was just created.

A constructor call is more complicated than an ordinary method call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.

2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.

3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.

4. The statements in the body of the constructor, if any, are executed.

5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

For another example, let's rewrite the Student class. I'll add a constructor, and I'll also take the opportunity to make the instance variable, name, private.

```
public class Student {
  private String name; // Student 's name.
  public double test1, test2, test3; // Grades on three tests .

  // Constructor for Student objects−provides a name for the Student.
  Student(String theName) {
      name = theName;
  }

  // Getter method for the private instance variable , name.
  public String getName() {
     return name;
  }

  // Compute average test grade.
  public double getAverage() {
     return (test1 + test2 + test3) / 3;
  }
}// end of class Student
```

An object of type Student contains information about some particular student. The constructor in this class has a parameter of type String, which specifies the name of that student. Objects of type Student can be created with statements such as:

```
std = new Student("John Smith");
std1 = new Student("Mary Jones");
```

In the original version of this class, the value of name had to be assigned by a program after it created the object of type Student. There was no guarantee that the programmer would always remember to set the name properly. In the new version of the class, there is no way to create a Student object except by calling the constructor, and that constructor automatically sets the name. The programmer's life is made easier, and whole hordes of frustrating bugs are squashed before they even have a chance to be born.

Another type of guarantee is provided by the **private** modifier. Since the instance variable, name, is **private**, there is no way for any part of the program outside the Student class to get at the name directly. The program sets the value of name, indirectly, when it calls the constructor. I've provided a method, getName(), that can be used from outside the class to find out the name of the student. But I haven't provided any setter method or other way to change the name. Once a student object is created, it keeps the same name as long as it exists.

1.2.5 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In JAVA, the destruction of objects takes place automatically.

An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("John Smith"); std = null;
```

In the first line, a reference to a newly created Student object is stored in the variable std. But in the next line, the value of std is changed, and the reference to the Student object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

JAVA uses a procedure called *garbage collection* to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage". In the above example, it was very easy to see that the Student object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a `memory leak`, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because JAVA uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

1.2.6 Everything is NOT an object

Wrapper Classes and Autoboxing

Recall that there are two kinds of types in JAVA: primitive types and object types (Classes). In some object-oriented languages, everything is an object. However in JAVA and in C++, the primitive types like **int** and **double** are not objects. This decision was made for memory and processing efficiency—it takes less memory to store an **int** than it is to store an object.

Sometimes, however, it is necessary to manipulate the primitive types as if they were objects. To make this possible, you can define *wrapper* classes whose sole aim is to contain one of the primitive types. They are used for creating objects that represent primitive type values.

For example the JAVA API contains the classes `Double` (that wraps a single **double**) and `Integer` that wraps a single integer. These classes contain various **static** methods including `Double.parseDouble` and `Integer.parseInteger` that are used to convert strings to numerical values. The `Character` class wraps a single **char** type. There is a similar class for each of the other primitive types, `Long`, `Short`, `Byte`, `Float`, and `Boolean`.

Remember that the primitive types are not classes, and values of primitive type are not objects. However, sometimes it's useful to treat a primitive value as if it were an object. You can't do that literally, but you can "wrap" the primitive type value in an object belonging to one of the wrapper classes.

For example, an object of type `Double` contains a single instance variable, of type **double**. The object is a wrapper for the **double** value. For example, you can create an object that wraps the **double** value `6.0221415e23` with

```
Double d = new Double(6.0221415e23);
```

The value of `d` contains the same information as the value of type **double**, but it is an object. If you want to retrieve the **double** value that is wrapped in the object, you can call the method `d.doubleValue()`. Similarly, you can wrap an **int** in an object of type `Integer`, a **boolean** value in an object of type `Boolean`, and so on. (As an example of where this would be useful, the collection classes that will be studied in Chapter 10 can only hold objects. If you want to add a primitive type value to a collection, it has to be put into a wrapper object first.)

In JAVA 5.0, wrapper classes have become easier to use. JAVA 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class. For example, if you use a value of type **int** in a context that requires an object of type

Integer, the **int** will automatically be wrapped in an Integer object. For example, you can say Integer answer = 42; and the computer will silently read this as if it were Integer answer = **new** Integer(42);.

This is called autoboxing. It works in the other direction, too. For example, if d refers to an object of type Double, you can use d in a numerical expression such as 2*d. The **double** value inside d is automatically unboxed and multiplied by 2. Autoboxing and unboxing also apply to method calls. For example, you can pass an actual parameter of type **int** to a method that has a formal parameter of type Integer. In fact, autoboxing and unboxing make it possible in many circumstances to ignore the difference between primitive types and objects.

The wrapper classes contain a few other things that deserve to be mentioned. Integer contains constants Integer.MIN_VALUE and Integer.MAX_VALUE, which are equal to the largest and smallest possible values of type **int**, that is, to -2147483648 and 2147483647 respectively. It's certainly easier to remember the names than the numerical values. There are similar named constants in Long, Short, and Byte. Double and Float also have constants named MIN_VALUE and MAX_VALUE. MAX_VALUE still gives the largest number that can be represented in the given type, but MIN_VALUE represents the smallest possible **positive** value. For type **double**, Double.MIN_VALUE is 4.9×10^{-324}. Since **double** values have only a finite accuracy, they can't get arbitrarily close to zero. This is the closest they can get without actually being equal to zero.

The class Double deserves special mention, since **double**s are so much more complicated than integers. The encoding of real numbers into values of type **double** has room for a few special values that are not real numbers at all in the mathematical sense. These values named constants in the class: Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, and Double.NaN. The infinite values can occur as values of certain mathematical expressions. For example, dividing a positive number by zero will give Double.POSITIVE_INFINITY. (It's even more complicated than this, actually, because the **double** type includes a value called "negative zero", written $-$0.0. Dividing a positive number by negative zero gives Double.NEGATIVE_INFINITY.) You also get Double.POSITIVE_INFINITY whenever the mathematical value of an expression is greater than Double.MAX_VALUE. For example, 1e200*1e200 is considered to be infinite. The value Double.NaN is even more interesting. "NaN" stands for Not a Number, and it represents an undefined value such as the square root of a negative number or the result of dividing zero by zero. Because of the existence of Double.NaN, no mathematical operation on real numbers will ever throw an exception; it simply gives Double.NaN as the result.

You can test whether a value, x, of type **double** is infinite or undefined by calling the boolean-valued static methods Double.isInfinite(x) and Double.isNaN(). (It's especially important to use Double.isNaN() to test for undefined values, because Double.NaN has really weird behavior when used with relational operators such as ==. In fact, the values of x == Double.NaN and x != Double.NaN are **both false**, no matter what the value of x, so you really can't use these expressions to test whether x is Double.NaN.)

Chapter 2

The Practice of Programming

Contents

2.1 Abstraction

ABSTRACTION IS A CENTRAL IDEA[1] in computer science and an understanding of this important term is crucial to successful programming.

> *Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structures.*
>
> Timothy Budd

Heres another definition from wikipedia.

[1] This discussion is based on a wikipedia article on abstraction: www.wikipedia.org.

In computer science, abstraction is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.

In general philosophical terminology, **abstraction** is the thought process wherein ideas are separated from objects. Our minds work mostly with abstractions. For example, when thinking about a chair, we do not have in mind a particular chair but an abstract idea of a chair—the concept of a chair. This why we are able to recognise an object as a chair even if it is different from any other chair we've seen previously. We form concepts of everyday objects and events by a process of abstraction where we remove unimportant details and concentrate on the essential attributes of the thing.

Abstraction in mathematics is the process of extracting the underlying essence of a mathematical concept, removing any dependence on real world objects with which it might originally have been connected, and generalising it so that it has wider applications. Many areas of mathematics began with the study of real world problems, before the underlying rules and concepts were identified and defined as abstract structures. For example, geometry has its origins in the calculation of distances and areas in the real world; statistics has its origins in the calculation of probabilities in gambling.

Roughly speaking, abstraction can be either that of control or data. Control abstraction is the abstraction of actions while data abstraction is that of data. For example, control abstraction in structured programming is the use of methods and formatted control flows. Data abstraction allows handling of data in meaningful ways. For example, it is the basic motivation behind datatype. Object-oriented programming can be seen as an attempt to abstract both data and control.

2.1.1 Control Abstraction

Control abstraction is one of the main purposes of using programming languages. Computer machines understand operations at the very low level such as moving some bits from one location of the memory to another location and producing the sum of two sequences of bits. Programming languages allow this to be done at a higher level. For example, consider the high-level expression/program statement: `a := (1 + 2) * 5`

To a human, this is a fairly simple and obvious calculation ("one plus two is three, times five is fifteen"). However, the low-level steps necessary to carry out this evaluation, and return the value "15", and then assign that value to the variable "a", are actually quite subtle and complex. The values need to be converted to binary representation (often a much more complicated task than one would think) and the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of one register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of "15" to the variable labeled "a", so that "a" can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label and the resultant location in physical or virtual memory, storing the binary representation of "15" to that memory location, etc. etc.

Without control abstraction, a programmer would need to specify all the register/ binary-level steps each time she simply wanted to add or multiply a couple of numbers and assign the result to a variable. This duplication of effort has two serious negative consequences:

- (a) it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed; and

- (b) it forces the programmer to program for the particular hardware and instruction set.

2.1.2 Data Abstraction

Data abstraction is the enforcement of a clear separation between the abstract properties of a data type and the concrete details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type–the interface to the data type–while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called `lookup table`, where keys are uniquely associated with values, and values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list. As far as client code is concerned, the abstract properties of the type are the same in each case.

2.1.3 Abstraction in Object-Oriented Programs

There are many important *layers of abstraction* in object-oriented programs. [2]

At the highest level, we view the program as a *community* of objects that interact with each other to achieve common goals. Each object provides a *service* that is used by other objects in the community. At this level we emphasize the lines of communication and cooperation and the interactions between the objects.

Another level of abstraction allows the grouping of related objects that work together. For example JAVA provides units called *packages* for grouping related objects. These units expose certain names to the system outside the unit while hiding certain features. For example the `java.net` package provides classes for networking applications. The JAVA Software Development Kit contains various packages that group different functionality.

The next levels of abstraction deal with interactions between individual objects. A useful way of thinking about objects, is to see them as providing a *service* to other objects. Thus, we can look at an object-oriented application as consisting of *service-providers* and *service-consumers* or *clients*. One level of abstraction looks at this relationship from the server side and the other looks at it from the client side.

Clients of the server are interested only in *what* the server provides (*its behaviour*) and not *how* it provides it (*its implementation*). The client only needs to know the public interface of a class it wants to use—what methods it can call, their input parameters, what they return and what they accomplish.

The next level of abstraction looks at the relationship from the server side. Here we consider the concrete implementation of the abstract behaviour. Here we are concerned with *how* the services are realized.

[2] This discussion is based on Chapter 2 of An Introduction to Object-Oriented Programming by Timothy Budd.

The last level of abstraction considers a single task in isolation i.e. a single method. Here we deal with the sequence of operations used to perform just this one activity.

Each level of abstraction is important at some point in the development of software. As programmers, we will constantly move from one level to another.

2.2 Methods as an Abstraction Mechanism

IN THIS SECTION we'll discuss an abstraction mechanism you're already familiar with: the method (variously called subroutines, procedures, and even functions).

2.2.1 Black Boxes

A Method is an abstraction mechanism and consists of instructions for performing some task, chunked together and given a name. ”Chunking” allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go though to perform that task, you just need to remember the name of the method. Whenever you want your program to perform the task, you just call the method. Methods are a major tool for dealing with complexity.

A method is sometimes said to be a ”black box” because you can't see what's ”inside” it (or, to be more precise, you usually don't want to see inside it, because then you would have to deal with all the complexity that the method is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of interface with the rest of the world, which allows some interaction between what's inside the box and what's outside. A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

> *The interface of a black box should be fairly straightforward, well-defined, and easy to understand.*

Your television, your car, your VCR, your refrigerator... are all examples of black boxes in the real world. You can turn your television on and off, change channels, and set the volume by using elements of the television's interface – dials, remote control, don't forget to plug in the power – without understanding anything about how the thing actually works. The same goes for a VCR, although if stories about how hard people find it to set the time on a VCR are true, maybe the VCR violates the simple interface rule.

Now, a black box does have an inside – the code in a method that actually performs the task, all the electronics inside your television set. The inside of a black box is called its implementation. The second rule of black boxes is that:

> *To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.*

In fact, it should be possible to change the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't even need to know about it – or even know what it means. Similarly, it should be possible to rewrite the inside of a method, to use more efficient code, for example, without affecting the programs that use that method.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as of the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn't need to know about any of that. The implementor just needs to make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

> *The implementor of a black box should not need to know anything about the larger systems in which the box will be used. In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.*

You should not think of an interface as just the physical connection between the box and the rest of the world. The interface also includes a specification of what the box does and how it can be controlled by using the elements of the physical interface. It's not enough to say that a TV set has a power switch; you need to specify that the power switch is used to turn the TV on and off!

To put this in computer science terms, the interface of a method has a *semantic* as well as a *syntactic* component. The syntactic part of the interface tells you just what you have to type in order to call the method. The semantic component specifies exactly what task the method will accomplish. To write a legal program, you need to know the syntactic specification of the method. To understand the purpose of the method and to use it effectively, you need to know the method's semantic specification. I will refer to both parts of the interface – syntactic and semantic – collectively as the contract of the method.

The contract of a method says, essentially, "Here is what you have to do to use me, and here is what I will do for you, guaranteed." When you write a method, the comments that you write for the method should make the contract very clear. (I should admit that in practice, methods' contracts are often inadequately specified, much to the regret and annoyance of the programmers who have to use them.)

You should keep in mind that methods are not the only example of black boxes in programming. For example, a class is also a black box. We'll see that a class can have a "public" part, representing its interface, and a "private" part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to methods.

2.2.2 Preconditions and Postconditions

When working with methods as building blocks, it is important to be clear about how a method interacts with the rest of the program. A convenient way to express the contract of a method is in terms of *preconditions* and *postconditions*.

> *The precondition of a method is something that must be true when the method is called, if the method is to work correctly.*

For example, for the built-in method `Math.sqrt(x)`, a precondition is that the parameter, x, is greater than or equal to zero, since it is not possible to take the square root of a negative number. In terms of a contract, a precondition represents an obligation of the caller of the method. If you call a method without meeting its precondition, then there is no reason to expect it to work properly. The program might crash or give incorrect results, but you can only blame yourself, not the method.

A postcondition of a method represents the other side of the contract. It is something that will be true after the method has run (assuming that its preconditions were met – and that there are no bugs in the method). The postcondition of the method `Math.sqrt()` is that the square of the value that is returned by this method is equal to the parameter that is provided when the method is called. Of course, this will only be true if the preconditiion – that the parameter is greater than or equal to zero – is met. A postcondition of the built-in method `System.out.print()` is that the value of the parameter has been displayed on the screen.

Preconditions most often give restrictions on the acceptable values of parameters, as in the example of `Math.sqrt(x)`. However, they can also refer to global variables that are used in the method. The postcondition of a method specifies the task that it performs. For a method, the postcondition should specify the value that the method returns. Methods are often described by comments that explicitly specify their preconditions and postconditions. When you are given a pre-written method, a statement of its preconditions and postcondtions tells you how to use it and what it does. When you are assigned to write a method, the preconditions and postconditions give you an exact specification of what the method is expected to do. Its a good idea to write preconditions and postconditions as part of comments are given in the form of Javadoc comments, but I will explicitly label the preconditions and postconditions. (Many computer scientists think that new doc tags @precondition and @postcondition should be added to the Javadoc system for explicit labeling of preconditions and postconditions, but that has not yet been done.

2.2.3 APIs and Packages

One of the important advantages of object-oriented programming is that it promotes reuse. When writing any piece of software, a programmer can use a large and growing body of pre-written software. The JAVA SDK (software development kit) consists of thousands of classes that can be used by programmers. So, learning the JAVA language means also being able to use this vast library of classes.

Toolboxes

Someone who wants to program for Macintosh computers – and to produce programs that look and behave the way users expect them to – must deal with the Macintosh Toolbox, a collection of well over a thousand different methods. There are methods for opening and closing windows, for drawing geometric figures and text to windows, for adding buttons to windows, and for responding to mouse clicks on the window. There are other methods for creating menus and for reacting to user selections from menus. Aside from the user interface, there are methods for opening files and reading data from them, for communicating over a network, for sending output to a printer, for handling communication between programs, and in general for doing all the standard things that a computer has to do. Microsoft Windows provides its own set of methods for programmers to use, and they are quite a bit different from the methods used on the Mac. Linux has several different GUI toolboxes for the programmer to choose from.

The analogy of a “toolbox” is a good one to keep in mind. Every programming project involves a mixture of innovation and reuse of existing tools. A programmer is given a set of tools to work with, starting with the set of basic tools that are built into the language: things like variables, assignment statements, if statements, and loops. To these, the programmer can add existing toolboxes full of methods that have already been written for performing certain tasks. These tools, if they are well-designed, can be used as true black boxes: They can be called to perform their assigned tasks without worrying about the particular steps they go through to accomplish those tasks. The innovative part of programming is to take all these tools and apply them to some particular project or problem (word-processing, keeping track of bank accounts, processing image data from a space probe, Web browsing, computer games,...). This is called applications programming.

A software toolbox is a kind of black box, and it presents a certain interface to the programmer. This interface is a specification of what methods are in the toolbox, what parameters they use, and what tasks they perform. This information constitutes the API, or Applications Programming Interface, associated with the toolbox. The Macintosh API is a specification of all the methods available in the Macintosh Toolbox. A company that makes some hardware device – say a card for connecting a computer to a network – might publish an API for that device consisting of a list of methods that programmers can call in order to communicate with and control the device. Scientists who write a set of methods for doing some kind of complex computation – such as solving “differential equations”, say – would provide an API to allow others to use those methods without understanding the details of the computations they perform.

The JAVA programming language is supplemented by a large, standard API. You’ve seen part of this API already, in the form of mathematical methods such as `Math.sqrt()`, the `String` data type and its associated methods, and the `System.out.print()` methods. The standard JAVA API includes methods for working with graphical user interfaces, for network communication, for reading and writing files, and more. It’s tempting to think of these methods as being built into the JAVA language, but they are technically methods that have been written and made available for use in JAVA programs.

JAVA is platform-independent. That is, the same program can run on platforms as diverse as Macintosh, Windows, Linux, and others. The same JAVA API must work on all these platforms. But notice that it is the **interface** that is platform-independent; the **implementation** varies from one platform to another. A JAVA system on a particular computer includes implementations of all the standard API methods. A JAVA program includes only **calls** to those methods. When the JAVA interpreter executes a program and encounters a call to one of the standard methods, it will pull up and execute the implementation of that method which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

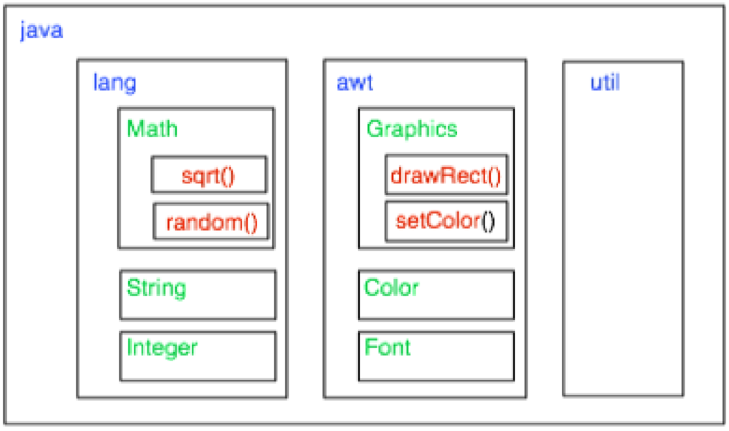
JAVA’s Standard Packages

Like all methods in JAVA, the methods in the standard API are grouped into classes. To provide larger-scale organization, classes in JAVA can be grouped into packages, which were introduced briefly in Subection2.6.4. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard JAVA API is implemented in several packages. One of these, which is named “java”, contains several non-GUI packages as well as the original AWT graphics user interface classes. Another package, “javax”, was added in JAVA version 1.2 and contains the classes used by the Swing graphical user interface and other additions to the API.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a “sub-package.” Both the java package and the javax package contain sub-packages. One of the sub-packages of java, for example, is called “awt”. Since awt is contained within java, its full name is actually java.awt. This package contains classes that represent GUI components such as buttons and menus in the AWT, the older of the two JAVA GUI toolboxes, which is no longer widely used. However, java.awt also contains a number of classes that form the foundation for all GUI programming, such as the Graphics class which provides methods for drawing on the screen, the Color class which represents colors, and the Font class which represents the fonts that are used to display characters on the screen. Since these classes are contained in the package java.awt, their full names are actually `java.awt.Graphics`, `java.awt.Color` and `java.awt.Font`. (I hope that by now you’ve gotten the hang of how this naming thing works in JAVA.) Similarly, `javax` contains a sub-package named `javax.swing`, which includes such classes as `javax.swing.JButton`, `javax.swing.JMenu`, and `javax.swing.JFrame`. The GUI classes in `javax.swing`, together with the foundational classes in `java.awt` are all part of the API that makes it possible to program graphical user interfaces in JAVA.

The java package includes several other sub-packages, such as `java.io`, which provides facilities for input/output, `java.net`, which deals with network communication, and java.util, which provides a variety of “utility” classes. The most basic package is called `java.lang`. This package contains fundamental classes such as `String`, `Math`, `Integer`, and `Double`.

It might be helpful to look at a graphical representation of the levels of nesting in the java package, its sub-packages, the classes in those sub-packages, and the methods in those classes. This is not a complete picture, since it shows only a very few of the many items in each element:



Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt()

The official documentation for the standard JAVA 5.0 API lists 165 different packages, including sub-packages, and it lists 3278 classes in these packages. Many of these are rather obscure or very specialized, but you might want to browse through the documentation to see what is available.

Even an expert programmer won't be familiar with the entire API, or even a majority of it. In this book, you'll only encounter several dozen classes, and those will be sufficient for writing a wide variety of programs.

Using Classes from Packages

Let's say that you want to use the class java.awt.Color in a program that you are writing. Like any class, java.awt.Color is a type, which means that you can use it declare variables and parameters and to specify the return type of a method. One way to do this is to use the full name of the class as the name of the type. For example, suppose that you want to declare a variable named rectColor of type java.awt.Color. You could say:

