Java Programming (JPR- 22412)

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Chapter Name** | **Page No** |
| **1.** | **Introduction to Java**     * Fundamentals of Object oriented programming * Java Features * Constants, variables and Data types * Operators and Expressions * Decision making and branching * Decision making and looping | 2 |
| **2.** | **Classes, Objects and Methods**     * Classes, objects and methods * Inheritance * Visibility Control * Arrays, Strings and Vectors | 13 |
| **3.** | **Interfaces and Packages**     * Interface: Multiple Inheritance * Packages | 39 |
| **4.** | **Multithreaded Programming and**  **Exception Handling**     * Multithreading * Managing Errors and Exceptions | 40 |
| **5.** | **Java Applets and Graphics Programming**     * Applet Programming * Graphics Programming | 64 |
| **6.** | **Streams and File I/O**     * Stream Classes * Character Stream, Byte Stream (Reading and Writing Stream) * Serialization | 77 |

Chapter 1: Introduction to JAVA

**1.1: Fundamentals of Object Oriented Programming:**

Object-oriented programming is at the core of Java. In fact, all Java programs are object oriented.

**Classes:** The class is at the core of Java. It defines the shape and nature of an object. When a class is defined, its exact form and nature are declared. This is done by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both.

**Object:** When a class is created, it means a new data type has been created. This type can be used to declare objects of that type. However, 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.

**Data Abstraction:** An essential element of object-oriented programming is *abstraction.* Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of 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 to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a

whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. A powerful way to manage abstraction is through the use of hierarchical classifications. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player.

**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.

**Inheritance:** *Inheritance* is the process by which one object acquires the properties of another object. by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.

**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. For example, a function *Area* to calculate can be used to calculate the area of a *circle* as well as of a *square*. Depending upon the input provided in the function call, appropriate function would be called.

**Dynamic Binding:** It is the mechanism by which a call to a method is resolved at runtime, rather than compile time. It is an important feature because this is how Java achieves run-time polymorphism.

**1.2: Java Features:**

**Compiled and Interpreted:** Java programs are executed in two steps, compile and execute (run). The Java compiler typically compiles the source code into an intermediate code, *bytecode*. That itself is not a machine code for the respective native hardware but is a highly optimized set of instructions designed to be executed by the Java run-time system i.e. *Java Virtual Machine* (JVM). The JVM is an interpreter for the *bytecode*. It accepts the *bytecode* from the Java compiler and translates it into respective native code or machine code.

**Platform Independent and Portable:** Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. The program can be written and compiled on one platform and executed on another. Thus by implementing the concept of *bytecode* Java became Platform independent and Portable.

**Distributed:** Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intraaddress- space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote Method Invocation* (*RMI*). This feature brings an unparalleled level of abstraction to client/server programming.

**Multithreaded and Interactive:** Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

**High Performance:** Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits.

**Robust:** The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas.At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. For example, two of the main reason for program failure: *memory management mistakes (garbage collector) and runtime errors* (exception handling).

**1.3: Constant, Variables and Data Types:**

**Java is a Strongly Typed language:**

Java is a strongly typed language; part of Java’s safety and robustness comes from this fact. In Java, every variable has a type, every expression has a type, and every type is strictly defined. 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.

**Data Types:**

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

■ **Integers:**

This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.

|  |  |  |
| --- | --- | --- |
| **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 |

■ **Floating-point numbers:**

This group includes **float** and **double**, which represent numbers with fractional precision.

**Name Width in Bits Approximate Range Double** 64 4.9e–324 to 1.8e+308

**Float** 32 1.4e−045 to 3.4e+038

■ **Characters:**

This group includes **char**, which represents symbols in a character set, like letters and numbers. **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integer type that 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. For this purpose, it requires 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 **char**s. 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.

■ **Boolean:**

This group includes **boolean**, which is a special type for representing true/false values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, such as **a < b**.

