UNIT 1

Need for oop paradigm

- The object oriented paradigm is a methodology for producing reusable software components
- The object-oriented paradigm is a programming methodology that promotes the efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.
- Object oriented programming has taken a completely different direction and will place an emphasis on object s and information. With object oriented programming, a problem will be broken down into a number of units .these are called objects .The foundation of oop is the fact that it will place an emphasis on objects and classes. There are number of advantages to be found with using the oop paradigm, and some of these are oop paradigm
- Object oriented programming is a concept that was created because of the need to overcome the problems that were found with using structured programming techniques. While structured programming uses an approach which is top down, oop uses an approach which is bottom up.
- A paradigm is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: procedural, object-oriented, functional and declarative • A paradigm shift from a function-centric approach to an object-centric approach to software development
- A program in a procedural paradigm is an active agent that uses passive objects that we refer to as data or data items.
- The basic unit of code is the class which is a template for creating run-time objects.
- Classes can be composed from other classes. For example, Clocks can be constructed as an aggregate of Counters.
- The object-oriented paradigm deals with active objects instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher and so on. The actions to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions.
- A file in an object-oriented paradigm can be packed with all the procedures—called methods in the object-oriented paradigm—to be performed by the file: printing, copying, Deleting and so on. The program in this paradigm just sends the corresponding request to the object.

- Java provides automatic garbage collection, relieving the programmer of the need to ensure that unreferenced memory is regularly deallocated.
-

Object Oriented Paradigm – Key Features

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation:

- Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
- Access to the code and data inside the wrapper is tightly controlled through a well defined interface.
- To relate this to the real world, consider the automatic transmission on an automobile.
- It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift.
- The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

Abstraction:

- Abstraction in Java or Object oriented programming is a way to segregate implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.
- An essential component of object oriented programming is Abstraction Humans manage complexity through abstraction.
- For example people do not think a car as a set of tens and thousands of individual parts. They think of it as a well defined object with its own unique behavior. This abstraction

allows people to use a car ignoring all details of how the engine, transmission and braking systems work.

- In computer programs the data from a traditional process oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects. Thus each object describes its own behavior.

Inheritance

- The derivation of one class from another so that the attributes and methods of one class are part of the definition of another class.
- The first class is often referred to the base or parent class.
- The child is often referred to as a derived or sub-class.
- Derived classes are always a kind of‘their base classes.
- Derived classes generally add to the attributes and/or behavior of the base class. Inheritance is one form of object-oriented code reuse.
- E.g. Both Motorbikes and Cars are kinds of MotorVehicles and therefore share some common attributes and behaviour but may add their own that are unique to that particular type.
- Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. Different kinds of objects often have a certain amount in common with each other.
- In the Java programming language, each class is allowed to have one direct superclass,and each superclass has the potential for an unlimited number of subclasses:
- Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).
- Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.
- In this example, Bicycle now becomes the super class of Mountain Bike, Road Bike, and Tandem Bike.

Polymorphism:

- Polymorphism (from the Greek, meaning ―many forms‖) is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). We might have a program that requires three types of stacks.
- One stack is used for integer values, one for floating-point values, and one for characters.
- The algorithm that implements each stack is the same, even though the data being stored differs.
- In Java we can specify a general set of stack routines that all share the same names.
- More generally, the concept of polymorphism is often expressed by the phrase ―one interface, multiple methods.
- This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action.
- Polymorphism allows us to create clean, sensible, readable, and resilient code.

Method Binding:

- Binding denotes association of a name with a class.
- Static binding is a binding in which the class association is made during compile time. This is also called as early binding.
- Dynamic binding is a binding in which the class association is not made until the object is created at execution time. It is also called as late binding.

THE CREATION OF JAVA:

- Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as Green Team), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.
- For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

James Gosling

- Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describe the history of java.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt.

4) After that, it was called Oak and was developed as a part of the Green project.

Why sun choosed "Oak" name?

5) Why Oak? Oak is a symbol of strength and choosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.

6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

Why sun choosed "Java" name?

7) Why they choosed java name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name not an acronym.

10) **Originally developed by James Gosling at Sun Microsystems and released in 1995.**

11) In 1995, Time magazine called Java one of the Ten Best Products of 1995.

12) JDK 1.0 released in (January 23, 1996).

Introducing Classes

- **Classes are made up of objects**. This is easy to understand if you are familiar with object-oriented programming. Think of an apartment building, for example.
- It is made up of apartments. Each apartment probably has doors, windows, a kitchen, a bedroom, and a bathroom, among other things. **Each apartment is a *unit.*** Sometimes an apartment building is referred to as a ***five-unit apartment building***. The apartments do not all have to be exactly alike. Some may be on the first floor, some may be in the basement, some may face south, and some may be recently refurbished. One apartment may have two bedrooms and another, just down the hall, only one. Each apartment has its own mail slot with an individual address. Even though the apartments are not all identical, they are all apartments.
- Apartments are the ***objects*** in this example. Even though they have smaller parts and are not identical, conceptually each is a unit. The apartment building is the ***class***. It is made up of objects, or units. These objects are not all exactly alike, but they have enough similar characteristics that they can be ***classed*** together.
- Another term useful in object-oriented programming is ***instance***. Apartment building 3 is an ***instance,*** or actual apartment building. It is real. **An instance is one specific object within the class of objects.**
- Each apartment has more interesting information. One might be empty. In this case, the empty status is the apartment's ***state***. A rental-application program could keep track of available apartments on the basis of this state. A ***method*** would be associated with testing for the state of this apartment.
- Now let's take these ideas to another level of abstraction. The complex is made up of apartment buildings, which are made up of apartment units. The whole complex can be referred to as Countrybrook or something else equally romantic and descriptive. The complex, then, is the conceptual gathering together of the apartment buildings in the area even though each apartment building is slightly different, with different addresses and other characteristics that make them unique.
- The apartment complex is an example of one of the classes in a ***class library***. A class library is a set of classes. Class libraries group classes that perform similar functions but are dissimilar enough to warrant their own classes. Recall that the apartment complex is