```
java.awt.Color  rectColor;
```

This is just an ordinary variable declaration of the form "type-name variable-name;". Of course, using the full name of every class can get tiresome, so JAVA makes it possible to avoid using the full names of a class by importing the class. If you put

```
import java.awt.Color;
```

at the beginning of a JAVA source code file, then, in the rest of the file, you can abbreviate the full name java.awt.Color to just the simple name of the class, Color. Note that the import line comes at the start of a file and is not inside any class. Although it is sometimes referred to as as a statement, it is more properly called an import directive since it is not a statement in the usual sense. Using this import directive would allow you to say

```
Color  rectColor;
```

to declare the variable. Note that the only effect of the import directive is to allow you to use simple class names instead of full "package.class" names; you aren't really importing anything substantial. If you leave out the import directive, you can still access the class – you just have to use its full name. There is a shortcut for importing all the classes from a given package. You can import all the classes from java.awt by saying

```
import java.awt.∗;
```

The "*" is a wildcard that matches every class in the package. (However, it does not match sub-packages; you **cannot** import the entire contents of all the sub-packages of the java packages by saying importjava.*.)

Some programmers think that using a wildcard in an import statement is bad style, since it can make a large number of class names available that you are not going to use and might not even know about. They think it is better to explicitly import each individual class that you want to use. In my own programming, I often use wildcards to import all the classes from the most relevant packages, and use individual imports when I am using just one or two classes from a given package.

In fact, any JAVA program that uses a graphical user interface is likely to use many classes from the java.awt and java.swing packages as well as from another package named java.awt.event, and I usually begin such programs with

```
import java.awt.∗;
import java.awt.event.∗;
import javax.swing.∗;
```

A program that works with networking might include: "**import** java.net.∗;", while one that reads or writes files might use "**import** java.io.∗;". (But when you start importing lots of packages in this way, you have to be careful about one thing: It's possible for two classes that are in different packages to have the same name. For example, both the java.awt package and the java.util package contain classes named List. If you import both java.awt.∗ and java.util.∗, the simple name List will be ambiguous. If you try to declare a variable of type List, you will get a compiler error message about an ambiguous class name. The solution is simple: use the full name of the class, either java.awt.List or java.util.List. Another solution, of course, is to use import to import the individual classes you need, instead of importing entire packages.)

Because the package java.lang is so fundamental, all the classes in java.lang are **automatically** imported into every program. It's as if every program began with the statement "**import** java.lang.∗;". This is why we have been able to use the class name String instead of java.lang.String, and Math.sqrt() instead of java.lang.Math.sqrt(). It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named utilities. Then the source code file that defines those classes must begin with the line

```
package utilities;
```

This would come even before any import directive in that file. Furthermore, the source code file would be placed in a folder with the same name as the package. A class that is in a package automatically has access to other classes in the same package; that is, a class doesn't have to import the package in which it is defined.

In projects that define large numbers of classes, it makes sense to organize those classes into packages. It also makes sense for programmers to create new packages as toolboxes that provide functionality and API's for dealing with areas not covered in the standard JAVA API. (And in fact such "toolmaking" programmers often have more prestige than the applications programmers who use their tools.)

However, I will not be creating any packages in this textbook. For the purposes of this book, you need to know about packages mainly so that you will be able to import the standard packages. These packages are always available to the programs that you write. You might wonder where the standard classes are actually located. Again, that can depend to some extent on the version of JAVA that you are using, but in the standard JAVA 5.0, they are stored in jar files in a subdirectory of the main JAVA installation directory. A jar (or "JAVA archive") file is a single file that can contain many classes. Most of the standard classes can be found in a jar file named classes.jar. In fact, JAVA programs are generally distributed in the form of jar files, instead of as individual class files.

Although we won't be creating packages explicitly, **every** class is actually part of a package. If a class is not specifically placed in a package, then it is put in something called the default package, which has no name. All the examples that you see in this book are in the default package.

2.3 Introduction to Error Handling

IN ADDITION TO THE CONTROL STRUCTURES that determine the normal flow of control in a program, Java has a way to deal with "exceptional" cases that throw the flow of control off its normal track. When an error occurs during the execution of a program, the default behavior is to terminate the program and to print an error message. However, Java makes it possible to “catch” such errors and program a response different from simply letting the program crash. This is done with the try..catch statement. In this section, we will take a preliminary, incomplete look at using try..catch to handle errors.

Exceptions

The term exception is used to refer to the type of error that one might want to handle with a **try..catch**. An exception is an exception to the normal flow of control in the program. The term is used in preference to “error” because in some cases, an exception might not be considered to be an error at all. You can sometimes think of an exception as just another way to organize a program.

Exceptions in Java are represented as objects of type `Exception`. Actual exceptions are defined by subclasses of `Exception`. Different subclasses represent different types of exceptions We will look at only two types of exception in this section: `NumberFormatException` and `IllegalArgumentException`.

A `NumberFormatException` can occur when an attempt is made to convert a string into a number. Such conversions are done, for example, by `Integer.parseInt` and `Integer.parseDouble` . Consider the method call `Integer.parseInt(str)` where `str` is a variable of type `String`. If the value of `str` is the string “42”, then the method call will correctly convert the string into the **int** 42. However, if the value of `str` is, say, “fred”, the method call will fail because “fred” is not a legal string representation of an int value. In this case, an exception of type `NumberFormatException` occurs. If nothing is done to handle the exception, the program will crash.

An `IllegalArgumentException` can occur when an illegal value is passed as a parameter to a method. For example, if a method requires that a parameter be greater than or equal to zero, an `IllegalArgumentException` might occur when a negative value is passed to the method. How to respond to the illegal value is up to the person who wrote the method, so we can’t simply say that every illegal parameter value will result in an `IllegalArgumentException`. However, it is a common response.

One case where an `IllegalArgumentException` can occur is in the `valueOf` method of an enumerated type. Recall that this method tries to convert a string into one of the values of the enumerated type. If the string that is passed as a parameter to valueOf is not the name of one of the enumerated type’s value, then an `IllegalArgumentException` occurs. For example, given the enumerated type

```
enum Toss { HEADS, TAILS };
```

`Toss.valueOf("HEADS")` correctly returns `Toss.HEADS`, but `Toss.valueOf(‘‘FEET′′)` results in an `IllegalArgumentException`.

try . . . catch

When an exception occurs, we say that the exception is “thrown”. For example, we say that `Integer.parseInt(str)` throws an exception of type `NumberFormatException` when the value of `str` is illegal. When an exception is thrown, it is possible to “catch” the exception and prevent it from crashing the program. This is done with a **try..catch** statement. In somewhat simplified form, the syntax for a **try..catch** is:

```
try {
   statements-1
}
catch ( exception-class-name  variable-name ) {
   statements-2
}
```

The exception-class-name in the catch clause could be NumberFormatException, IllegalArgumentException, or some other exception class. When the computer executes this statement, it executes the statements in the try part. If no error occurs during the execution of statements-1, then the computer just skips over the catch part and proceeds with the rest of the program. However, if an exception of type exception-class-name occurs during the execution of statements-1, the computer immediately jumps to the catch part and executes statements-2, skipping any remaining statements in statements-1. During the execution of statements-2, the variable-name represents the exception object, so that you can, for example, print it out. At the end of the catch part, the computer proceeds with the rest of the program; the exception has been caught and handled and does not crash the program. Note that only one type of exception is caught; if some other type of exception occurs during the execution of statements-1, it will crash the program as usual.

(By the way, note that the braces, { and }, are part of the syntax of the try..catch statement. They are required even if there is only one statement between the braces. This is different from the other statements we have seen, where the braces around a single statement are optional.)

As an example, suppose that str is a variable of type String whose value might or might not represent a legal real number. Then we could say:

```
try {
   double x;
   x = Double.parseDouble(str);
   System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "Not a legal number." );
}
```

If an error is thrown by the call to Double.parseDouble(str), then the output statement in the try part is skipped, and the statement in the catch part is executed.

It's not always a good idea to catch exceptions and continue with the program. Often that can just lead to an even bigger mess later on, and it might be better just to let the exception crash the program at the point where it occurs. However, sometimes it's possible to recover from an error. For example, suppose that we have the enumerated type

```
enum Day {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY}
```

and we want the user to input a value belonging to this type. TextIO does not know about this type, so we can only read the user's response as a string. The method Day.valueOf can be used to convert the user's response to a value of type Day. This will throw an exception of type IllegalArgumentException if the user's response is not the name of one of the values of type Day, but we can respond to the error easily enough by asking the user to enter another response. Here is a code segment that does this. (Converting the user's response to upper case will allow responses such as "Monday" or "monday" in addition to "MONDAY".)

```
Scanner keyboard = new Scanner(System.in);
Day  weekday;  // User's response as a value of type Day.
while ( true ) {
   String response;  // User's response as a String.
   keyboard.put("Please enter a day of the week: ");
   response = keyboard.nextLinen();
   response = response.toUpperCase();
   try {
      weekday = Day.valueOf(response);
      break;
   }
   catch ( IllegalArgumentException e ) {
      System.out.println( response +
               " is not the name of a day of the week." );
   }
}
```

The break statement will be reached only if the user's response is acceptable, and so the loop will end only when a legal value has been assigned to weekday.

2.4 Javadoc

Good programming means extensive comments and documentation. At the very least, explain the method of each instance variable, and for each method explain its purpose, parameters, returns, where applicable. You should also strive for a consistent layout and for expressive variable names.

A program that is well-documented is much more valuable than the same program without the documentation. Java comes with a tool called **javadoc** that can make it easier to produce the documentation is a readable and organized format. **JavaDoc** is a program that will automatically extract/generate an HTML help-page from code that is properly commented. In particular, it is designed produce a help file that, for a class, lists the methods, constructors and public fields, and for each method explains what it does together with pre-conditions, post-conditions, the meaning of the parameters, exceptions that may be thrown and other things.

Javadoc is especially useful for documenting classes and packages of classes that are meant to be used by other programmers. A programmer who wants to use prewritten classes shouldn't need to search through the source code to find out how to use them. If the documentation in the source code is in the correct format, javadoc can separate out the documentation and make it into a set of web pages. The web pages are automatically formatted and linked into an easily browseable Web site. Sun Microsystem's documentation for the standard Java API was produced using javadoc.

Javadoc documentation is prepared from special comments that are placed in the Java source code file. Recall that one type of Java comment begins with /* and ends with */. A Javadoc comment takes the same form, but it begins with /** rather than simply /*.