**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.

* **Declaring a variable:** The basic form of a variable declaration is: Syntax: *type identifier* [ = *value*][, *identifier* [= *value*] ...] ;

E.g.: int a, b, c; int d=3, e, f=5; double pi=3.14;

char x=’x’;

* **Dynamic Initialization:** Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. E.g.: double a=3.0, b=4.0;

double c= a \* b ;

**Scope of variables:**

Java allows variables to be declared within any block. A block begins with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time a new block is started, a new scope is created. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when a variable is declared within a scope, that variable is localized and protected from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

E.g.:

// Demonstrate block scope.

class Scope {

public static void main(String args[]) { int x; // known to all code within main x = 10;

if(x == 10) { // start new scope

int y = 20; // known only to this block // x and y both known here.

System.out.println("x and y: " + x + " " + y); x = y \* 2;

}

// y = 100; // Error! y not known here // x is still known here.

System.out.println("x is” + x);

}

}

**Constant:**

A constant value in Java is created by using a *literal* representation of it. It is typed directly into the program whenever it is needed.

E.g.: 100 98.6 ‘X’ “This is a test”

In the above example 100 is a constant, it cannot be assigned any value. A constant is represented by its value.

**Symbolic Constant:**

A symbolic constant, on the other hand is represented by a name, just as a variable is represented. Unlike a variable, however, after a symbolic constant is initialized its value cannot be changed. In Java, a symbolic constant is created using the keyword final. E.g.: final int x=100;

**Type Conversion and Casting:**

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. This is called **type conversion.** 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 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.

1. **Java’s Automatic Conversion (widening):**

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.

1. **Casting Incompatible Types:**

When one type of data is assigned to an incompatible type of variable and the destination is smaller than the source type *narrowing conversion* takes place. This conversion will not be performed automatically. To create a conversion between two incompatible types, *cast* must be used.

Syntax: (target-type) value

E.g.: int a; byte b;

…..

b=(byte) a;

**1.4: Operators and Expressions:**

**Arithmetic operators:**

**Operator Result**

+ Addition

– Subtraction (also unary minus)

\* Multiplication

/ Division

% Modulus

**Relational operators:**

**Operator Result**

== Equal to

!= Not equal to

> Greater than

|  |  |
| --- | --- |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

**Logical operators:**

**Operator Result**

& Logical AND | Logical OR

|  |  |
| --- | --- |
| ^ | Logical XOR (exclusive OR) |
| == | Equal to |
| != | Not equal to |

**Assignment operators:**

**Operator Result**

= Assignment

|  |  |
| --- | --- |
| += | Addition assignment |
| –= | Subtraction assignment |
| \*= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulus assignment |

**Increment and Decrement operators:**

**Operator Result**

++ Increment

-- Decrement

**Conditional operators:**

**Bitwise operators:**

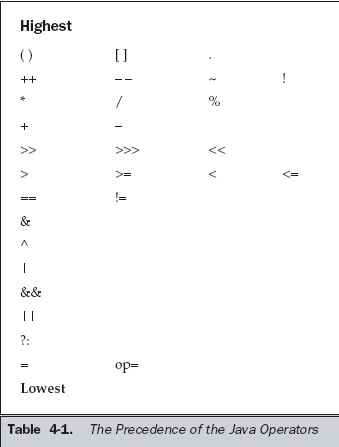
|  |  |
| --- | --- |
| **Operator** | **Result** |
| ~ | Bitwise unary NOT |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| >> | Shift right |
| >>> | Shift right zero fill |
| << | Shift left |
| &= | Bitwise AND assignment |
| |= | Bitwise OR assignment |
| ^= | Bitwise exclusive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

**Special operators:**

**Operator Result**

?: Ternary if-then-else

**Operator Precedence:**



**1.5: Decision making and Branching:**

**Simple If statement:**

The **if** statement is Java’s conditional branch statement. It can be used to execute a part of program only if some conditions holds true.

Syntax:

if (*condition*) *statement1*;

**The if else statement:**

It can be used to route program execution through two different paths.

Syntax:

if (*condition*) *statement1*;

else *statement2*;

**Nested ifs:**

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, 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**.

Syntax:

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

**Else-if ladder:**

A common programming construct that is based upon a sequence of nested **if**s is the ***if-else-if*** *ladder*. Syntax:

if(*condition*)

*statement;*

else if*(condition*)

*statement*;

else if(*condition*)

*statement*;

... else

*statement*;

**Switch:**

The **switch** statement is Java’s multiway branch statement. It provides an easy way to transfer 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.