one class. Suppose right down the street is a mall. The mall is made up of smaller units-stores. The mall is another class in the class library. It is a building, but it has quite a different structure and function than the apartment building. It could be set up as its own class.

- Java comes with a set of class libraries that handle many tasks, such as input/output, screen painting, and mouse clicks. The class libraries of Java are its heart and strength.

Class Fundamentals

- Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a template for an object, and an object is an instance of a class.
- Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The General Form of a Class

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form.Classes can get much more complex. A simplified general form of a class definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
```

```
type methodnameN(parameter-list) {
// body of method
}  }
```

- The data, or variables, defined within a class are called **instance variables**. The code is contained within methods.
- Collectively, the methods and variables defined within a class are called **members of the class.**
- In most classes, the instance variables are acted upon and accessed by the methods defined for that class.
- Thus, as a general rule, it is the methods that determine how a class' data can be used.
- Variables defined within a class are called **instance variables** because each instance of the class (that is, each object of the class) contains its **own copy of these variables.**
- Thus, the data for **one object is separate and unique from the data for another**.
- All methods have the same general form as main( ), which we have been using thus far.
- However, most methods will not be specified as static or public. Notice that the general form of a class does not specify a main( ) method.
- **Java classes do not need to have a main( ) method**. You only specify one if that class is the starting point for your program.
- Further, some kinds of **Java applications, such as applets**, **don't require** a main( ) method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: **width, height, and depth**. Currently, Box does not contain any methods

```
class Box {
double width;
double height;
double depth;
}
```

- As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box.
- It is important to remember that a class declaration only creates a template;
- it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

- As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
- Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator.
- The dot operator links the name of the object with the name of an instance variable.
- For example, to assign the width variable of mybox the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100.

Note: In general, you use the dot operator to access both the instance variables and the methods within an object.

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
```

```
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

- You should call the file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box.
- When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo.
- The Java compiler automatically puts each class into its own .class file.
- It is not necessary **for both the Box and the BoxDemo class** to actually be in the same source file.
- You could put each class in its own file, called Box.java and BoxDemo.java, respectively.
- To run this program, **you must execute BoxDemo.class.**

When you do, you will see the following output:

```
Volume is 3000.0
```

- As stated earlier, each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height.
- It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.
- For example, the following program declares two Box objects:

```
// This program declares two Box objects.
```

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

Declaring Objects

Obtaining objects of a class is a two-step process.

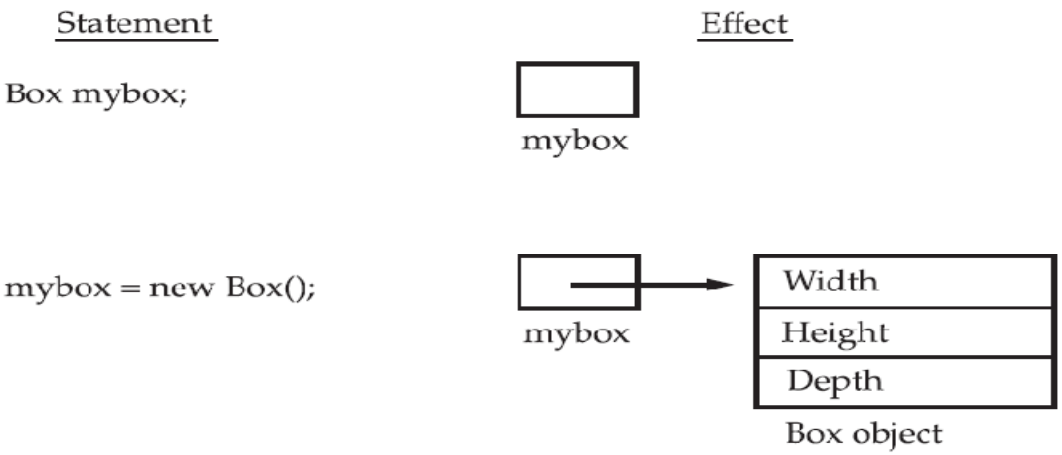
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator.
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.
- In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