```
/**
* This method prints a 3N+1 sequence to standard output, using
* startingValue as the initial value of N.  It also prints the number
* of terms in the sequence. The value of the parameter, startingValue,
* must be a positive integer.
*/
```

```
static void print3NSequence(int startingValue) { ...
```

You can have Javadoc comments for methods, member variables, and for classes. The Javadoc comment always immediately precedes the thing it is commenting on. Like any comment, a Javadoc comment is ignored by the computer when the file is compiled. But there is a tool called javadoc that reads Java source code files, extracts any Javadoc comments that it finds, and creates a set of Web pages containing the comments in a nicely formatted, interlinked form. By default, javadoc will only collect information about public classes, methods, and member variables, but it allows the option of creating documentation for non-public things as well. If javadoc doesn't find any Javadoc comment for something, it will construct one, but the comment will contain only basic information such as the name and type of a member variable or the name, return type, and parameter list of a method. This is syntactic information. To add information about semantics and pragmatics, you have to write a Javadoc comment.

In addition to normal text, the comment can contain certain special codes. For one thing, the comment can contain HTML mark-up commands. (HTML is the language that is used to create web pages, and Javadoc comments are meant to be shown on web pages.) The javadoc tool will copy any HTML commands in the comments to the web pages that it creates. As an example, you can add <p> to indicate the start of a new paragraph. (Generally, in the absence of HTML commands, blank lines and extra spaces in the comment are ignored.)

In addition to HTML commands, Javadoc comments can include doc tags, which are processed as commands by the javadoc tool. A doc tag has a name that begins with the character . I will only discuss three tags: @param, @return, and @throws. These tags are used in Javadoc comments for methods to provide information about its parameters, its return value, and the exceptions that it might throw. These tags are always placed at the end of the comment, after any description of the method itself. The syntax for using them is:

```
@param parameter–name description–of–parameter

@return description–of–return–value

@throws exception–class–name description–of–exception
```

The descriptions can extend over several lines. The description ends at the next tag or at the end of the comment. You can include a @param tag for every parameter of the method and a @throws for as many types of exception as you want to document. You should have a @return tag only for a non-void method. These tags do not have to be given in any particular order. Here is an example that doesn't do anything exciting but that does use all three types of doc tag:

If you want to create Web-page documentation, you need to run the javadoc tool. You can use javadoc in a command line interface similarly to the way that the javac and java commands are used. Javadoc can also be applied in the Eclipse integrated development environment: Just right-click the class or package that you want to document in the Package Explorer, select ”Export,” and select ”Javadoc” in the window that pops up. Consult the documentation for more details.

```
/**
 * This method computes the area of a rectangle, given its width
 * and its height. The length and the width should be positive numbers.
 * @param width the length of one side of the rectangle
 * @param height the length the second side of the rectangle
 * @return the area of the rectangle
 * @throws IllegalArgumentException if either the width or the height
 *    is a negative number.
 */
public static double areaOfRectangle( double length, double width ) {
    if ( width < 0  ||  height < 0 )
       throw new IllegalArgumentException("Sides must have positive length.");
    double area;
    area = width * height;
    return area;
}
```

2.5 Creating Jar Files

As the final topic for this chapter, we look again at jar files. Recall that a jar file is a "java archive" that can contain a number of class files. When creating a program that uses more than one class, it's usually a good idea to place all the classes that are required by the program into a jar file, since then a user will only need that one file to run the program. Jar files can also be used for stand-alone applications. In fact, it is possible to make a so-called `executable jar file`. A user can run an executable jar file in much the same way as any other application, usually by double-clicking the icon of the jar file. (The user's computer must have a correct version of Java installed, and the computer must be configured correctly for this to work. The configuration is usually done automatically when Java is installed, at least on Windows and Mac OS.)

The question, then, is how to create a jar file. The answer depends on what programming environment you are using. There are two basic types of programming environment – command line and IDE. Any IDE (Integrated Programming Environment) for Java should have a command for creating jar files. In the Eclipse IDE, for example, it's done as follows: In the Package Explorer pane, select the programming project (or just all the individual source code files that you need). Right-click on the selection, and choose "Export" from the menu that pops up. In the window that appears, select "JAR file" and click "Next". In the window that appears next, enter a name for the jar file in the box labeled "JAR file". (Click the "Browse" button next to this box to select the file name using a file dialog box.) The name of the file should end with ".jar". If you are creating a regular jar file, not an executable one, you can hit "Finish" at this point, and the jar file will be created. You could do this, for example, if the jar file contains an applet but no main program. To create an executable file, hit the "Next" button *twice* to get to the "Jar Manifest Specification" screen. At the bottom of this screen is an input box labeled "Main class". You have to enter the name of the class that contains the `main()` method that will be run when the jar file is executed. If you hit the "Browse" button next to the "Main class" box, you can select the class from a list of classes that contain `main()` methods. Once you've selected the main class, you can click the "Finish" button to create the executable jar file.

It is also possible to create jar files on the command line. The Java Development

Kit includes a command-line program named `jar` that can be used to create jar files. If all your classes are in the default package (like the examples in this book), then the `jar` command is easy to use. To create a non-executable jar file on the command line, change to the directory that contains the class files that you want to include in the jar. Then give the command `jar cf JarFileName.jar *.class` where `JarFileName` can be any name that you want to use for the jar file. The "`*`" in "`*.class`" is a wildcard that makes `*.class` match every class file in the current directory. This means that all the class files in the directory will be included in the jar file. If you want to include only certain class files, you can name them individually, separated by spaces. (Things get more complicated if your classes are not in the default package. In that case, the class files must be in subdirectories of the directory in which you issue the jar file.)

Making an executable jar file on the command line is a little more complicated. There has to be some way of specifying which class contains the `main()` method. This is done by creating a `manifest file`. The manifest file can be a plain text file containing a single line of the form `Main−Class: ClassName` where `ClassName` should be replaced by the name of the class that contains the `main()` method. For example, if the `main()` method is in the class `MosaicDrawFrame`, then the manifest file should read "`Main−Class: MosaicDrawFrame`". You can give the manifest file any name you like. Put it in the same directory where you will issue the `jar` command, and use a command of the form `jar cmf ManifestFileName JarFileName.jar *.class` to create the jar file. (The `jar` command is capable of performing a variety of different operations. The first parameter to the command, such as "`cf`" or "`cmf`", tells it which operation to perform.)

By the way, if you have successfully created an executable jar file, you can run it on the command line using the command "`java −jar`". For example:

```
java −jar JarFileName.jar
```

2.6 Creating Abstractions

IN THIS SECTION, we look at some specific examples of object-oriented design in a domain that is simple enough that we have a chance of coming up with something reasonably reusable. Consider card games that are played with a standard deck of playing cards (a so-called "poker" deck, since it is used in the game of poker).

2.6.1 Designing the classes

When designing object-oriented software, a crucial first step is to identify the objects that will make up the application. One approach to do this is to identify the nouns in the problem description. These become candidates for objects. Next we can identify verbs in the description: these suggest methods for the objects.

Consider the following description of a card game:

> In a typical card game, each player gets a *hand* of cards. The deck is shuffled and cards are dealt one at a time from the deck and added to the players' hands. In some games, cards can be removed from a hand, and new cards can be added. The game is won or lost depending on the value (ace, 2, ..., king) and suit (spades, diamonds, clubs, hearts) of the cards that a player receives.

If we look for nouns in this description, there are several candidates for objects: game, player, hand, card, deck, value, and suit. Of these, the value and the suit of a card are simple values, and they will just be represented as instance variables in a Card object. In a complete program, the other five nouns might be represented by classes. But let's work on the ones that are most obviously reusable: card, hand, and deck.

If we look for verbs in the description of a card game, we see that we can shuffle a deck and deal a card from a deck. This gives use us two candidates for instance methods in a Deck class: shuffle() and dealCard(). Cards can be added to and removed from hands. This gives two candidates for instance methods in a Hand class: addCard() and removeCard(). Cards are relatively passive things, but we need to be able to determine their suits and values. We will discover more instance methods as we go along.

The Deck Class:

First, we'll design the deck class in detail. When a deck of cards is first created, it contains 52 cards in some standard order. The Deck class will need a constructor to create a new deck. The constructor needs no parameters because any new deck is the same as any other. There will be an instance method called shuffle() that will rearrange the 52 cards into a random order. The dealCard() instance method will get the next card from the deck. This will be a method with a return type of Card, since the caller needs to know what card is being dealt. It has no parameters – when you deal the next card from the deck, you don't provide any information to the deck; you just get the next card, whatever it is. What will happen if there are no more cards in the deck when its dealCard() method is called? It should probably be considered an error to try to deal a card from an empty deck, so the deck can throw an exception in that case. But this raises another question: How will the rest of the program know whether the deck is empty? Of course, the program could keep track of how many cards it has used. But the deck itself should know how many cards it has left, so the program should just be able to ask the deck object. We can make this possible by specifying another instance method, cardsLeft(), that returns the number of cards remaining in the deck. This leads to a full specification of all the methods in the Deck class:

```
/** Constructor.  Create a shuffled deck of cards.
* @precondition: None
* @postcondition: A deck of 52, shuffled cards is created.*/
public Deck()

/** Shuffle all cards in the deck into a random order.
* @precondition: None
* @postcondition: The existing deck of cards with the cards
*       in random order.*/
public void shuffle()

/** Returns the size of the deck
* @return the number of cards that are still left in the deck.
* @precondition: None
* @postcondition: The deck is unchanged.  */
public int size()
```

```
/** Determine if this deck is empty
 * @return true if this deck has no cards left in the deck.
 * @precondition: None
 * @postcondition: The deck is unchanged.  */
public boolean isEmpty()

/** Deal one card from this deck
 * @return a Card from the deck.
 * @precondition: The deck is not empty
 * @postcondition: The deck has one less card.   */
public Card deal()
```

This is everything you need to know in order to use the Deck class. Of course, it doesn't tell us how to write the class. This has been an exercise in design, not in programming. With this information, you can use the class in your programs without understanding the implementation. The description above is a contract between the users of the class and implementors of the class—it is the "*public interface*" of the class.

The Hand Class:

We can do a similar analysis for the Hand class. When a hand object is first created, it has no cards in it. An addCard() instance method will add a card to the hand. This method needs a parameter of type Card to specify which card is being added. For the removeCard() method, a parameter is needed to specify which card to remove. But should we specify the card itself ("Remove the ace of spades"), or should we specify the card by its position in the hand ("Remove the third card in the hand")? Actually, we don't have to decide, since we can allow for both options. We'll have two removeCard() instance methods, one with a parameter of type Card specifying the card to be removed and one with a parameter of type **int** specifying the position of the card in the hand. (Remember that you can have two methods in a class with the same name, provided they have different types of parameters.) Since a hand can contain a variable number of cards, it's convenient to be able to ask a hand object how many cards it contains. So, we need an instance method getCardCount() that returns the number of cards in the hand. When I play cards, I like to arrange the cards in my hand so that cards of the same value are next to each other. Since this is a generally useful thing to be able to do, we can provide instance methods for sorting the cards in the hand. Here is a full specification for a reusable Hand class:

```
/** Create a Hand object that is initially empty.
 * @precondition: None
 * @postcondition: An empty hand object is created.*/
public Hand() {

/** Discard all cards from the hand, making the hand empty.
 * @precondition: None
 * @postcondition: The hand object is empty. */
public void clear() {

/** If the specified card is in the hand, it is removed.
 * @param c the Card to be removed.
 * @precondition: c is a Card object and is non-null.
 * @postcondition: The specified card is removed if it exists.*/
public void removeCard(Card c) {
 /** Add the card c to the hand.
 * @param The Card to be added.
 * @precondition: c is a Card object and is non-null.
 * @postcondition: The hand object contains the Card c
 *      and now has one more card.
 * @throws NullPointerException is thrown if c is not a
 *      Card or is null.    */
 public void addCard(Card c) {

/** Remove the card in the specified position from the hand.
 * @param position the position of the card that is to be removed,
 *        where positions start from zero.
 * @precondition: position is valid i.e. 0 < position < number cards
 * @postcondition: The card in the specified position is removed
 *      and there is one less card in the hand.
 * @throws IllegalArgumentException if the position does not exist in
 *        the hand.   */
 public void removeCard(int position) {

 /** Return the number of cards in the hand.
  * @return int the number of cards in the hand
 * @precondition: none
 * @postcondition: No change in state of Hand.   */
 public int getCardCount() {

/** Gets the card in a specified position in the hand.
 *  (Note that this card is not removed from the hand!)
 * @param position the position of the card that is to be returned
 * @return Card the Card at the specified position.
 * @throws IllegalArgumentException if position does not exist.
 * @precondition: position is valid i.e. 0 < position < number cards.
 * @postcondition: The state of the Hand is unchanged.   */
 public Card getCard(int position) {

 /** Sorts the cards in the hand in suit order and in value order
  * within suits.  Note that aces have the lowest value, 1.
  * @precondition: none
  * @postcondition: Cards of the same
  *     suit are grouped together, and within a suit the cards
  *     are sorted by value.    */
 public void sortBySuit() {

 /** Sorts the cards in the hand so that cards are sorted into
  * order of increasing value.  Cards with the same value
  * are sorted by suit. Note that aces are considered
  * to have the lowest value.
  * @precondition: none
  * @postcondition: Cards are sorted in order of increasing value.*/
 public void sortByValue() {
```

The Card Class

The class will have a constructor that specifies the value and suit of the card that is being created. There are four suits, which can be represented by the integers 0, 1, 2, and 3. It would be tough to remember which number represents which suit, so I've defined named constants in the Card class to represent the four possibilities. For example, Card.SPADES is a constant that represents the suit, spades. (These constants are declared to be **public final static** ints. It might be better to use an enumerated type, but for now we will stick to integer-valued constants. I'll return to the question of using enumerated types in this example at the end of the chapter.) The possible values of a card are the numbers 1, 2, ..., 13, with 1 standing for an ace, 11 for a jack, 12 for a queen, and 13 for a king. Again, I've defined some named constants to represent the values of aces and face cards.

A Card object can be constructed knowing the value and the suit of the card. For example, we can call the constructor with statements such as:

```
card1 = new Card( Card.ACE, Card.SPADES );  // Construct ace of spades.
card2 = new Card( 10, Card.DIAMONDS );   // Construct 10 of diamonds.
card3 = new Card( v, s );  // This is OK, as long as v and s
                           // are integer expressions.
```

A Card object needs instance variables to represent its value and suit. I've made these **private** so that they cannot be changed from outside the class, and I've provided getter methods getSuit() and getValue() so that it will be possible to discover the suit and value from outside the class. The instance variables are initialized in the constructor, and are never changed after that. In fact, I've declared the instance variables suit and value to be **final**, since they are never changed after they are initialized. (An instance variable can be declared **final** provided it is either given an initial value in its declaration or is initialized in every constructor in the class.)

Finally, I've added a few convenience methods to the class to make it easier to print out cards in a human-readable form. For example, I want to be able to print out the suit of a card as the word “Diamonds”, rather than as the meaningless code number 2, which is used in the class to represent diamonds. Since this is something that I'll probably have to do in many programs, it makes sense to include support for it in the class. So, I've provided instance methods getSuitAsString() and getValueAsString() to return string representations of the suit and value of a card. Finally, I've defined the instance method toString() to return a string with both the value and suit, such as “Queen of Hearts”. Recall that this method will be used whenever a Card needs to be converted into a String, such as when the card is concatenated onto a string with the + operator. Thus, the statement

```
System.out.println( "Your card is the " + card );
```

is equivalent to

```
System.out.println( "Your card is the " + card.toString() );
```

If the card is the queen of hearts, either of these will print out
‘‘Your card is the Queen of Hearts''.

Here is the complete Card class. It is general enough to be highly reusable, so the work that went into designing, writing, and testing it pays off handsomely in the long run.