Syntax:

switch (*expression*) { case *value1*:

// statement sequence

break; case *value2*:

// statement sequence

break; ... case *valueN*:

// statement sequence

break; default:

// default statement sequence

}

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed. Similar to an if statement, nested switch can also be implemented.

The value of the expression is compared with each of the literal 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**.

**Ternary operator ( ? : )**

**1.6: Decision making and Looping:**

Java’s iteration statements are for, while, and do-while. These statements create what we commonly call *loops.*

**While:**

The **while** loop is Java’s most fundamental looping statement. It repeats a statement or block while its controlling expression is true.

Syntax:

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.

**Do-While:**

Sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.

Syntax:

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.

**For:**

The general form of the **for** statement:

for(*initialization*; *condition*; *iteration*) {

// body

}

In its most common form, the *initialization* portion of the loop sets a loop control variable to an initial value. The *condition* is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the **for** loop continues to iterate. If it is false, the loop terminates. The *iteration* expression determines how the loop control variable is changed each time the loop iterates.

**Jump statements:**

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

**break:**

The **break** statement has two uses. First, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop.

**Break to exit Loop:**

By 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. E.g.:

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.");

} } o/p:

i:0

i: 1

i: 2

i: 3

i: 4

i: 5

i: 6

i: 7

i: 8

i: 9

Loop complete.

The **break** statement can be used with any of Java’s loops, including intentionally infinite loops. When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.

**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.

E.g.: Here is an example program that uses **continue** to cause two numbers to be printed on each

line:

// Demonstrate continue. 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("");

}

}

}

\*\*\*\*\*\*\*

Chapter 2: Classes, Objects and Methods

**2.1: Classes, objects and methods:**

**Defining a Class:**

A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. A class is a logical construct.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.* The code that operates on that data is referred to as *member methods* or just *methods.*

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class’ public methods, you can ensure that no improper actions take place. The general form of class definition is shown below: class *classname* {  *type instance-variable1*;  *type instance-variable2*;

// ...

*type instance-variableN*;  *type methodname1*(*parameter-list*) {

// body of method

}

*type methodname2*(p*arameter-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.

E.g.:

class Box { double width; double height;

double depth;

}

**Creating Methods:**

Classes usually consist of two things: instance variables and methods. The general form of a method is shown below:

*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*;

E.g.:

// 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 //Calling the method. mybox1.volume();

// display volume of second box mybox2.volume();

}

}

**Creating Objects:**

When a class is created, it creates a new *data type*. This type is used to create objects, which will be of the same type.

Creating object of a class is two step process:

1. Declare a variable of the class type.
2. Acquire an actual, physical copy of the object and assign it to that variable. This can be achieved using **new** operator. The **new** operator dynamically allocates memory for an object and returns a reference to it.

Syntax:

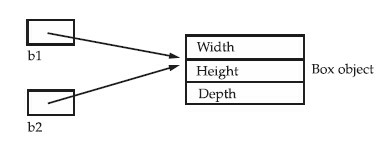
*class\_name object\_name=new class\_name();*

E.g.:

Box b1=new Box();

Box b2=b1;

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.



**Accessing class members:**

To access class members, *dot (.) operator* is used. The dot operator links the name of the object with the name of an instance variable or method

The example shown in **creating object** section displays the use of *dot (.) operator*.

**Constructor:**

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 is the constructor’s job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.
* 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 immediately after the object is created, before the **new** operator completes.
* Constructors have no return type, not even **void**. This is because the implicit return type of a class’ constructor is the class type itself.
* Similar to a method, a constructor can have parameters.

E.g.:

/\* 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);

}

}

Output:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

**Method Overloading:**

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 implements 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. The following example illustrates method overloading:

// 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);

}

}

Output:

No parameters a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

**Overloading Constructor:**

In addition to overloading normal methods, constructor can also be overloaded.