```
Box mybox = new Box();
```

- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

- The first line declares mybox as a reference to an object of type Box. At this point, mybox does not yet refer to an actual object.
- The next line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object.
- But in reality, mybox simply holds, in essence, the memory address of the actual Box object. The effect of these two lines of code is depicted in below Figure



Assigning Object Reference Variables

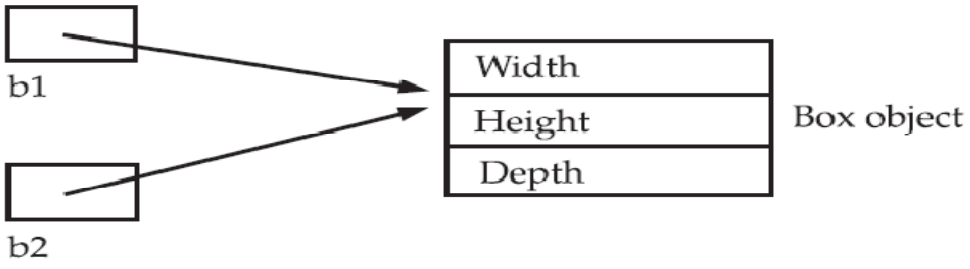
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong.

Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1.

Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.



Although b1 and b2 both refer to the same object, they are not linked in any other way.For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.

Introducing Methods

Classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility.

This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create.

- If the method does not return a value, its return type must be void.
- The name of the method is specified by name. This can be any **legal identifier** other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas.
- Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

  return value;

  Here, value is the value returned.

Adding a Method to the Box Class

- Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class.

- In fact, methods define the interface to most classes. This allows the class implement or to hide the specific layout of internal data structures behind cleaner method abstractions.
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

```
// This program includes a method inside the box class.
    class Box {
    double width;
    double height;
    double depth;
    // display volume of a box
    void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
    }
    }
    class BoxDemo3 {
    public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
    /* assign different values to mybox2's
    instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
    // display volume of first box
    mybox1.volume();
    // display volume of second box
```

```
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

- The first line here invokes the volume( ) method on mybox1. That is, it calls volume( ) relative to the mybox1 object, using the object's name followed by the dot operator.
- Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1, and the call to mybox2.volume( ) displays the volume of the box defined by mybox2.
- Each time volume( ) is invoked, it displays the volume for the specified box.

Returning a Value

While the implementation of volume( ) does move the computation of a box's volume inside the Box class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement volume( ) is to have it compute the volume of the boxand return the result to the caller.

The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
```

```
}
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

- As you can see, when volume( ) is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the value returned by volume( ).
- Thus, after vol = mybox1.volume();executes, the value of mybox1.volume( ) is 3,000 and this value then is stored in vol.
- There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.
- The variable receiving the value returned by a method (such as vol, in this case) must also be compatible with the return type specified for the method.
- One more point: The preceding program can be written a bit more efficiently because there is actually no need for the vol variable.
- The call to volume( ) could have been used in the println( ) statement directly, as shown here:

System.out.println("Volume is" + mybox1.volume());

- In this case, when println( ) is executed, mybox1.volume( ) will be called automatically and its value will be passed to println( ).

Adding a Method That Takes Parameters

It is important to keep the two terms parameter and argument straight. A parameter is a variable defined by a method that receives a value when the method is called. An argument is a value that is passed to a method when it is invoked.

```
// This program uses a parameterized method.
    class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
    return width * height * depth; }
    // sets dimensions of box
    void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
    }
    }
```

```
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, the setDim( ) method is used to set the dimensions of each box. For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter w, 20 is copied into h, and 15 is copied into d.Inside setDim( ) the values of w, h, and d are then assigned to width, height, and depth,respectively.

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like setDim( ), it would be simpler and more concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. **This automatic initialization is performed** through the use of a constructor.
- A constructor **initializes an object immediately upon creation**. It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically **called when the object is created**, before the new operator completes.

- Constructors look a little strange because they **have no return type**, not even void. This is because the implicit return type of a class' constructor is **the class type itself.**

```
/* Here, Box uses a constructor to initialize the  dimensions of a box. */
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
} }
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
```

} }

When this program is run, it generates the following results:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

Parameterized Constructors

- While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions.
- The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful.
- For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.

```
/* Here, Box uses a parameterized constructor to initialize the dimensions of a box. */
    class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
    }
    // compute and return volume
    double volume() {
    return width * height * depth;
```

```
}
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor.For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object. Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15,respectively.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted. To better understand what this refers to, consider the following version of Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts, one of which is explained in the next ssection. redundant in this case, this is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However,when a local variable has the same name as an instance variable, the local variable hides the instance variable.
- This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If they had been, then width, for example, would have referred to the formal parameter, hiding the instance variable width.
- While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.
- For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
```

}

Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example,if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

  The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Java Data Types:

Java Is a Strongly Typed Language

- It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means.
- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types

- Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and both terms will be used in this book.
- These can be put in four groups:
- Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
- Characters This group includes char, which represents symbols in a character set,like letters and numbers.
- Boolean This group includes boolean, which is a special type for representing true/false values.
- You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.
- The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not.
- They are analogous to the simple types found in most other non–object-oriented languages.

- The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.
- The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment.
- However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range.
- For example, an int is always 32 bits,regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture.
- While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Integers

- Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values.
- Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers.
- However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value.
- Java manages the meaning of the high order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.
- The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

- The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127.
- Variables of type byte are especially useful when you're working with a stream of data from a network or file.
- They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the byte keyword.

For example, the following declares two byte variables called b and c:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the leastused Java type.

Here are some examples of short variable declarations:

```
short s;
short t;
```

int

- The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
- Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case.
- The reason is that when byte and short values are used in an expression, they are promoted to int when the expression is evaluated.
- Therefore, int is often the best choice when an integer is needed.

long

- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.

- The range of a long is quite large. This makes it useful when big, whole numbers are needed.
- For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
public static void main(String args[]) {
int lightspeed;
long days;
long seconds;
long distance;
// approximate speed of light in miles per second
lightspeed = 186000;
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

Note: Clearly, the result could not have been held in an int variable.

Floating-Point Types

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two

kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Each of these floating-point types is examined next.

Float

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.For example, float can be useful when representing dollars and cents.

Here are some example float variable declarations:

float hightemp, lowtemp;

Double

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
public static void main(String args[]) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
```

```
}
```

Characters

- In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide.
- This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.
- It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type.
- The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.
- Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.
- Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

```
// char variables behave like integers.
class CharDemo2 {
public static void main(String args[]) {
char ch1;
ch1 = 'X';
System.out.println("ch1 contains " + ch1);
ch1++; // increment ch1
System.out.println("ch1 is now " + ch1);
}
}
```

The output generated by this program is shown here:

```
ch1 contains X
```

```
ch1 is now Y
```

In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence.