```
/** An object of type Card represents a playing card from a
 * standard Poker deck, including Jokers.  The card has a suit, which
 * can be spades, hearts, diamonds, clubs, or joker.  A spade, heart,
 * diamond, or club has one of the 13 values: ace, 2, 3, 4, 5, 6, 7,
 * 8, 9, 10, jack, queen, or king.  Note that "ace" is considered to be
 * the smallest value.  A joker can also have an associated value;
 * this value can be anything and can be used to keep track of several
 * different jokers.
 */
public class Card {
   public final static int SPADES = 0;   // Codes for the 4 suits.
   public final static int HEARTS = 1;
   public final static int DIAMONDS = 2;
   public final static int CLUBS = 3;

   public final static int ACE = 1;      // Codes for the non-numeric cards.
   public final static int JACK = 11;    // Cards 2 through 10 have their
   public final static int QUEEN = 12;   // numerical values for their codes.
   public final static int KING = 13;

   /** This card's suit, one of the constants SPADES, HEARTS, DIAMONDS,
    * CLUBS.  The suit cannot be changed after the card is
    * constructed.   */
   private final int suit;

   /** The card's value.  For a normal cards, this is one of the values
    * 1 through 13, with 1 representing ACE. The value cannot be changed
    * after the card is constructed.   */
   private final int value;


   /** Creates a card with a specified suit and value.
    * @param theValue the value of the new card.  For a regular card (non-joker),
    * the value must be in the range 1 through 13, with 1 representing an Ace.
    * You can use the constants Card.ACE, Card.JACK, Card.QUEEN, and Card.KING.
    * For a Joker, the value can be anything.
    * @param theSuit the suit of the new card.  This must be one of the values
    * Card.SPADES, Card.HEARTS, Card.DIAMONDS, Card.CLUBS, or Card.JOKER.
    * @throws IllegalArgumentException if the parameter values are not in the
    * permissible ranges   */
   public Card(int theValue, int theSuit) {
      if (theSuit != SPADES && theSuit != HEARTS && theSuit != DIAMONDS &&
            theSuit != CLUBS )
         throw new IllegalArgumentException("Illegal playing card suit");
      if (theValue < 1 || theValue > 13)
         throw new IllegalArgumentException("Illegal playing card value");
      value = theValue;
      suit = theSuit;
   }

   /** Returns the suit of this card.
    * @returns the suit, which is one of the constants Card.SPADES,
    * Card.HEARTS, Card.DIAMONDS, Card.CLUBS   */
   public int getSuit() {
      return suit;
   }

   /** Returns the value of this card.
    * @return the value, which is one the numbers 1 through 13. */
   public int getValue() {
      return value;
   }
```

```
    /** Returns a String representation of the card's suit.
     * @return one of the strings "Spades", "Hearts", "Diamonds", "Clubs". */
    public String getSuitAsString() {
       switch ( suit ) {
       case SPADES:   return "Spades";
       case HEARTS:   return "Hearts";
       case DIAMONDS: return "Diamonds";
       case CLUBS:    return "Clubs";
       default:       return "Null";
       }
    }

    /** Returns a String representation of the card's value.
     * @return for a regular card, one of the strings "Ace", "2",
     * "3", ..., "10", "Jack", "Queen", or "King".    */
    public String getValueAsString() {
        switch ( value ) {
          case 1:   return "Ace";
          case 2:   return "2";
          case 3:   return "3";
          case 4:   return "4";
          case 5:   return "5";
          case 6:   return "6";
          case 7:   return "7";
          case 8:   return "8";
          case 9:   return "9";
          case 10:  return "10";
          case 11:  return "Jack";
          case 12:  return "Queen";
          default:  return "King";
          }
    }

    /** Returns a string representation of this card, including both
     * its suit and its value. Sample return values
     * are: "Queen of Hearts", "10 of Diamonds", "Ace of Spades",    */
    public String toString() {
          return getValueAsString() + " of " + getSuitAsString();
    }

 } // end class Card
```

2.7 Example: A Simple Card Game

We will finish this section by presenting a complete program that uses the Card and Deck classes. The program lets the user play a very simple card game called **High-Low**. A deck of cards is shuffled, and one card is dealt from the deck and shown to the user. The user predicts whether the next card from the deck will be higher or lower than the current card. If the user predicts correctly, then the next card from the deck becomes the current card, and the user makes another prediction. This continues until the user makes an incorrect prediction. The number of correct predictions is the user's score.

My program has a method that plays one game of HighLow. This method has a return value that represents the user's score in the game. The main()method lets the user play several games of HighLow. At the end, it reports the user's average score.

Note that the method that plays one game of HighLow returns the user's score in the game as its return value. This gets the score back to the main program, where it is needed. Here is the program:

```
import java.util.Scanner;
/**
 * This program lets the user play HighLow, a simple card game
 * that is described in the output statements at the beginning of
 * the main() method.  After the user plays several games,
 * the user's average score is reported.
 */
public class HighLow {

   Scanner keyboard = new Scanner(System.in);

   public static void main(String[] args) {


      System.out.println("This program lets you play the simple card game,");
      System.out.println("HighLow.  A card is dealt from a deck of cards.");
      System.out.println("You have to predict whether the next card will be");
      System.out.println("higher or lower.  Your score in the game is the");
      System.out.println("number of correct predictions you make before");
      System.out.println("you guess wrong.");
      System.out.println();

      int gamesPlayed = 0;     // Number of games user has played.
      int sumOfScores = 0;     // The sum of all the scores from
                               //      all the games played.
      double averageScore;     // Average score, computed by dividing
                               //      sumOfScores by gamesPlayed.
      boolean playAgain;       // Record user's response when user is
                               //   asked whether he wants to play
                               //   another game.

      do {
         int scoreThisGame;        // Score for one game.
         scoreThisGame = play();   // Play the game and get the score.
         sumOfScores += scoreThisGame;
         gamesPlayed++;
         System.out.print("Play again? ");
         playAgain = keyboard.nextBoolean();
      } while (playAgain);

      averageScore = ((double)sumOfScores) / gamesPlayed;

      System.out.println();
      System.out.println("You played " + gamesPlayed + " games.");
      System.out.printf("Your average score was %1.3f.\n", averageScore);

   }  // end main()
```

```
/**
 * Let's the user play one game of HighLow, and returns the
 * user's score on that game.  The score is the number of
 * correct guesses that the user makes.
 */
private static int play() {

   Deck deck = new Deck();  // Get a new deck of cards, and
                            //   store a reference to it in
                            //   the variable, deck.

   Card currentCard;  // The current card, which the user sees.

   Card nextCard;   // The next card in the deck.  The user tries
                    //    to predict whether this is higher or lower
                    //    than the current card.

   int correctGuesses ;  // The number of correct predictions the
                         //   user has made.  At the end of the game,
                         //   this will be the user's score.

   char guess;   // The user's guess.  'H' if the user predicts that
                 //   the next card will be higher, 'L' if the user
                 //   predicts that it will be lower.

   deck.shuffle();  // Shuffle the deck into a random order before
                    //    starting the game.

   correctGuesses = 0;
   currentCard = deck.dealCard();
   System.out.println("The first card is the " + currentCard);

   while (true) {  // Loop ends when user's prediction is wrong.

      /* Get the user's prediction, 'H' or 'L' (or 'h' or 'l'). */

      System.out.print("Will the next card be higher(H) or lower(L)? ");
      do {
          guess = keyboard.next().charAt(0);
          guess = Character.toUpperCase(guess);
          if (guess != 'H' && guess != 'L')
             System.out.print("Please respond with H or L:  ");
      } while (guess != 'H' && guess != 'L');

      /* Get the next card and show it to the user. */

      nextCard = deck.dealCard();
      System.out.println("The next card is " + nextCard);
```

```
         /* Check the user's prediction. */

         if (nextCard.getValue() == currentCard.getValue()) {
            System.out.println("The value is the same as the previous card.");
            System.out.println("You lose on ties.  Sorry!");
            break;  // End the game.
         }
         else if (nextCard.getValue() > currentCard.getValue()) {
            if (guess == 'H') {
                System.out.println("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                System.out.println("Your prediction was incorrect.");
                break;  // End the game.
            }
         }
         else {  // nextCard is lower
            if (guess == 'L') {
                System.out.println("Your prediction was correct.");
                correctGuesses++;
            }
            else {
                System.out.println("Your prediction was incorrect.");
                break;  // End the game.
            }
         }

         /* To set up for the next iteration of the loop, the nextCard
            becomes the currentCard, since the currentCard has to be
            the card that the user sees, and the nextCard will be
            set to the next card in the deck after the user makes
            his prediction.  */

         currentCard = nextCard;
         System.out.println();
         System.out.println("The card is " + currentCard);

      } // end of while loop

      System.out.println();
      System.out.println("The game is over.");
      System.out.println("You made " + correctGuesses
                                            + " correct predictions.");
      System.out.println();

      return correctGuesses;

   }  // end play()
} // end class
```

Chapter 3

Tools for Working with Abstractions

3.1 Introduction to Software Engineering

THE DIFFICULTIES INHERENT with the development of software has led many computer scientists to suggest that software development should be treated as an engineering activity. They argue for a disciplined approach where the software engineer uses carefully thought out methods and processes.

The term software engineering has several meanings (from wikipedia):

* As the broad term for all aspects of the practice of computer programming, as opposed to the theory of computer programming, which is called computer science;

* As the term embodying the advocacy of a specific approach to computer programming, one that urges that it be treated as an engineering profession rather than an art or a craft, and advocates the codification of recommended practices in the form of software engineering methodologies.

* Software engineering is

  (1) "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software," and
  (2) "the study of approaches as in (1)." IEEE Standard 610.12

3.1.1 Software Engineering Life-Cycles

A decades-long goal has been to find repeatable, predictable processes or methodologies that improve productivity and quality of software. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management is proving difficult.

Software engineering requires performing many tasks, notably the following, some of which may not seem to directly produce software.

- **Requirements Analysis** Extracting the requirements of a desired software product is the first task in creating it. While customers probably believe they know what the software is to do, it may require skill and experience in software engineering to recognize incomplete, ambiguous or contradictory requirements.

- **Specification** Specification is the task of precisely describing the software to be written, usually in a mathematically rigorous way. In reality, most successful specifications are written to understand and fine-tune applications that were already well-developed. Specifications are most important for external interfaces, that must remain stable.

- **Design and Architecture** Design and architecture refer to determining how software is to function in a general way without being involved in details. Usually this phase is divided into two sub-phases.

- **Coding** Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion.

- **Testing** Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.

- **Documentation** An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. Documentation is most important for external interfaces.

- **Maintenance** Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. Not only may it be necessary to add code that does not fit the original design but just determining how software works at some point after it is completed may require significant effort by a software engineer. About 2/3 of all software engineering work is maintenance, but this statistic can be misleading. A small part of that is fixing bugs. Most maintenance is extending systems to do new things, which in many ways can be considered new work. In comparison, about 2/3 of all civil engineering, architecture, and construction work is maintenance in a similar way.

3.1.2 Object-oriented Analysis and Design

A large programming project goes through a number of stages, starting with specification of the problem to be solved, followed by analysis of the problem and design of a program to solve it. Then comes coding, in which the program's design is expressed in some actual programming language. This is followed by testing and debugging of the program. After that comes a long period of maintenance, which means fixing any new problems that are found in the program and modifying it to adapt it to changing requirements. Together, these stages form what is called the software life cycle. (In the real world, the ideal of consecutive stages is seldom if ever achieved. During the analysis stage, it might turn out that the specifications are incomplete or inconsistent. A problem found during testing requires at least a brief return to the coding stage. If the problem is serious enough, it might even require a new design. Maintenance usually involves redoing some of the work from previous stages....)

Large, complex programming projects are only likely to succeed if a careful, systematic approach is adopted during all stages of the software life cycle. The systematic approach to programming, using accepted principles of good design, is called software engineering. The software engineer tries to efficiently construct programs that verifyably meet their specifications and that are easy to modify if necessary. There is a wide range of "methodologies" that can be applied to help in the systematic design of programs. (Most of these methodologies seem to involve drawing little boxes to represent program components, with labeled arrows to represent relationships among the boxes.)

We have been discussing object orientation in programming languages, which is relevant to the coding stage of program development. But there are also object-oriented methodologies for analysis and design. The question in this stage of the software life cycle is, How can one discover or invent the overall structure of a program? As an example of a rather simple object-oriented approach to analysis and design, consider this advice: Write down a description of the problem. Underline all the nouns in that description. The nouns should be considered as candidates for becoming classes or objects in the program design. Similarly, underline all the verbs. These are candidates for methods. This is your starting point. Further analysis might uncover the need for more classes and methods, and it might reveal that subclassing can be used to take advantage of similarities among classes.

This is perhaps a bit simple-minded, but the idea is clear and the general approach can be effective: Analyze the problem to discover the concepts that are involved, and create classes to represent those concepts. The design should arise from the problem itself, and you should end up with a program whose structure reflects the structure of the problem in a natural way.

3.1.3 Object Oriented design

OOP design rests on three principles:

- **Abstraction**: Ignore the details. In philosophical terminology, abstraction is the thought process wherein ideas are distanced from objects. In computer science, abstraction is a mechanism and practice to reduce and factor out details so that one can focus on few concepts at a time.

  Abstraction uses a strategy of simplification, wherein formerly concrete details are left ambiguous, vague, or undefined. [wikipedia:Abstraction]

- **Modularization**: break into pieces. A module can be defined variously, but generally must be a component of a larger system, and operate within that system independently from the operations of the other components. Modularity is the property of computer programs that measures the extent to which they have been composed out of separate parts called modules.

  Programs that have many direct interrelationships between any two random parts of the program code are less modular than programs where those relationships occur mainly at well-defined interfaces between modules.

- **Information hiding**: separate the implementation and the function. The principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Protecting a design decision involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change).

We strive for **responsibility-driven design**: each class should be responsible for its own data. We strive for **loose coupling**: each class is largely independent and communicates with other classes via a small well-defined interface. We strive for **cohesion**: each class performs one and only one task (for readability, reuse).

3.2 Class-Responsibility-Collaboration cards

CLASS-RESPONSIBILITY-COLLABORATION CARDS (CRC cards) are a brainstorming tool used in the design of object-oriented software. They were proposed by Ward Cunningham. They are typically used when first determining which classes are needed and how they will interact.

CRC cards are usually created from index cards on which are written:

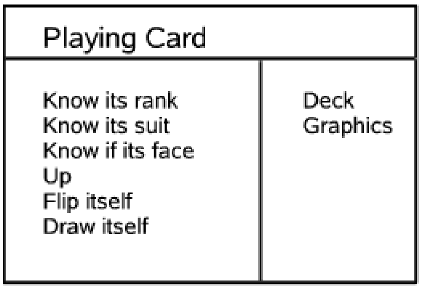
1. The class name.

2. The package name (if applicable).

3. The responsibilities of the class.

4. The names of other classes that the class will collaborate with to fulfill its responsibilities.

For example consider the CRC Card for a `Playing Card` class:



Playing Card	
Know its rank Know its suit Know if its face Up Flip itself Draw itself	Deck Graphics

The responsibilities are listed on the left. The classes that the `Playing Card` class will collaborate with are listed on the right.

The idea is that class design is undertaken by a team of developers. CRC cards are used as a brainstorming technique. The team attempts to determine all the classes and their responsibilities that will be needed for the application. The team runs through various usage scenarios of the application. For e.g. one such scenario for a game of cards may be “the player picks a card from the deck and hand adds it to his hand”. The team uses the CRC cards to check if this scenario can be handled by the responsibilites assigned to the classes. In this way, the design is refined until the team agrees on a set of classes and has agreed on their responsibilities.

Using a small card keeps the complexity of the design at a minimum. It focuses the designer on the essentials of the class and prevents him from getting into its details and inner workings at a time when such detail is probably counter-productive. It also forces the designer to refrain from giving the class too many responsibilities.

3.3 The Unified Modelling Language

THIS SECTION WILL GIVE YOU A QUICK OVERVIEW of the basics of UML. It is taken from the user documentation of the UML tool `Umbrello` and wikipedia. Keep in mind that this is not a comprehensive tutorial on UML but rather a brief introduction to UML which can be read as a UML tutorial. If you would like to learn more about the Unified Modelling Language, or in general about software analysis and design, refer to one of the many books available on the topic. There are also a lot of tutorials on the Internet which you can take as a starting point.

The Unified Modelling Language (UML) is a diagramming language or notation to specify, visualize and document models of Object Oriented software systems. UML is not a development method, that means it does not tell you what to do first and what to do next or how to design your system, but it helps you to visualize your design and communicate with others. UML is controlled by the Object Management Group (OMG) and is the industry standard for graphically describing software. The OMG have recently completed version 2 of the UML standard—known as UML2.

UML is designed for Object Oriented software design and has limited use for other programming paradigms.

UML is not a method by itself, however it was designed to be compatible with the leading object-oriented software development methods of its time (e.g., OMT, Booch, Objectory). Since UML has evolved, some of these methods have been recast to take advantage of the new notation (e.g., OMT) and new methods have been created based on UML. Most well known is the Rational Unified Process (RUP) created by the Rational Software Corporation.

3.3.1 Modelling

There are three prominent parts of a system's model:

- Functional Model

  Showcases the functionality of the system from the user's Point of View. Includes Use Case Diagrams.

- Object Model

  Showcases the structure and substructure of the system using objects, attributes, operations, and associations. Includes Class Diagrams.

- Dynamic Model

  Showcases the internal behavior of the system. Includes Sequence Diagrams, Activity Diagrams and State Machine Diagrams.

UML is composed of many model elements that represent the different parts of a software system. The UML elements are used to create diagrams, which represent a certain part, or a point of view of the system. In UML 2.0 there are 13 types of diagrams. Some of the more important diagrams are:

- *Use Case Diagrams* show actors (people or other users of the system), use cases (the scenarios when they use the system), and their relationships

- *Class Diagrams* show classes and the relationships between them

- *Sequence Diagrams* show objects and a sequence of method calls they make to other objects.
- *Collaboration Diagrams* show objects and their relationship, putting emphasis on the objects that participate in the message exchange
- *State Diagrams* show states, state changes and events in an object or a part of the system
- *Activity Diagrams* show activities and the changes from one activity to another with the events occurring in some part of the system
- *Component Diagrams* show the high level programming components (such as KParts or Java Beans).
- *Deployment Diagrams* show the instances of the components and their relationships.

3.3.2 Use Case Diagrams

Use Case Diagrams describe the relationships and dependencies between a group of **Use Cases** and the Actors participating in the process.

It is important to notice that Use Case Diagrams are not suited to represent the design, and cannot describe the internals of a system. Use Case Diagrams are meant to facilitate the communication with the future users of the system, and with the customer, and are specially helpful to determine the required features the system is to have. Use Case Diagrams tell, *what* the system should do but do not–and cannot–specify *how* this is to be achieved.

A **Use Case** describes–from the point of view of the actors–a group of activities in a system that produces a concrete, tangible result.

Use Cases are descriptions of the typical interactions between the users of a system and the system itself. They represent the external interface of the system and specify a form of requirements of what the system has to do (remember, only what, not how).

When working with Use Cases, it is important to remember some simple rules:

- Each Use Case is related to at least one actor
- Each Use Case has an initiator (i.e. an actor)
- Each Use Case leads to a relevant result (a result with a business value)

An *a*ctor is an external entity (outside of the system) that interacts with the system by participating (and often initiating) a Use Case. Actors can be in real life people (for example users of the system), other computer systems or external events.

Actors do not represent the *physical* people or systems, but their *role* . This means that when a person interacts with the system in different ways (assuming different roles) he will be represented by several actors. For example a person that gives customer support by the telephone and takes orders from the customer into the system would be represented by an actor “Support Staff” and an actor “Sales Representative”

*U*se Case Descriptions are textual narratives of the Use Case. They usually take the form of a note or a document that is somehow linked to the Use Case, and explains the processes or activities that take place in the Use Case.

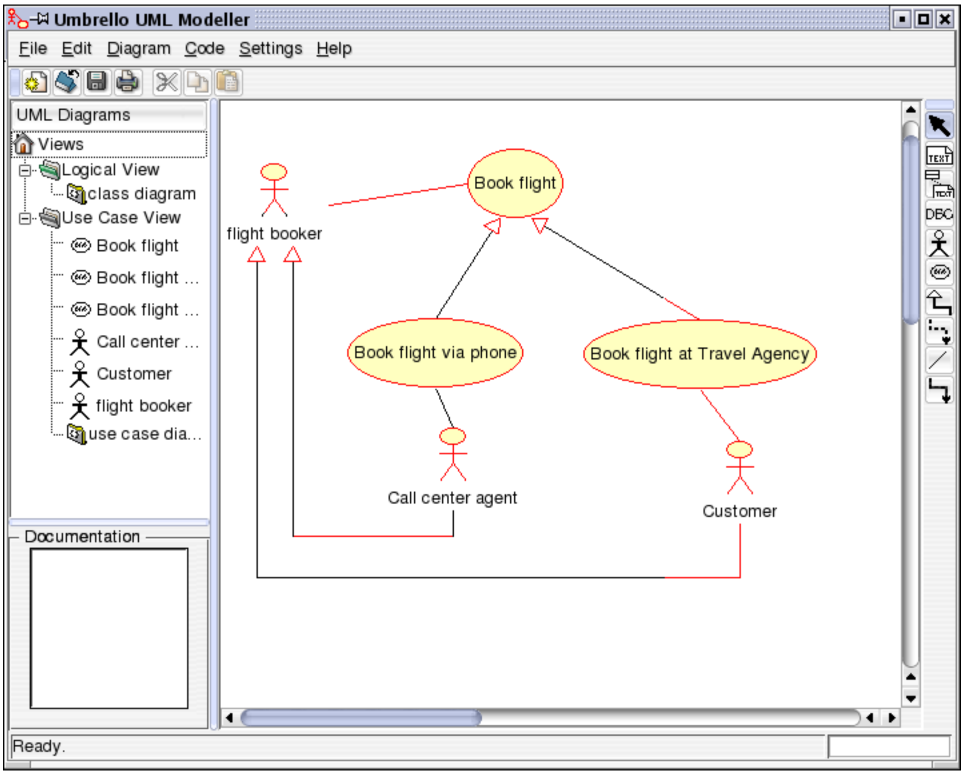


Figure 3.1: Umbrello UML Modeller showing a Use Case Diagram

3.3.3 Class Diagrams

Class Diagrams show the different classes that make up a system and how they relate to each other. Class Diagrams are said to be ”static” diagrams because they show the classes, along with their methods and attributes as well as the static relationships between them: which classes ”know” about which classes or which classes ”are part” of another class, but do not show the method calls between them.

A **Class** defines the attributes and the methods of a set of objects. All objects of this class (instances of this class) share the same behavior, and have the same set of attributes (each object has its own set). The term ”Type” is sometimes used instead of Class, but it is important to mention that these two are not the same, and Type is a more general term.

In UML, Classes are represented by rectangles, with the name of the class, and can also show the attributes and operations of the class in two other ”compartments” inside the rectangle.

In UML, **Attributes** are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

- + Stands for *public* attributes
- # Stands for *protected* attributes
- - Stands for *private* attributes

Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as Attributes, display their visibility:

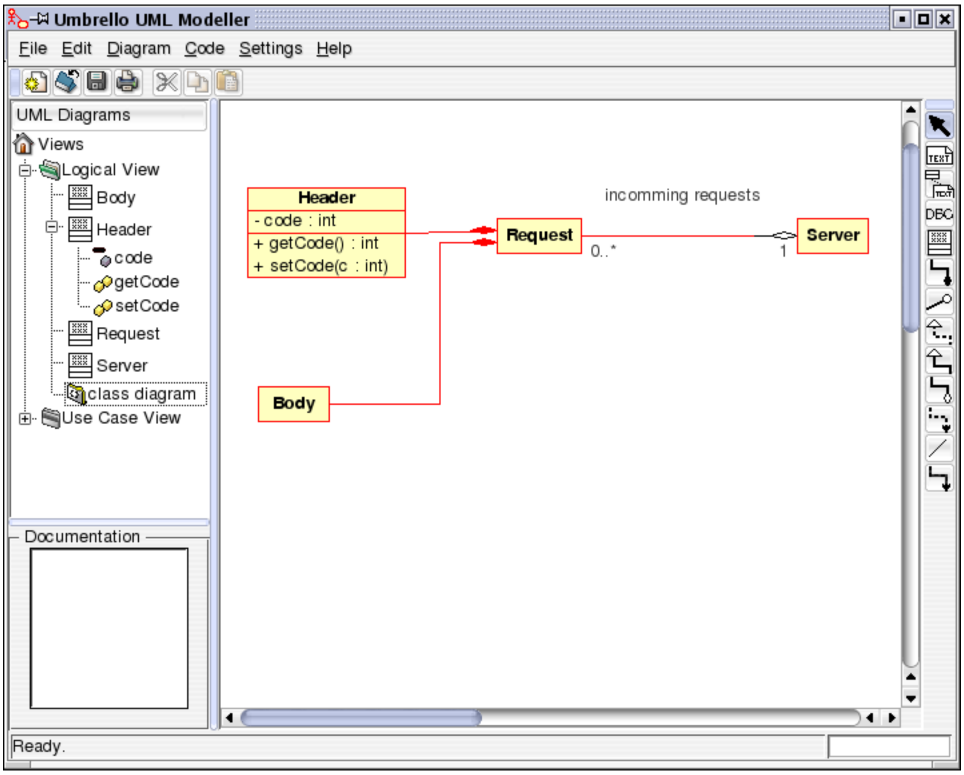


Figure 3.2: Umbrello UML Modeller showing a Class Diagram

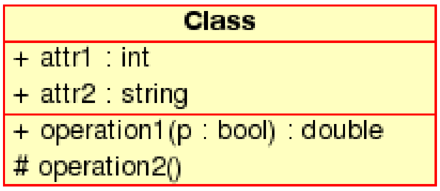


Figure 3.3: Visual representation of a Class in UML

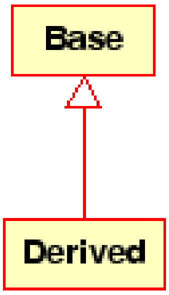


Figure 3.4: Visual representation of a generalization in UML

- + Stands for *public* operations
- # Stands for *protected* operations
- - Stands for *private* operations

Class Associations

Classes can relate (be associated with) to each other in different ways:

Inheritance is one of the fundamental concepts of Object Orientated programming, in which a class ”gains” all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own.

- **Generalization** In UML, a *Generalization* association between two classes puts them in a hierarchy representing the concept of inheritance of a derived class from a base class. In UML, Generalizations are represented by a line connecting the two classes, with an arrow on the side of the base class.
- **Association** An association represents a relationship between classes, and gives the common semantics and structure for many types of ”connections” between objects.

  Associations are the mechanism that allows objects to communicate to each other. It describes the connection between different classes (the connection between the actual objects is called object connection, or *link* .

  Associations can have a role that specifies the purpose of the association and can be uni- or bidirectional (indicates if the two objects participating in the relationship can send messages to the other, of if only one of them knows about the other). Each end of the association also has a multiplicity value, which dictates how many objects on this side of the association can relate to one object on the other side.

  In UML, associations are represented as lines connecting the classes participating in the relationship, and can also show the role and the multiplicity of each of the participants. Multiplicity is displayed as a range [min..max] of non-negative values, with a star (*) on the maximum side representing infinite.
- **Aggregations** Aggregations are a special type of associations in which the two participating classes don’t have an equal status, but make a ”whole-part” relationship. An Aggregation describes how the class that takes the role of the

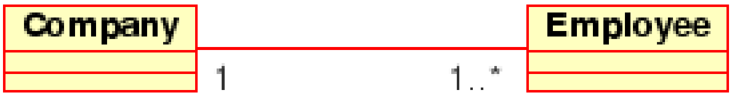


Figure 3.5: Visual representation of an Association in UML **Aggregation**

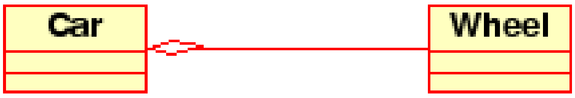


Figure 3.6: Visual representation of an Aggregation relationship in UML

whole, is composed (has) of other classes, which take the role of the parts. For Aggregations, the class acting as the whole always has a multiplicity of one.

In UML, Aggregations are represented by an association that shows a rhomb on the side of the whole.

- **Composition** Compositions are associations that represent *very strong* aggregations. This means, Compositions form whole-part relationships as well, but the relationship is so strong that the parts cannot exist on its own. They exist only inside the whole, and if the whole is destroyed the parts die too.

  In UML, Compositions are represented by a solid rhomb on the side of the whole.

Other Class Diagram Items Class diagrams can contain several other items besides classes.

- **Interfaces** are abstract classes which means instances can not be directly created of them. They can contain operations but no attributes. Classes can inherit from interfaces (through a realisation association) and instances can then be made of these diagrams.
- **Datatypes** are primitives which are typically built into a programming language. Common examples include integers and booleans. They can not have relationships to classes but classes can have relationships to them.
- **Enums** are a simple list of values. A typical example is an enum for days of the week. The options of an enum are called Enum Literals. Like datatypes they can not have relationships to classes but classes can have relationships to them.
- **Packages** represent a namespace in a programming language. In a diagram they are used to represent parts of a system which contain more than one class, maybe hundereds of classes.

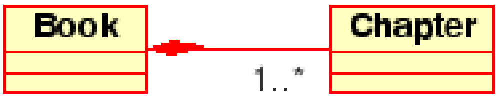


Figure 3.7:

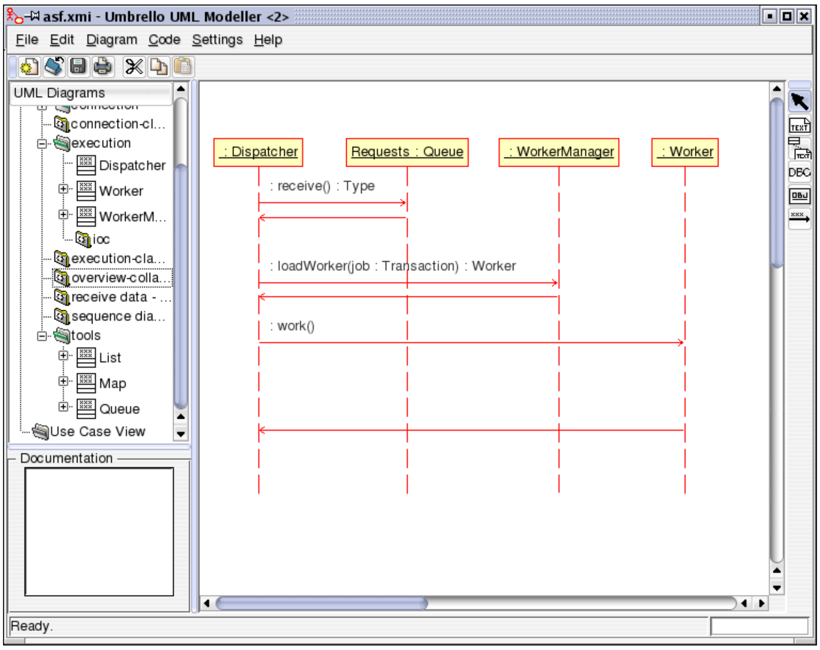


Figure 3.8: Umbrello UML Modeller showing a Sequence Diagram

3.3.4 Sequence Diagrams

Sequence Diagrams show the message exchange (i.e. method call) between several Objects in a specific time-delimited situation. Objects are instances of classes. Sequence Diagrams put special emphasis in the order and the times in which the messages to the objects are sent.

In Sequence Diagrams objects are represented through vertical dashed lines, with the name of the Object on the top. The time axis is also vertical, increasing downwards, so that messages are sent from one Object to another in the form of arrows with the operation and parameters name.

Messages can be either synchronous, the normal type of message call where control is passed to the called object until that method has finished running, or asynchronous where control is passed back directly to the calling object. Synchronous messages have a vertical box on the side of the called object to show the flow of program control.

3.3.5 Collaboration Diagrams

Collaboration Diagrams show the interactions occurring between the objects participating in a specific situation. This is more or less the same information shown by Sequence Diagrams but there the emphasis is put on how the interactions occur in time while the Collaboration Diagrams put the relationships between the objects and their topology in the foreground.

In Collaboration Diagrams messages sent from one object to another are represented by arrows, showing the message name, parameters, and the sequence of the message. Collaboration Diagrams are specially well suited to showing a specific program flow or situation and are one of the best diagram types to quickly demonstrate or explain one process in the program logic.

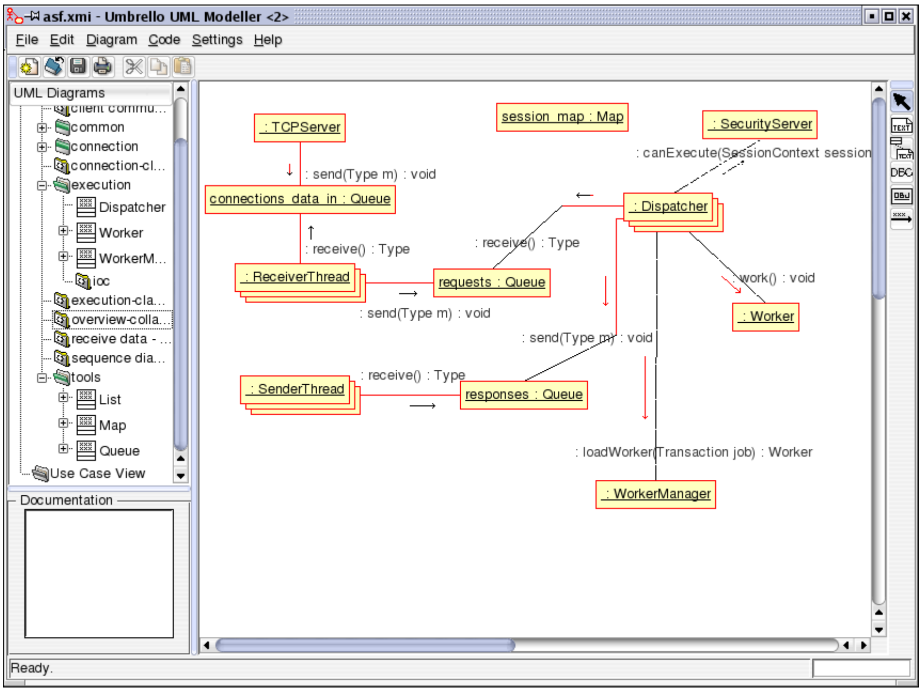


Figure 3.9: Umbrello UML Modeller showing a Collaboration Diagram

3.3.6 State Diagram

State Diagrams show the different states of an Object during its life and the stimuli that cause the Object to change its state.

State Diagrams view Objects as *state machines* or finite automata that can be in one of a set of finite states and that can change its state via one of a finite set of stimuli. For example an Object of type *NetServer* can be in one of following states during its life:

- Ready
- Listening
- Working
- Stopped

and the events that can cause the Object to change states are

- Object is created
- Object receives message listen
- A Client requests a connection over the network
- A Client terminates a request
- The request is executed and terminated
- Object receives message stop ... etc

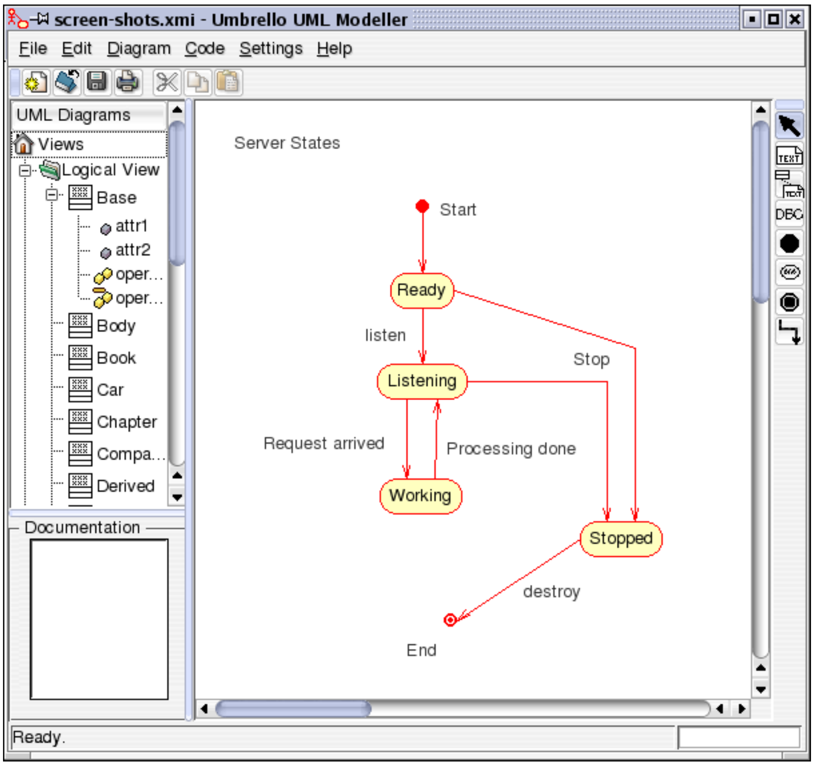


Figure 3.10: Umbrello UML Modeller showing a State Diagram

Activity Diagram

Activity Diagrams describe the sequence of activities in a system with the help of Activities. Activity Diagrams are a special form of State Diagrams, that only (or mostly) contains Activities.

Activity Diagrams are always associated to a *Class* , an *Operation* or a *Use Case* .

Activity Diagrams support sequential as well as parallel Activities. Parallel execution is represented via Fork/Wait icons, and for the Activities running in parallel, it is not important the order in which they are carried out (they can be executed at the same time or one after the other)

Component Diagrams

Component Diagrams show the software components (either component technologies such as KParts, CORBA components or Java Beans or just sections of the system which are clearly distinguishable) and the artifacts they are made out of such as source code files, programming libraries or relational database tables.

Deployment Diagrams

Deployment diagrams show the runtime component instances and their associations. They include Nodes which are physical resources, typically a single computer. They also show interfaces and objects (class instances).

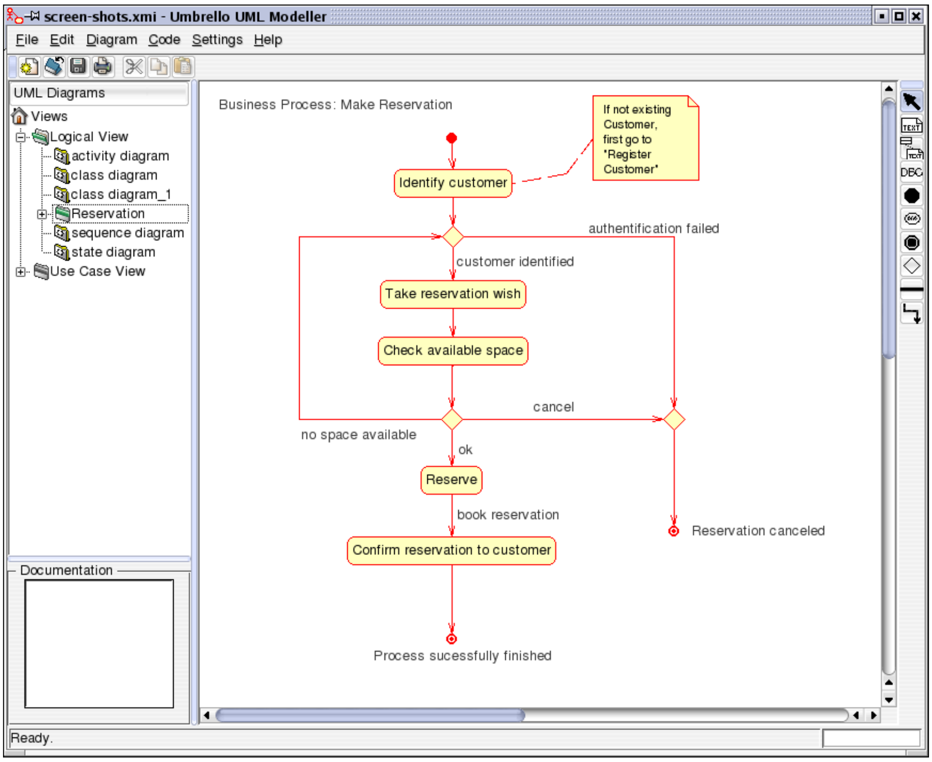


Figure 3.11: Umbrello UML Modeller showing an Activity Diagram

Chapter 4

Inheritance, Polymorphism, and Abstract Classes

Contents

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming–the idea that really distinguishes it from traditional programming–is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using `inheritance` and `polymorphism`.

4.1 Extending Existing Classes

IN DAY-TO-DAY PROGRAMMING, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation: There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be *extended* to make a subclass. The syntax for this is