Booleans

Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
        class BoolTest {
        public static void main(String args[]) {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
        }
        }
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a boolean value is output by println( ), "true" or "false" is displayed. Second, the value of a boolean variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this:

if(b == true) …

Third, the outcome of a relational operator, such as <, is a boolean value. This is why the expression 10>9 displays the value "true." Further, the extra set of parentheses around 10>9 is necessary because the + operator has a higher precedence than the >.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value ][, identifier [= value ] …];
```

- Here, type is one of Java's atomic types, or the name of a class or interface. The identifier is the name of the variable.
- You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable.
- To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
// d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Type Conversion and Casting

- If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type.
- If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types.
- To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
    - The two types are compatible.
    - The destination type is larger than the source type.
- When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
- For widening conversions, the numeric types, including integer and floating-point types,are compatible with each other.
- However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.
- As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

- Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable?

- This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte.

If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// …
b = (byte) a;
```

- A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components.
- Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
public static void main(String args[]) {
byte b;
int i = 257;
double d = 323.142;
System.out.println("\nConversion of int to byte.");
b = (byte) i;
```

```
System.out.println("i and b " + i + " " + b);
System.out.println("\nConversion of double to int.");
i = (int) d;
System.out.println("d and i " + d + " " + i);
System.out.println("\nConversion of double to byte.");
b = (byte) d;
System.out.println("d and b " + d + " " + b);
}
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions

- In addition to assignments, there is another place where certain type conversions may occur: in expressions.
- To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.
- For example, examine the following expression:
    - byte a = 40;
    - byte b = 50;
    - byte c = 100;
    - int d = a * b / c;

- The result of the intermediate term a * b easily exceeds the range of either of its byte operands.
- To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- This means that the subexpression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.
- As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:
    - byte b = 50;
    - b = b * 2; // Error! Cannot assign an int to a byte!
- The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable.
- However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int.
- Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.
- This is true even if, as in this particular case, the value being assigned would still fit in the target type.
- In cases where you understand the consequences of overflow, you should use an explicit
- cast, such as
    - byte b = 50;
    - b = (byte)(b * 2);
    - which yields the correct value of 100.

The Type Promotion Rules

- Java defines several type promotion rules that apply to expressions. They are as follows: First,all byte, short, and char values are promoted to int, as just described.
- Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float.
- If any of the operands are double, the result is double.
- The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
}
}
```

- Let's look closely at the type promotions that occur in this line from the program:

double result = (f * b) + (i / c) - (d * s);

- In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float.
- Next, in the subexpression i/c, c is promoted to int, and the result is of type int. Then, in d * s, the value of s is promoted to double, and the type of the subexpression is double.
- Finally, these three intermediate values, float, int, and double, are considered.
- The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Arrays

- An array is a group of like-typed variables that are referred to by a common name. Arrays of
- any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

NOTE: If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

One-Dimensional Arrays

- A one-dimensional array is, essentially, a list of like-typed variables.
- To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
type var-name[ ];
```

- Here, type declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array.
- Thus, the element type for the array determines what type of data the array will hold.
- For example, the following declares an array named month_days with the type "array of int": int month_days[];
- Although this declaration establishes the fact that month_days is an array variable, no array actually exists.
- To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.
- You will look more closely at new in a later chapter, but you need to use it now to allocate memory for arrays.
- The general form of new as it applies to one-dimensional arrays appears as follows:

```
array-var = new type [size];
```

- Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array.
- That is, to use new to allocate an array, you must specify the type and number of elements to allocate.
- The elements in the array allocated by new will automatically be initialized to zero (for numeric types), alse (for boolean), or null (for reference types, which are described in a later chapter).
- This example allocates a 12-element array of integers and links them to month_days:

month_days = new int[12];

- After this statement executes, month_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.
- Let's review: Obtaining an array is a two-step process.
- First, you must declare a variable of the desired array type.
- Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable.
- Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don't worry.
- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

For example, this statement assigns the value 28 to the second element of month_days:

month_days[1] = 28;

The next line displays the value stored at index 3:

System.out.println(month_days[3]);

Putting together all the pieces, here is a program that creates an array of the number of days in each month:

```
// Demonstrate a one-dimensional array.
    class Array {
    public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
```

```
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int month_days[] = new int[12];

- This is the way that you will normally see it done in professionally written Java programs.
- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types.
- An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use new. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```

- When you run this program, you see the same output as that generated by the previous version.

- Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array.
- The Java run-time system will check to be sure that all array indexes are in the correct range.
- For example, the run-time system will check the value of each index into month_days to make sure that it is between 0 and 11 inclusive.
- If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.
        class Average {
        public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;

        for(i=0; i<5; i++)
        result = result + nums[i];
        System.out.println("Average is " + result / 5);
        }
        }
```