```
public class Subclass-name
  extends Existing-class-name {
     .
     .    // Changes and additions.
     .
}
```

As an example, suppose you want to write a program that plays the card game, Blackjack. You can use the Card, Hand, and Deck classes developed previously. However, a hand in the game of Blackjack is a little different from a hand of cards in general, since it must be possible to compute the “value” of a Blackjack hand according to the rules of the game. The rules are as follows: The value of a hand is obtained by adding up the values of the cards in the hand.

- The value of a numeric card such as a three or a ten is its numerical value.
- The value of a Jack, Queen, or King is 10.
- The value of an Ace can be either 1 or 11. An Ace should be counted as 11 unless doing so would put the total value of the hand over 21. Note that this means that the second, third, or fourth Ace in the hand will always be counted as 1.

One way to handle this is to extend the existing Hand class by adding a method that computes the Blackjack value of the hand. Here’s the definition of such a class:

```
public class BlackjackHand extends Hand {

    /**
     * Computes and returns the value of this hand in the game
     * of Blackjack.
     */
    public int getBlackjackValue() {

        int val;      // The value computed for the hand.
        boolean ace;  // This will be set to true if the
                      //   hand contains an ace.
        int cards;    // Number of cards in the hand.

        val = 0;
        ace = false;
        cards = getCardCount();

        for ( int i = 0;  i < cards;  i++ ) {
                // Add the value of the i-th card in the hand.
            Card card;    // The i-th card;
            int cardVal;  // The blackjack value of the i-th card.
            card = getCard(i);
            cardVal = card.getValue();  // The normal value, 1 to 13.
            if (cardVal > 10) {
                cardVal = 10;   // For a Jack, Queen, or King.
            }
            if (cardVal == 1) {
                ace = true;     // There is at least one ace.
            }
            val = val + cardVal;
         }
```

```
            // Now, val is the value of the hand, counting any ace as 1.
            // If there is an ace, and if changing its value from 1 to
            // 11 would leave the score less than or equal to 21,
            // then do so by adding the extra 10 points to val.

            if ( ace == true  &&  val + 10 <= 21 )
                val = val + 10;

            return val;

        }  // end getBlackjackValue()

    } // end class BlackjackHand
```

Since BlackjackHand is a subclass of Hand, an object of type BlackjackHand contains all the instance variables and instance methods defined in Hand, plus the new instance method named getBlackjackValue(). For example, if bjh is a variable of type BlackjackHand, then all of the following are legal: bjh.getCardCount(), bjh.removeCard(0), and bjh.getBlackjackValue(). The first two methods are defined in Hand, but are inherited by BlackjackHand.

Inherited variables and methods from the Hand class can also be used in the definition of BlackjackHand (except for any that are declared to be **private**, which prevents access even by subclasses). The statement "cards = getCardCount();" in the above definition of getBlackjackValue() calls the instance method getCardCount(), which was defined in Hand.

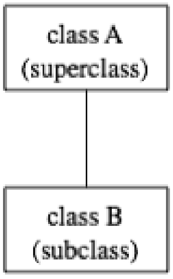
Extending existing classes is an easy way to build on previous work. We'll see that many standard classes have been written specifically to be used as the basis for making subclasses.

Access modifiers such as **public** and **private** are used to control access to members of a class. There is one more access modifier, **protected**, that comes into the picture when subclasses are taken into consideration. When **protected** is applied as an access modifier to a method or member variable in a class, that member can be used in subclasses – direct or indirect – of the class in which it is defined, but it cannot be used in non-subclasses. (There is one exception: A **protected** member can also be accessed by any class in the same package as the class that contains the protected member. Recall that using no access modifier makes a member accessible to classes in the same package, and nowhere else. Using the **protected** modifier is strictly more liberal than using no modifier at all: It allows access from classes in the same package and from **subclasses** that are not in the same package.)

When you declare a method or member variable to be **protected**, you are saying that it is part of the implementation of the class, rather than part of the public interface of the class. However, you are allowing subclasses to use and modify that part of the implementation.

For example, consider a PairOfDice class that has instance variables die1 and die2 to represent the numbers appearing on the two dice. We could make those variables **private** to make it impossible to change their values from outside the class, while still allowing read access through getter methods. However, if we think it possible that PairOfDice will be used to create subclasses, we might want to make it possible for subclasses to change the numbers on the dice. For example, a GraphicalDice subclass that draws the dice might want to change the numbers at other times besides when the dice are rolled. In that case, we could make die1 and die2 **protected**, which would allow the subclass to change their values without making them public to the rest of the world. (An even better idea would be to define **protected** setter methods for the variables. A setter method could, for example, ensure that the value that is being assigned to the variable is in the legal range 1 through 6.)

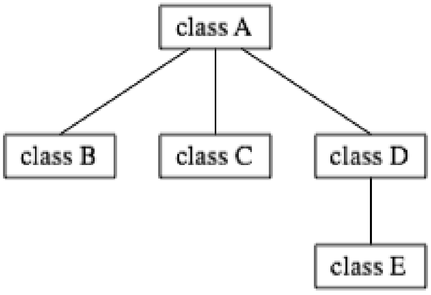
4.2 Inheritance and Class Hierarchy



The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived **class** and base **class** are used instead of subclass and superclass; this is the common terminology inC++.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass.

In Java, to create a class named “B” as a subclass of a class named “A”, you would write

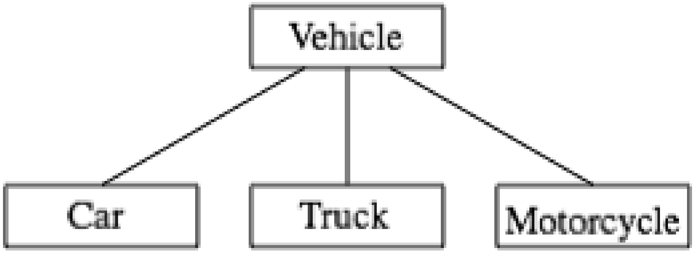
```
class B extends A {
     .
     .  // additions to, and modifications of,
     .  // stuff inherited from class A
     .
}
```



Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as “sibling classes,” share some structures and behaviors – namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several “generations” of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes forms a small **class** hierarchy.

4.3 Example: Vehicles

Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named Vehicle to represent all types of vehicles. Since cars, trucks, and motorcycles are types of vehicles, they would be represented by subclasses of the Vehicle class, as shown in this class hierarchy diagram:



The Vehicle class would include instance variables such as registrationNumber and owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles. The three subclasses of Vehicle – Car, Truck, and Motorcycle – could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxels, and the Motorcycle class could have a boolean variable hasSidecar. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in Java program would look, in outline, like this (although in practice, they would probably be **public** classes, defined in separate files):

```
class Vehicle {
   int registrationNumber;
   Person owner;  // (Assuming that a Person class has been defined!)
   void transferOwnership(Person newOwner) {
       . . .
   }
   . . .
}
class Car extends Vehicle {
   int numberOfDoors;
   . . .
}
class Truck extends Vehicle {
   int numberOfAxels;
   . . .
}
class Motorcycle extends Vehicle {
   boolean hasSidecar;
   . . .
}
```

Suppose that myCar is a variable of type Car that has been declared and initialized with the statement Car myCar = **new** Car(); Given this declaration, a program could refer to myCar.numberOfDoors, since numberOfDoors is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle. This means that myCar.registrationNumber, myCar.owner, and myCar.transferOwnership() also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Caror Truck or Motorcycle is automatically an object of type Vehicle too. This brings us to the following Important Fact:

> *A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.*

The practical effect of this is that an object of type Car can be assigned to a variable of type Vehicle; i.e. it would be legal to say Vehicle myVehicle = myCar; or even Vehicle myVehicle = **new** Car();.

After either of these statements, the variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object "remembers" that it is in fact a Car, and not **just** a Vehicle. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the **instanceof** operator. The test: **if** (myVehicle **instanceof** Car) ... determines whether the object referred to by myVehicle is in fact a car.

On the other hand, the assignment statement myCar = myVehicle; would be illegal because myVehicle could potentially refer to other types of vehicles that are not cars. This is similar to a problem we saw previously: The computer will not allow you to assign an **int** value to a variable of type **short**, because not every **int** is a **short**. Similarly, it will not allow you to assign a value of type Vehicle to a variable of type Car because not every vehicle is a car. As in the case of **int** s and **short**s, the solution here is to use type-casting. If, for some reason, you happen to know that myVehicle does in fact refer to a Car, you can use the type cast (Car)myVehicle to tell the computer to treat myVehicle as if it were actually of type Car. So, you could say myCar = (Car)myVehicle; and you could even refer to ((Car)myVehicle).numberOfDoors. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

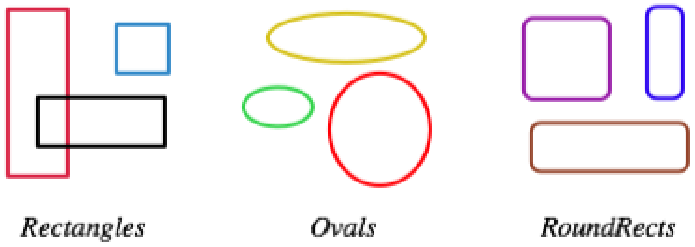
```
System.out.println("Vehicle Data:");
System.out.println("Registration number: "+ myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
   Truck t;
   t = (Truck)myVehicle;
   System.out.println("Number of axels:  " + t.numberOfAxels);
}
```

```
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle:  Motorcycle");
   Motorcycle m;
   m = (Motorcycle)myVehicle;
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if myVehicle refers to an object of type Truck, then the type cast (Car)myVehicle would be an error. When this happes, an exception of type ClassCastException is thrown.

4.4 Polymorphism

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A "roundrect" is just a rectangle with rounded corners.)



Three classes, Rectangle, Oval, and RoundRect, could be used to represent the three types of shapes. These three classes would have a common superclass, Shape, to represent features that all three shapes have in common. The Shape class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the color, position, and size. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {

    Color color;    // Color of the shape.  (Recall that class Color
                    // is defined in package java.awt.  Assume
                    // that this class has been imported.)

    void setColor(Color newColor) {
          // Method to change the color of the shape.
       color = newColor; // change value of instance variable
       redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
          // method for drawing the shape
       ? ? ?  // what commands should go here?
    }

    . . .           // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method redraw(). The problem is that each different type of shape is drawn differently. The method setColor() can be called for any type of shape. How does the computer know which shape to draw when it executes the redraw() ? Informally, we can answer the question like this: The computer executes redraw() by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

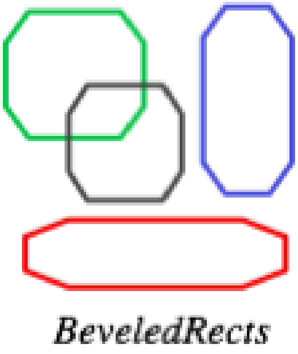
In practice, this means that each of the specific shape classes has its own redraw() method:

```
class Rectangle extends Shape {
   void redraw() {
      . . .  // commands for drawing a rectangle
   }
   . . . // possibly, more methods and variables
}
class Oval extends Shape {
   void redraw() {
      . . .  // commands for drawing an oval
   }
   . . . // possibly, more methods and variables
}
class RoundRect extends Shape {
   void redraw() {
      . . .  // commands for drawing a rounded rectangle
   }
   . . . // possibly, more methods and variables
}
```

If oneShape is a variable of type Shape, it could refer to an object of any of the types, Rectangle, Oval, or RoundRect. As a program executes, and the value of oneShape changes, it could even refer to objects of different types at different times! Whenever the statement oneShape.redraw(); is executed, the redraw method that is actually called is the one appropriate for the type of object to which oneShape actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that oneShape happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of oneShape changes as the loop is executed, it is possible that the very same statement "oneShape.redraw();" will call different methods and draw different shapes as it is executed over and over. We say that the redraw() method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object. The object responds to the message by executing the appropriate method. The statement "oneShape.redraw();" is a message to the object referred to by oneShape. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "oneShape.redraw();" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages, and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. Suppose that I decide to add beveled rectangles to the types of shapes my program can deal with. A beveled rectangle has a triangle cut off each corner:



To implement beveled rectangles, I can write a new subclass, BeveledRect, of class Shape and give it its own redraw() method. Automatically, code that I wrote previously – such as the statement oneShape.redraw() – can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

In the statement "oneShape.redraw();", the redraw message is sent to the object oneShape. Look back at the method from the Shape class for changing the color of a shape:

```
void setColor(Color newColor) {
   color = newColor; // change value of instance variable
   redraw(); // redraw shape, which will appear in new color
}
```

A redraw message is sent here, but which object is it sent to? Well, the setColor method is itself a message that was sent to some object. The answer is that the redraw message is sent to that same object, the one that received the setColor message. If that object is a rectangle, then it is the redraw() method from the Rectangle class that is executed. If the object is an oval, then it is the redraw() method from the Oval class. This is what you should expect, but it means that the redraw(); statement in the setColor() method does **not** necessarily call the redraw() method in the Shape class! The redraw() method that is executed could be in any subclass of Shape.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a Rectangle object is created, it contains a redraw() method. The source code for that method is in the Rectangle class. The object also contains a setColor() method. Since the Rectangle class does not define a setColor() method, the **source code** for the rectangle's setColor() method comes from the superclass, Shape, but the **method itself** is in the object of type Rectangle. Even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's setColor() method is executed and calls redraw(), the redraw() method that is executed is the one in the same object.

4.5 Abstract Classes

Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw() method in the appropriate class that is executed. This leaves open the question, What does the redraw() method in the Shape class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a redraw() method in the Shape class? Well, it has to be there, or it would be illegal to call it in the setColor() method of the Shape class, and it would be illegal to write "oneShape.redraw() ;", where oneShape is a variable of type Shape. The compiler would complain that oneShape is a variable of type Shape and there's no redraw() method in the Shape class.

Nevertheless the version of redraw() in the Shape class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type Shape ! You can have **variables** of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape. We say that Shape is an **abstract class**. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the redraw() method in class Shape is an **abstract** method, since it is never meant to be called. In fact, there is nothing for it to do – any actual redrawing is done by redraw() methods in the subclasses of Shape. The redraw() method in Shape has to be there. But it is there only to tell the computer that all Shapes understand the redraw message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape. There is no reason for the abstract redraw() in class Shape to contain any code at all.

Shape and its redraw() method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "**abstract**" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here's what the Shape class would look like as an abstract class:

```
public abstract class Shape {

     Color color;   // color of shape.

     void setColor(Color newColor) { //method to change the color of the shape
        color = newColor; // change value of instance variable
        redraw(); // redraw shape, which will appear in new color
     }

     abstract void redraw();
     // abstract method—must be defined in concrete subclasses

     . . .   // more instance variables and methods
} // end of class Shape
```

Once you have declared the class to be **abstract**, it becomes illegal to try to create actual objects of type Shape, and the computer will report a syntax error if you try to do so.

Recall that a class that is not explicitly declared to be a subclass of some other class is automatically made a subclass of the standard class Object. That is, a class declaration with no "**extends**" part such as **public class** myClass { . . . is exactly equivalent to **public class** myClass **extends** Object { . . ..

This means that class Object is at the top of a huge class hierarchy that includes every other class. (Semantially, Object is an abstract class, in fact the most abstract class of all. Curiously, however, it is not declared to be **abstract** syntactially, which means that you can create objects of type Object. What you would do with them, however, I have no idea.)

Since every class is a subclass of Object, a variable of type Object can refer to any object whatsoever, of any type. Java has several standard data structures that are designed to hold Object s, but since every object is an instance of class Object, these data structures can actually hold any object whatsoever. One example is the "ArrayList" data structure, which is defined by the class ArrayList in the package java.util. An ArrayList is simply a list of Object s. This class is very convenient, because an ArrayList can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type Object, the list can actually hold objects of any type.

A program that wants to keep track of various Shape s that have been drawn on the screen can store those shapes in an ArrayList. Suppose that the ArrayList is named listOfShapes. A shape, oneShape for example, can be added to the end of the list by calling the instance method "listOfShapes.add(oneShape);" and removed from the list with the instance method "listOfShapes.remove(oneShape);". The number of shapes in the list is given by the method "listOfShapes.size()". It is possible to retrieve the i^{th} object from the list with the call "listOfShapes.get(i)". (Items in the list are numbered from 0 to listOfShapes.size()−1.) However, note that this method returns an Object, not a Shape. (Of course, the people who wrote the ArrayList class didn't even know about Shapes, so the method they wrote could hardly have a return type of Shape!) Since you know that the items in the list are, in fact, Shapes and not just Objects, you can type-cast the Object returned by listOfShapes.get(i) to be a value of type Shape by saying:
oneShape = (Shape)listOfShapes.get(i);.

Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple **for** loop, which is lovely example of object-oriented programming and of polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
    Shape s; // i-th element of the list , considered as a Shape
    s = (Shape)listOfShapes.get(i);
    s.redraw(); //What's drawn here depends on what type of shape s is!
}
```

The sample source code file ShapeDraw.java uses an abstract Shape class and an ArrayList to hold a list of shapes. The file defines an applet in which the user can add various shapes to a drawing area. Once a shape is in the drawing area, the user can use the mouse to drag it around.

You might want to look at this file, even though you won't be able to understand all of it at this time. Even the definitions of the shape classes are somewhat different from those that I have described in this section. (For example, the draw() method has a parameter of type Graphics. This parameter is required because of the way Java handles all drawing.) I'll return to this example in later chapters when you know more about GUI programming. However, it would still be worthwhile to look at the definition of the Shape class and its subclasses in the source code. You might also check how an ArrayList is used to hold the list of shapes.

If you click one of the buttons along the bottom of this applet, a shape will be added to the screen in the upper left corner of the applet. The color of the shape is given by the "pop-up menu" in the lower right. Once a shape is on the screen, you can drag it around with the mouse. A shape will maintain the same front-to-back order with respect to other shapes on the screen, even while you are dragging it. However, you can move a shape out in front of all the other shapes if you hold down the shift key as you click on it.

In the applet the only time when the actual class of a shape is used is when that shape is added to the screen. Once the shape has been created, it is manipulated entirely as an abstract shape. The method that implements dragging, for example, works only with variables of type Shape. As the Shape is being dragged, the dragging method just calls the Shape's draw method each time the shape has to be drawn, so it doesn't have to know how to draw the shape or even what type of shape it is. The object is responsible for drawing itself. If I wanted to add a new type of shape to the program, I would define a new subclass of Shape, add another button to the applet, and program the button to add the correct type of shape to the screen. No other changes in the programming would be necessary.

4.6 this and super

ALTHOUGH THE BASIC IDEAS of object-oriented programming are reasonably simple and clear, they are subtle, and they take time to get used to. And unfortunately, beyond the basic ideas there are a lot of details. This section and the next cover more of those annoying details. You should not necessarily master everything in these two sections the first time through, but you should read it to be aware of what is possible. For the most part, when I need to use this material later in the text, I will explain it again briefly, or I will refer you back to it. In this section, we'll look at two variables, **this** and **super**, that are automatically defined in any instance method.

4.6.1 The Special Variable this

A static member of a class has a simple name, which can only be used inside the class. For use outside the class, it has a full name of the form **class**−name.simple−name. For example, "System.out" is a static member variable with simple name "out" in the class "System". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable of the same name.

Instance variables and instance methods also have simple names. The simple name of such an instance member can be used in instance methods in the class where the instance member is defined. Instance members also have full names, but remember that instance variables and methods are actually contained in objects, not classes. The full name of an instance member has to contain a reference to the object that contains the instance member. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form variable−name.simple−name. But suppose you are writing the definition of an instance method in some class. How can you get a reference to the object that contains that instance method? You might need such a reference, for example, if you want to use the full name of an instance variable, because the simple name of the instance variable is hidden by a local variable or parameter.

Java provides a special, predefined variable named "**this** " that you can use for such purposes. The variable, **this**, is used in the source code of an instance method to refer to the object that contains the method. This intent of the name, **this**, is to refer to "this object," the one right here that this very method is in. If x is an instance variable in the same object, then **this**.x can be used as a full name for that variable. If otherMethod() is an instance method in the same object, then **this**.otherMethod() could be used to call that method. Whenever the computer executes an instance method, it automatically sets the variable, **this**, to refer to the object that contains the method.

One common use of **this**is in constructors. For example:

```
public class Student {

    private String name;  // Name of the student.

    public Student(String name) {
         // Constructor.  Create a student with specified name.
       this.name = name;
    }      .
      .    // More variables and methods.
      .
}
```

In the constructor, the instance variable called name is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, **this**.name. In the assignment statement, the value of the formal parameter, name, is assigned to the instance variable, **this**.name. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There are other uses for **this**. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a method, as an actual parameter. In that case, you can use **this** as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say "System.out.println(**this**);". Or you could assign the value of **this** to another variable in an assignment statement. In fact, you can do anything with **this** that you could do with any other variable, except change its value.

4.6.2 The Special Variable super

Java also defines another special variable, named "**super**", for use in the definitions of instance methods. The variable **super** is for use in a subclass. Like **this**, **super** refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. **super** doesn't know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Let's say that the class you are writing contains an instance method doSomething(). Consider the method call statement **super**.doSomething(). Now, **super** doesn't know anything about the doSomething() method in the subclass. It only knows about things in the superclass, so it tries to execute a method named doSomething() from the superclass. If there is none – if the doSomething() method was an addition rather than a modification – you'll get a syntax error.

The reason **super** exists is so you can get access to things in the superclass that are **hidden** by things in the subclass. For example, **super**.x always refers to an instance variable named x in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not **replace** the variable of the same name in the superclass; it merely hides it. The variable from the superclass can still be accessed, using **super**.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass overrides the method from the superclass. Again, however, **super** can be used to access the method from the superclass.

The major use of **super** is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use **super** to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a PairOfDice class that includes a roll() method. Suppose that you want a subclass, GraphicalDice, to represent a pair of dice drawn on the computer screen. The roll() method in the GraphicalDice class should do everything that the roll() method in the PairOfDice class does. We can express this with a call to **super**.roll(), which calls the method in the superclass. But in addition to that, the roll() method for a GraphicalDice object has to redraw the dice to show the new values. The GraphicalDice class might look something like this:

```
public class GraphicalDice extends PairOfDice {

    public void roll() {
            // Roll the dice, and redraw them.
         super.roll();  // Call the roll method from PairOfDice.
         redraw();      // Call a method to draw the dice.
    }       .
       .  // More stuff, including definition of redraw().
       .
}
```

Note that this allows you to extend the behavior of the roll() method even if you don't know how the method is implemented in the superclass!

4.6.3 Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do not become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you.

This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't know how it works, or if the constructor in the superclass initializes **private** member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, **super**. As the very first statement in a constructor, you can use **super** to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling **super** as a method (even though **super** is not a method and you can't call constructors the same way you call other methods anyway). As an example, assume that the PairOfDice class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

     public GraphicalDice() {  // Constructor for this class.

          super(3,4);  // Call the constructor from the
                       //   PairOfDice class, with parameters 3, 4.

          initializeGraphics();  // Do some initialization specific
                                 //   to the GraphicalDice class.
     }         .
          .  // More constructors, methods, variables...
          .
}
```

The statement "**super**(3,4);" calls the constructor from the superclass. This call must be the first line of the constructor in the subclass. Note that if you don't explicitly call a constructor from the superclass in this way, then the default constructor from the superclass, the one with no parameters, will be called automatically.