Multidimensional Arrays

- In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays.
- However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a twodimensional array variable called twoD:

```
int twoD[][] = new int[4][5];
```

- This allocates a 4 by 5 array and assigns it to twoD. Internally, this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in Figure
- The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

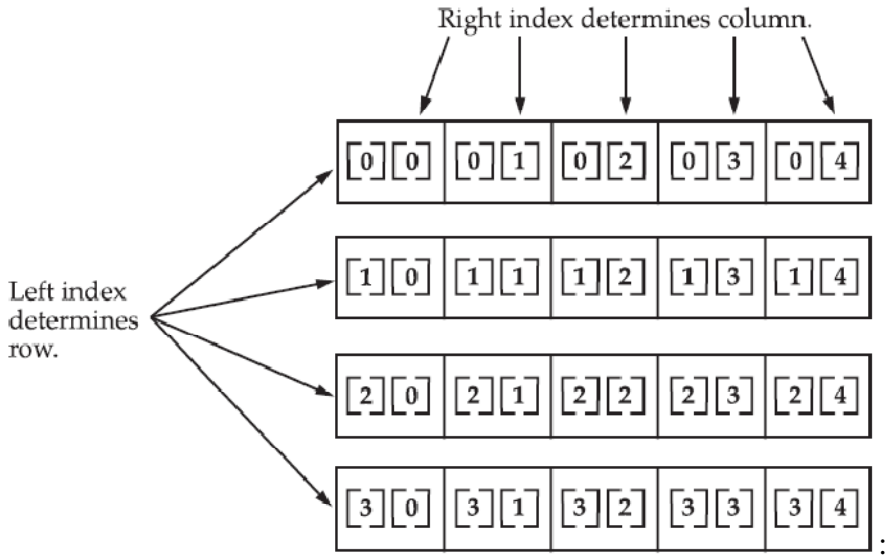
// Demonstrate a two-dimensional array.

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions



Given : int twoD[ ][ ]= new int [4][5];

Figure: A conceptual view of a 4 by 5; two-dimensional array:separately.

For example, this following code allocates memory for the first dimension of twoD when it is declared.

It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others.
- For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.
- As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.
- For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

// Manually allocate differing size second dimensions.

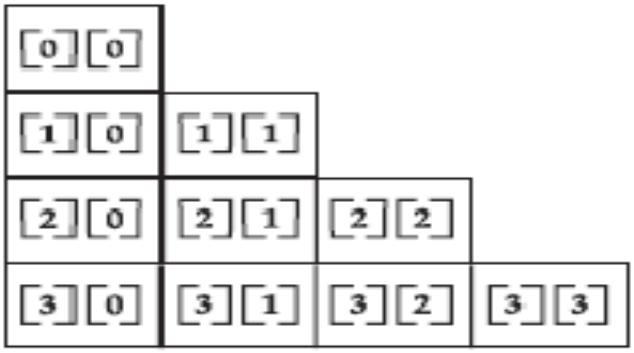
```
class TwoDAgain {
```

```
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
} } }
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



- The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered.
- However, irregular arrays can be used effectively in some situations.
- For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.
- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.
- Also notice that you can use expressions as well as literal values inside of array initializers.
- // Initialize a two-dimensional array.

```
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 } };

int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}
}
}
```

When you run this program, you will get the following output:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
```

```
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

- As you can see, each row in the array is initialized as specified in the initialization lists.Let's look at one more example that uses a multidimensional array.
- The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

// Demonstrate a three-dimensional array.

```
class ThreeDMatrix {
public static void main(String args[]) {
int threeD[][][] = new int[3][4][5];
int i, j, k;
for(i=0; i<3; i++)
for(j=0; j<4; j++)
for(k=0; k<5; k++)
threeD[i][j][k] = i * j * k;
for(i=0; i<3; i++) {
for(j=0; j<4; j++) {
for(k=0; k<5; k++)
System.out.print(threeD[i][j][k] + " ");
System.out.println();
}
System.out.println();
}
}
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

```
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
```

```
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

```
type[ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time.

For example,

int[] nums, nums2, nums3; // create three arrays creates three array variables of type int. It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator Result

+	Addition (also unary plus)
–	Subtraction(also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types. The unary minus operator negates its single operand. The unary plus operator simply returns the value of its operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

```
class BasicMath {
public static void main(String args[]) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
```

```
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;

System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

The Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the %:

```
// Demonstrate the % operator.
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);}}
```

When you run this program, you will get the following output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. Statements like the following are quite common in programming:

a = a + 4; In Java, you can rewrite this statement as shown here: a += 4;

This version uses the += compound assignment operator. Both statements perform the same action: they increase the value of a by 4.

- There are compound assignment operators for all of the arithmetic, binary operators.Thus, any statement of the form **var = var op expression** can be rewritten as **var op= expression** The compound assignment operators provide two benefits.
- First, they save you a bit of typing, because they are "shorthand" for their equivalent long forms. Second, in some cases they are more efficient than are their equivalent long forms.
- For these reasons, you will often see the compound assignment operators used in professionally written Java programs

```
// Demonstrate several assignment operators.
class OpEquals {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator: x++; Similarly, this statement:x = x - 1; is equivalent to x--; These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.