This might seem rather technical, but unfortunately it is sometimes necessary. By the way, you can use the special variable **this** in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

Chapter 5

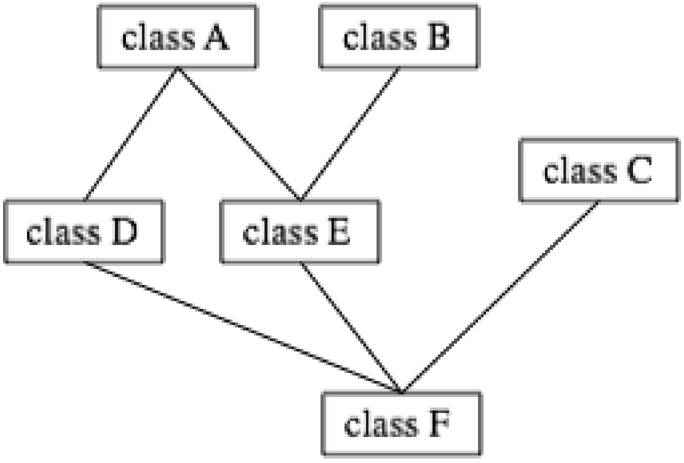
Interfaces, Nested Classes, and Other Details

Contents

THIS SECTION simply pulls together a few more miscellaneous features of object oriented programming in Java. Read it now, or just look through it and refer back to it later when you need this material. (You will need to know about the first topic, interfaces, almost as soon as we begin GUI programming.)

5.1 Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called `multiple inheritance` . In the illustration below, for example, classE is shown as having both classA and classB as direct superclasses, while classF has three direct superclasses.



Multiple inheritance (NOT allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: `interfaces`.

We've encountered the term "interface" before, in connection with black boxes in general and methods in particular. The interface of a method consists of the name of the method, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the method. A method also has an implementation: the block of code which defines it and which is executed when the method is called.

In Java, **interface** is a reserved word with an additional, technical meaning. An "**interface**" in this sense consists of a set of instance method interfaces, without any associated implementations. (Actually, a Java interface can contain other things as well, but we won't discuss them here.) A class can `implement` an **interface** by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java **interface**:

```
public interface Drawable {
   public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the `draw()` method is omitted. A class that implements the **interface** `Drawable` must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
public class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something —— presumably, draw a line
    }   . . . // other methods and variables
}
```

Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also **state** that it implements the interface, using the reserved word **implements** as in this example: "**public class** Line **implements** Drawable". Any class that implements the Drawable interface defines a draw() instance method. Any object created from such a class includes a draw() method. We say that an **object** implements an **interface** if it belongs to a class that implements the interface. For example, any object of type Line implements the Drawable interface.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend one other class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
   . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The methods in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. And as with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if Drawable is an interface, and if Line and FilledCircle are classes that implement Drawable, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It can
                  //    refer to any object that implements the
                  //    Drawable interface.

figure = new Line();  // figure now refers to an object of class Line
figure.draw(g);   // calls draw() method from class Line

figure = new FilledCircle();   // Now, figure refers to an object
                               //   of class FilledCircle.
figure.draw(g);   // calls draw() method from class FilledCircle
```

A variable of type Drawable can refer to any object of any class that implements the Drawable interface. A statement like figure.draw(g), above, is legal because figure is of type Drawable, and **any** Drawable object has a draw() method. So, whatever object figure refers to, that object must have a draw() method.

Note that a type is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a method, or the return type of a method. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are a few interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters.

5.2 Nested Classes

A class seems like it should be a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a nested **class** is any class whose definition is inside the definition of another class. Nested classes can be either named or anonymous. I will come back to the topic of anonymous classes later in this section. A named nested class, like most other things that occur in classes, can be either static or non-static.

The definition of a static nested looks just like the definition of any other class, except that it is nested inside another class and it has the modifier **static** as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named WireFrameModel represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the WireFrameModel class contains a static nested class, Line, that represents a single line. Then, outside of the class WireFrameModel, the Line class would be referred to as WireFrameModel.Line. Of course, this just follows the normal naming convention for static members of a class. The definition of the WireFrameModel class with its nested Line class would look, in outline, like this:

```
public class WireFrameModel {

   . . . // other members of the WireFrameModel class

   static public class Line {
         // Represents a line from the point (x1,y1,z1)
         // to the point (x2,y2,z2) in 3-dimensional space.
      double x1, y1, z1;
      double x2, y2, z2;
   } // end class Line

   . . . // other members of the WireFrameModel class

} // end WireFrameModel
```

Inside the WireFrameModel class, a Line object would be created with the constructor "**new** Line()". Outside the class, "**new** WireFrameModel.Line()" would be used.

A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes.

When you compile the above class definition, two class files will be created. Even though the definition of Line is nested inside WireFrameModel, the compiled Line class is stored in a separate file. The full name of the class file for the Line class will be WireFrameModel$Line.**class**.

Non-static nested classes are referred to as inner classes. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true – at least logically – for inner classes. It's as if each object that belongs to the containing class has its **own copy** of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared **private**. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method, make it non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form variableName. NestedClassName, where variableName is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object "**this**" is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the PokerGame class could be:

```
public class PokerGame {  // Represents a game of poker.

    private class Player {  //Represents one of the players in this game.
       ...

    } // end class Player

    private Deck deck;       // A deck of cards for playing the game.
    private int pot;         // The amount of money that has been bet.
    ...
} // end class PokerGame
```

If game is a variable of type PokerGame, then, conceptually, game contains its own copy of the Player class. In an an instance method of a PokerGame object, a new Player object would be created by saying "**new** Player()", just as for any other class. (A Player object could be created outside the PokerGame class with an expression such as "game.newÂăPlayer()". Again, however, this is very rare.) The Player object will have access to the deck and pot instance variables in the PokerGame object. Each PokerGame object has its own deck and pot and Players. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the Player class non-static. This is the most natural way for players to behave. A Player object represents a player of one particular poker game. If Player were a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

5.2.1 Anonymous Inner Classes

In some cases, you might find yourself writing an inner class and then using that class in just a single line of your program. Is it worth creating such a class? Indeed, it can be, but for cases like this you have the option of using an anonymous inner **class**. An anonymous class is created with a variation of the **new**operator that has the form

```
new  superclass−or−interface( parameter−list) {
     methods−and−variables
}
```

This constructor defines a new class, without giving it a name, and it simultaneously creates an object that belongs to that class. This form of the **new**operator can be used in any statement where a regular "**new**" could be used. The intention of this expression is to create: "a new object belonging to a class that is the same as superclass-or-interface but with these methods-and-variables added." The effect is to create a uniquely customized object, just at the point in the program where you need it. Note that it is possible to base an anonymous class on an interface, rather than a class. In this case, the anonymous class must implement the interface by defining all the methods that are declared in the interface. If an interface is used as a base, the parameter−list is empty. Otherwise, it contains parameters for a constructor in the superclass.

Anonymous classes are often used for handling events in graphical user interfaces, and we will encounter them several times in the chapters on GUI programming. For now, we will look at one not-very-plausible example. Consider the Drawable interface, which is defined earlier in this section. Suppose that we want a Drawable object that draws a filled, red, 100-pixel square. Rather than defining a new, separate class and then using that class to create the object, we can use an anonymous class to create the object in one statement:

```
Drawable redSquare = new Drawable() {
       void draw(Graphics g) {
          g.setColor(Color.red);
          g.fillRect(10,10,100,100);
       }
};
```

The semicolon at the end of this statement is not part of the class definition. It's the semicolon that is required at the end of every declaration statement.

When a Java class is compiled, each anonymous nested class will produce a separate class file. If the name of the main class is MainClass, for example, then the names of the class files for the anonymous nested classes will be MainClass$1.**class**, MainClass$2.**class**, MainClass$3.**class**, and so on.

5.3 Mixing Static and Non-static

Classes, as I've said, have two very distinct purposes. A class can be used to group together a set of static member variables and static member methods. Or it can be used as a factory for making objects. The non-static variables and methods in the class definition specify the instance variables and methods of the objects. In most cases, a class performs one or the other of these roles, not both.

Sometimes, however, static and non-static members are mixed in a single class. In this case, the class plays a dual role. Sometimes, these roles are completely separate. It is also possible for the static and non-static parts of a class to interact. This happens when instance methods use static member variables or call static member methods. An instance method belongs to an object, not to the class itself, and there can be many objects with their own versions of the instance method. But there is only one copy of a static member variable. So, effectively, we have many objects sharing that one variable.

Suppose, for example, that we want to write a PairOfDice class that uses the Random class for rolling the dice. To do this, a PairOfDice object needs access to an object of type Random. But there is no need for each PairOfDice object to have a separate Randomobject. (In fact, it would not even be a good idea: Because of the way ran dom number generators work, a program should, in general, use only one source of random numbers.) A nice solution is to have a single Random variable as a **static** member of the PairOfDice class, so that it can be shared by all PairOfDice objects. For example:

```
import java.util.Random;

 public class PairOfDice {

    \code{private static Random randGen = new Random();}


    public int die1;   // Number showing on the first die.
    public int die2;   // Number showing on the second die.

    public PairOfDice() {
           // Constructor.  Creates a pair of dice that
           // initially shows random values.
         roll();
    }
    public void roll() {
           // Roll the dice by setting each of the dice to be
           // a random number between 1 and 6.
         die1 = randGen.nextInt(6) + 1;
         die2 = randGen.nextInt(6) + 1;
    }
} // end class PairOfDice
```

As another example, let's rewrite the Student class. I've added an ID for each student and a **static** member called nextUniqueID . Although there is an ID variable in each student object, there is only one nextUniqueID variable.

```
public class Student {

   private String name;  // Student's name.
   private int ID;  // Unique ID number for this student.
   public double test1, test2, test3;   // Grades on three tests.

   private static int nextUniqueID = 0;
            // keep track of next available unique ID number

   Student(String theName) {
        // Constructor for Student objects; provides a name for the Student,
        // and assigns the student a unique ID number.
      name = theName;
      nextUniqueID++;
      ID = nextUniqueID;
   }
   public String getName() {
        // Accessor method for reading value of private
        // instance variable, name.
      return name;
   }
   public int getID() {
        // Accessor method for reading value of ID.
      return ID;
   }
   public double getAverage() {
        // Compute average test grade.
      return (test1 + test2 + test3) / 3;
   }
}  // end of class Student
```

The initialization "nextUniqueID = 0" is done once, when the class is first loaded. Whenever a Student object is constructed and the constructor says "nextUniqueID++;", it's always the same static member variable that is being incremented. When the very first Student object is created, nextUniqueID becomes 1. When the second object is created, nextUniqueID becomes 2. After the third object, it becomes 3. And so on. The constructor stores the new value of nextUniqueID in the ID variable of the object that is being created. Of course, ID is an instance variable, so every object has its own individual ID variable. The class is constructed so that each student will automatically get a different value for its IDvariable. Furthermore, the ID variable is **private**, so there is no way for this variable to be tampered with after the object has been created. You are guaranteed, just by the way the class is designed, that every student object will have its own permanent, unique identification number. Which is kind of cool if you think about it.

5.3.1 Static Import

The **import** directive makes it possible to refer to a class such as java.awt.Color using its simple name, Color. All you have to do is say **import** java.awt.Color or **import** java.awt.∗. Uou still have to use compound names to refer to static member variables such as System.out and to static methods such as Math.sqrt.

Java 5.0 introduced a new form of the **import** directive that can be used to import **static** members of a class in the same way that the ordinary **import** directive imports classes from a package. The new form of the directive is called a **static import**, and it has syntax

```
import static package−name class−name static−member−name;
```

to import one static member name from a class, or

```
import static package−name class−name.∗;
```

to import all the public static members from a class. For example, if you preface a class definition with

```
import static java.lang.System.out;
```

then you can use the simple name out instead of the compound name System.out. This means you can use out.println instead of System.out.println. If you are going to work extensively with the Mathclass, you can preface your class definition with

```
import static java.lang.Math.∗;
```

This would allow to say sqrtinstead of Math.sqrt, log instead of Math.log, PI instead of Math.PI, and so on.

Note that the static import directive requires a **package**−name, even for classes in the standard package java.lang. One consequence of this is that you can't do a static import from a class in the default package.

5.4 Enums as Classes

Enumerated types are actually classes, and each enumerated type constant is a pubic, **final**, **static** member variable in that class (even though they are not declared with these modifiers). The value of the variable is an object belonging to the enumerated type class. There is one such object for each enumerated type constant, and these are the only objects of the class that can ever be created. It is really these objects that represent the possible values of the enumerated types. The enumerated type constants are actually variables that refer to these objects.

When an enumerated type is defined inside another class, it is a nested class inside the enclosing class. In fact, it is a static nested class, whether you declare it to be **static** or not. But it can also be declared as a non-nested class, in a file of its own. For example, we could define the following enumerated type in a file named Suit.java:

```
public enum Suit {

   SPADES, HEARTS, DIAMONDS, CLUBS

}
```

This enumerated type represents the four possible suits for a playing card, and it could have been used in the example Card.java.

Furthermore, in addition to its list of values, an enumerated type can contain some of the other things that a regular class can contain, including methods and additional member variables. Just add a semicolon (;) at the end of the list of values, and then add definitions of the methods and variables in the usual way. For example, we might make an enumerated type to represent the possible values of a playing card. It might be useful to have a method that returns the corresponding value in the game of Blackjack. As another example, suppose that when we print out one of the values, we'd like to see something different from the default string representation (the identifier that names the constant). In that case, we can override the toString() method in the class to print out a different string representation. This would gives something like:

```
public enum CardValue {

   ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
         NINE, TEN, JACK, QUEEN, KING;

   /**
    * Return the value of this CardValue in the game of Blackjack.
    * Note that the value returned for an ace is 1.
    */
   public int blackJackValue() {
      if (this == JACK || this == QUEEN || this == KING)
         return 10;
      else
         return 1 + ordinal();
   }
   /**
    * Return a String representation of this CardValue, using numbers
    * for the numerical cards and names for the ace and face cards.
    */
   public String toString() {
      switch (this) {        // "this" is one of the enumerated type values
      case ACE:              // ordinal number of ACE
         return "Ace";
      case JACK:             // ordinal number of JACK
         return "Jack";
      case QUEEN:             // ordinal number of QUEEN
         return "Queen";
      case KING:             // ordinal number of KING
         return "King";
      default:             // it's a numeric card value
         int numericValue = 1 + ordinal();
      return "" + numericValue;
   }

} // end CardValue
```

blackjackValue() and toString() are instance methods in CardValue. Since CardValue.JACK is an object belonging to the class CardValue, you can, off-course, call CardValue.JACK.blackjackValue(). Suppose that cardVal is declared to be a variable of type CardValue, so that it can refer to any of the values in the enumerated type. We can call cardVal.blackjackValue() to find the Blackjack value of the CardValue object to which cardVal refers, and System.out.println(cardVal) will implicitly call the method cardVal.toString() to obtain the print representation of that CardValue. (One other thing to keep in mind is that since CardValue is a class, the value of cardVal can be **null**, which means it does not refer to any object.)

Remember that ACE, TWO, ..., KING are the only possible objects of type CardValue, so in an instance methods in that class, **this** will refer to one of those values. Recall that the instance method ordinal() is defined in any enumerated type and gives the position of the enumerated type value in the list of possible values, with counting starting from zero.

(If you find it annoying to use the class name as part of the name of every enumerated type constant, you can use static import to make the simple names of the constants directly available – but only if you put the enumerated type into a package. For example, if the enumerated type CardValue is defined in a package named cardgames, then you could place

```
import static cardgames.CardValue.*;
```

at the beginning of a source code file. This would allow you, for example, to use the name JACK in that file instead of CardValue.JACK.)