For example: x = 42;y = ++x; In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. x = 42;y = x++;the value of x is obtained before the increment operator is executed, so the value of y is 42.

```
class IncDec {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

The Bitwise Operators

Java defines several bitwise operators that can be applied to the integer types: long, int, short, char, and byte.

These operators act upon the individual bits of their operands. They are summarized in the following table: Since the bitwise operators manipulate the bits within an integer

- All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is 00101010, where each position represents a power of two, starting with 20 at the rightmost bit.
- The next bit position to the left would be 21, or 2, continuing toward the left with 22, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of 21 + 23 + 25, which is 2 + 8 + 32.
- All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones.
- Java uses an encoding known as two's complement, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.
- For example, –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42. To decode a negative number, first invert all of the bits, then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

.

- The reason Java (and most other computer languages) uses two's complement is easy to see when you consider the issue of zero crossing.
- Assuming a byte value, zero is represented by 00000000. In one's complement, simply inverting all of the bits creates 11111111, which creates negative zero.

- The trouble is that negative zero is invalid in integer math. This problem is solved by using two's complement to represent negative values.
- When using two's complement, 1 is added to the complement, producing 100000000. This produces a 1 bit too far to the left to fit back into the byte value, resulting in the desired behavior, where –0 is the same as 0, and 11111111 is the encoding for –1. Although we used a byte value in the preceding example, the same basic principle applies to Java's entire integer types.
- Because Java uses two's complement to store negative numbers—and because all integers are signed values in Java—applying the bitwise operators can easily produce unexpected results.
- For example, turning on the high-order bit will cause the resulting value to be interpreted as a negative number, whether this is what you intended or not.
- To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its Operand. For example, the number 42, which has the following bit pattern?

00101010
Becomes
11010101

after the NOT operator is applied.

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
 00101010  42
&00001111  15
----------
 00001010  10
```

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
  00101010  42
| 00001111  15
  ________
  00101111  47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
  00101010  42
^ 00001111  15
  ________
  00100101  37
```

```
class BitLogic {
public static void main(String args[]) {
String binary[] = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b)|(a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

Output:

a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 110

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form: value << num

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num.

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("i and b: " + i + " " + b);
}
}
```

The output generated by this program is shown here:

Original value of a: 64
i and b: 256 0

Since a is promoted to int for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in i containing the value 256 (1 0000 0000). However, the value in b contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

The Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here: value >> num
- Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num
- The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

  ```
  int a = 32;
  a = a >> 2; // a now contains 8
  ```

```
    class Bits1{
      public static void main(String args[]){
        System.out.println(" >> opeartor");
// shift all the bits in 20 (in binary form) to the right by 2
    System.out.println("20>>2 = "+20>>2);
   }}
```

Output: 20>>2 = 5

The Unsigned Right Shift

Generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java's unsigned, shift right operator, >>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>. Here, a is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int
class Bits3{
  public static void main(String args[]){
    System.out.println(" >>> opeartor");


// shift all the bits in -1 (in binary form) to the right by 2
// without sign extension using >>>

int y = -1>>>2;
System.out.println("-1>>>2 = "+y);

// -1>>>2 in binary form
System.out.println("-1>>>2 in binary form is "+Integer.toBinaryString(y));

   }
}
```

when you run the program above you obtain:

```
-1>>>2 = 1073741823
-1>>>2 in binary form is 11111111111111111111111111111111
```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in a right by four bits, are equivalent:

```
a = a >> 4;
a >>= 4;
```

Likewise, the following two statements, which result in a being assigned the bitwise expression a OR b, are equivalent:

```
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

```
class OpBitEquals {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

Relational Operators

The relational operators determine the relationship that one operand has to the other.Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a Boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=. Notice that in Java equality is denoted with two equal signs, not one.

For example:
the following code fragment is perfectly valid:
int a = 4;
int b = 1;
boolean c = a < b;
In this case, the result of a<b (which is false) is stored in c.

Boolean Logical Operators

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state:

!true == false and !false == true. The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

```
class BoolLogic {
public static void main(String args[]) {
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
```

```
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
}
```

the following output, the string representation of a Java boolean value is one of the literal values true or false:

```
a= true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

The Assignment Operator

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of var must be compatible with the type of expression.The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

```
expression1 ? expression2 : expression3
```

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same (or compatible) type, which can't be void.

Here is an example of the way that the ? is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark. If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression. If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to ratio.

```
class Ternary {
public static void main(String args[]) {
int i, k;
i = 10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
i = -10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
}
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest.Operators in the same row is equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the [ ], ( ), and. can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator (->). It was added by JDK 8 and is used in lambda expressions.

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
–>						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

a >> b + 3

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

a >> (b + 3)

However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:

(a >> b) + 3

Control Statements

- A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program.
- Java's program control statements can be put into the following categories: selection, iteration, and jump.
- Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

- Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- Jump statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Java's Selection Statements

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

If

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition)
statement1;
else
statement2;
```

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

For example, consider the following:

```
int a, b;
//...
if(a < b) a = 0;
else   b =0 ;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

- It is possible to control the if using a single Boolean variable, as shown in this code fragment:

```
boolean dataAvailable;
//...
if (dataAvailable)
```

```
ProcessData();
else
waitForMoreData();
```

- It is possible to control the if using a single Boolean variable, as shown in this code fragment: Here, both statements within the if block will execute if bytesAvailable is greater than zero.

```
boolean dataAvailable;
//...
if (dataAvailable)
ProcessData();
else
waitForMoreData();
```

Remember, only one statement can appear directly after the if or the else. If you want to include more statements, you'll need to create a block, as in this fragment:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
ProcessData();
bytesAvailable -= n;
}
} else
waitForMoreData();
```

Nested ifs

- A nested if is an if statement that is the target of another if or else.
- Nested ifs are very common in programming.

- When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

- The if statements are executed from the top down. As soon as one of the conditions controlling
    - The if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed.
    - If none of the conditions is true, then the final else statement will be executed.
    - The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed.
    - If there is no final else and all other conditions are false, then no action will take place.

```
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

Here is the output produced by the program: April is in the Spring.

Switch

- The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```

- For versions of Java prior to JDK 7, expression must be of **type byte, short, int, char, or an enumeration.**
- Beginning with JDK 7, **expression can also be of type String**. Each value specified in the case statements must be a unique constant expression Duplicate case values are not allowed. The type of each value must be compatible with the type of expression.
- The switch statement works like this: The value of the expression is compared with each of the values in the case statements.
- If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed.

- However, the default statement is optional. If no case matches and no default is present, then no further action is taken. The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. **This has the effect of "jumping out" of the switch**.

```
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
```

i is three.

i is greater than 3.

i is greater than 3.

Note: As you can see, each time through the loop, the statements associated with the case constant that matches i are executed. All others are bypassed. After i is greater than 3, no case statements match, so the default statement is executed.

The break statement is optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple cases without break statements between them. For example, consider the following program:

```
class MissingBreak {
public static void main(String args[]) {
for(int i=0; i<12; i++)
switch(i) {
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
```

```
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

As mentioned, beginning with JDK 7, you can use a string to control a switch statement. For example,

```
class StringSwitch {
public static void main(String args[]) {
String str = "two";
switch(str) {
case "one":
System.out.println("one");
break;
case "two":
System.out.println("two");
break;
case "three":
System.out.println("three");
break;
default:
System.out.println("no match");
```

```
break;
}
}
}
```

As you would expect, the output from the program is
two

Nested switch Statements

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch. For example, the following fragment is perfectly valid:

```
switch(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
break;
case 1: // no conflicts with outer switch
System.out.println("target is one");
break;
}
break;
case 2: // ...
```

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is compared only with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

In summary, there are three important features of the switch statement to note:

• The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.

• No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

• A switch statement is usually more efficient than a set of nested ifs.

Note: When it compiles a switch statement, the Java compiler will inspect each of the case constants and create a **"jump table"** that it will use for **selecting the path of execution** depending on the value of the expression

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of Instructions until a termination condition are met.

While

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
// body of loop
}
```

- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.
- The curly braces are unnecessary if only a single statement is being repeated.

```
class While {
public static void main(String args[]) {
int n = 10;
while(n > 0) {
System.out.println("tick " + n);
```

```
n--;
}
}
}
```

When you run this program, it will “tick” ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

- Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.
- For example, in the following fragment, the call to println() is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

- The body of the while (or any other of Java’s loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.
- For example, consider the following program:

```
// The target of a loop can be empty.
class NoBody {
public static void main(String args[]) {
int i, j;
i = 100;
j = 200;
```

```
// find midpoint between i and j
while(++i < --j); // no body in this loop
System.out.println("Midpoint is " + i);
}
}
```

This program finds the midpoint between i and j. It generates the following output:

Midpoint is 150

Here is how this while loop works.

- The value of i is incremented, and the value of j is decremented. These values are then compared with one another.
- If the new value of i is still less than the new value of j, then the loop repeats. If i is equal to or greater than j, the loop stops.
- Upon exit from the loop, i will hold a value that is midway between the original values of i and j.

do-while

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
// body of loop
} while (condition);
```

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.
- Here is a reworked version of the "tick" program that demonstrates the do-while loop. It generates the same output as before.

```
class DoWhile {
public static void main(String args[]) {
int n = 10;
```

```
do {
System.out.println("tick " + n);
n--;
} while(n > 0);
}
}
```

The loop in the preceding program, while technically correct, can be written more Efficiently as follows:

```
do {
System.out.println("tick " + n);
} while(--n > 0);
```

- In this example, the expression (– –n > 0) combines the decrement of n and the test for zero into one expression.
- Here is how it works. First, the – –n statement executes, decrementing n and returning the new value of n. This value is then compared with zero.
- If it is greater than zero, the loop continues; otherwise, it terminates

For

Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer "for-each" form.
Both types of for loops are discussed here, beginning with the traditional form. Here is the general form of the traditional for statement:

```
for(initialization; condition; iteration) {
// body
}
```

- If only one statement is being repeated, there is no need for the curly braces.
- The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed.
- Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop.

- It is important to understand that the initialization expression is executed only once. Next, condition is evaluated.
- This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed.
- If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.
- The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass.
- This process repeats until the controlling expression is false.

```
class ForTick {
public static void main(String args[]) {
int n;
for(n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a for loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the for. For example, here is the preceding program recoded so that the loop control variable n is declared as an int inside the for:

```
class ForTick {
public static void main(String args[]) {
// here, n is declared inside of the for loop
for(int n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Using the Comma

To allow two or more variables to control a for loop, Java permits you to include multiple statements in both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

```
class Comma {
public static void main(String args[]) {
int a, b;
for(a=1, b=4; a<b; a++, b--) {
System.out.println("a = " + a);
System.out.println("b = " + b);
}
}
}
```

In this example, the initialization portion sets the values of both a and b. The two comma separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

The For-Each Version of the for Loop

The for-each style for automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array.

Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end.

```
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
```

```
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

```
The output from the program is shown here:Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Jump Statements

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program. Each is examined here.

Using break

In Java, the break statement has three uses.

- First, as you have seen, it terminates a statement sequence in a switch statement.
- Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

- Using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using break as a Form of Goto

Java defines an expanded form of the break statement. By using this form of break, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a switch. They can be any block.

Further, you can specify precisely where execution will resume, because this form of break works with a label.

The general form of the labeled break statement is shown here:

```
break label;
```

- Most often, label is the name of a label that identifies a block of code.
- This can be a standalone block of code but it can also be a block that is the target of another statement.
- When this form of break executes, control is transferred out of the named block. The labeled
- block must enclose the break statement, but it does not need to be the immediately enclosing block.
- This means, for example, that you can use a labeled break statement to exit from a set of nested blocks.
- But you cannot use break to transfer control out of a block that does not enclose the break statement.

```
class Break {
public static void main(String args[]) {
boolean t = true;
first: {
second: {
third: {
System.out.println("Before the break.");
if(t) break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}  }
}
```

Running this program generates the following output:

```
Before the break.
This is after second block.
```

One of the most common uses for a labeled break statement is to exit from nested loops.

For example, in the following program, the outer loop executes only once:

```
class BreakLoop4 {
public static void main(String args[]) {
outer: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break outer; // exit both loops
System.out.print(j + " ");
}
System.out.println("This will not print");
}
System.out.println("Loops complete.");
}
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.
```

Using continue

- Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.
- In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.
- For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
```

```
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline.

Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

As with the break statement, continue may specify a label to describe which enclosing loop to continue. Here is an example program that uses continue to print a triangular multiplication table for 0 through 9:

```
class ContinueLabel {
public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
for(int j=0; j<10; j++) {
if(j > i) {
System.out.println();
continue outer;
}
System.out.print(" " + (i * j));
}
}
System.out.println();
```

```
}
}
```

The continue statement in this example terminates the loop counting j and continues with the next iteration of the loop counting i.

Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Return

- The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method.
- Thus, the return statement immediately terminates the method in which it is executed.
- he following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ):

```
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return; // return to caller
System.out.println("This won't execute.");
} }
```

The output from this program is shown here:

Before the return.

Overloading Methods

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
} // Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
```

```
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

```
This program generates the following output:
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- As you can see, test( ) is overloaded four times. The first version takes no parameters,the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

- As you can see, this version of OverloadDemo does not define test(int).

- Therefore,when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call.
- Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Of course, if test(int) had been defined, it would have been called instead.
- Java will employ its automatic type conversions only if no exact match is found.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the Box class developed in the preceding chapter. Following is the latest version of Box:

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

As you can see, the Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since Box( ) requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the Box class is currently written, these other options are not available to you.

s/* Here, Box defines three constructors to initialize the dimensions of a box various ways.*/

```
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the parameters specified when new is executed

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
```

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equalTo(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

- As you can see, the equalTo( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the parameter o in equalTo( ) specifies Test as its type.
- Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of Box allows one object to initialize another:

```
// Here, Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
```

```
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine.The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine.

- Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference.
- In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.
- When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
} class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

- As you can see, the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.
- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Keep in mind that when you create a variable of a class type, you are only creating a reference to an object.
- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
} }
class PassObjRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
```

```
} }
```

This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

Returning Objects

- A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "
+ ob2.a);
}
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.

Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3 ×, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
```

```
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of fact( ) may seem a bit confusing. Here is how it works. When fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(n–1)*n. To evaluate this expression, fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning. The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

Here is one more example of recursion. The recursive method printArray( ) prints the first i elements in the array values.

```
// Another example that uses recursion.
    class RecTest {
    int values[];
    RecTest(int i) {
    values = new int[i];
    }
    // display array -- recursively
    void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
    }
    }
    class Recursion2 {
```

```
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<10; i++) ob.values[i] = i;
ob.printArray(10);
}
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

I Exploring the String Class

- String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type String. Even string constants are actually String objects. For example, in the statement

  System.out.println("This is a String, too");
- The string "This is a String, too" is a String object.
- The second thing to understand about strings is that objects of type String are immutable;
- once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of String, called StringBuffer and StringBuilder, which allow strings to be altered, so all of the normal string manipulations are still available in Java.
- Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

String myString = "this is a test";

- Once you have created a String object, you can use it anywhere that a string is allowed. For example, this statement displays myString:

System.out.println(myString);

- Java defines one operator for String objects: +. It is used to concatenate two strings. For example, this statement
- String myString = "I" + " like " + "Java."; results in myString containing "I like Java."

```
// Demonstrating Strings.
    class StringDemo {
    public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;
    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
    }
    }
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The String class contains several methods that you can use. Here are a few.

- You can test two strings for equality by using equals( ).

- You can obtain the length of a string by calling the length( ) method.
- You can obtain the character at a specified index within a string by calling charAt( ).

The general forms of these three methods are shown here:

```
boolean equals(secondStr)
int length( )
char charAt(index)
```

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " +
strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
        class StringDemo3 {
        public static void main(String args[]) {
        String str[] = { "one", "two", "three" };
        for(int i=0; i<str.length; i++)
        System.out.println("str[" + i + "]: " +
        str[i]);
        }
        }
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```