E.g.:

/\* 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);

}

}

Output:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

**Static Members:**

To define a class member that will be used independently of any object of that class, precede the declaration with the keyword **static.** When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

* They can only call other static methods.
* They must only access static data.
* They cannot refer to this or super in any way.

A **static** block which gets executed exactly once, when the class is first loaded.

E.g.:

// Demonstrate static variables, methods, and blocks.

class UseStatic { static int a = 3; static int b;

static void meth(int x) { System.out.println("x = " + x);

System.out.println("a = " + a);

System.out.println("b = " + b);

}

static {

System.out.println("Static block initialized.");

b = a \* 4;

}

public static void main(String args[]) {

meth(42);

}

}

Output:

Static block initialized.

x = 42 a = 3

b = 12

**2.2: Inheritance:**

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. The following program creates a superclass called **A** and a subclass called **B**.

// Create a superclass.

class A {

int i, j;

void showij() {

System.out.println("i and j: " + i + " " + j);

}

}

// Create a subclass by extending class A.

class B extends A {

int k; void showk() {

System.out.println("k: " + k);

}

void sum() {

System.out.println("i+j+k: " + (i+j+k));

}

}

class SimpleInheritance {

public static void main(String args[]) {

A superOb = new A(); B subOb = new B();

// The superclass may be used by itself. superOb.i = 10; superOb.j = 20;

System.out.println("Contents of superOb: "); superOb.showij(); System.out.println();

/\* The subclass has access to all public members of its superclass. \*/ subOb.i = 7; subOb.j = 8;

subOb.k = 9;

System.out.println("Contents of subOb: "); subOb.showij();

subOb.showk();

System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();

}

}

Output:

Contents of superOb:

i and j: 10 20 Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

✔ Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

**Defining a subclass constructor:**

// Create a superclass.

class A {

int i, j;

A(int x, int y)

{ i=x; j=y;

}

void showij() {

System.out.println("i and j: " + i + " " + j);

} }

// Create a subclass by extending class A.

class B extends A {

int k;

B(int x, int y, int z)

{ i=x; j=y; k=z;

}

void showk() {

System.out.println("k: " + k);

}

void sum() {

System.out.println("i+j+k: " + (i+j+k));

}

}

class SimpleInheritance {

public static void main(String args[]) {

B subOb = new B(10.20.30);

System.out.println("Contents of superOb: "); superOb.showij(); System.out.println();

System.out.println("Contents of subOb: ");

subOb.showk();

System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();

}

}

**Super:**

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. **super** has two general forms.

* The first calls the superclass’ constructor.
* The second is used to access a member of the superclass that has been hidden by a member of a subclass.

**Using super to call a superclass constructor:**

A subclass can call a constructor method defined by its superclass by use of the following form of **super**: super(*parameter*-*list*);

Here, *parameter*-*list* specifies any parameters needed by the constructor in the superclass. **super()** must always be the first statement executed inside a subclass’ constructor.

// Create a superclass.

class A { int i, j;

A(int x, int y)

{ i=x; j=y;

} void showij() {

System.out.println("i and j: " + i + " " + j);

} }

// Create a subclass by extending class A.

class B extends A {

int k;

B(int x, int y, int z)

{

Super(x,y); k=z;

} void showk() {

System.out.println("k: " + k);

}

void sum() {

System.out.println("i+j+k: " + (i+j+k));

}

}

class SimpleInheritance {

public static void main(String args[]) {

B subOb = new B(10.20.30);

System.out.println("Contents of superOb: "); superOb.showij(); System.out.println();

System.out.println("Contents of subOb: ");

subOb.showk();

System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();

}

}

**Using super to access a member of the superclass that has been hidden by a member of a subclass:**

super can also be used to access a member of the superclass that has been hidden by a member of a subclass. It has the following general form:

super.*member*

// Using super to overcome name hiding.

class A { int i;

}

// Create a subclass by extending class A.

class B extends A {

int i; // this i hides the i in A B(int a, int b) { super.i = a; // i in A i = b; // i in B

}

**Types of Inheritance:**

**Multilevel inheritance:**

it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.

**Hierarchical inheritance:**

In hierarchical type of inheritance, **one class is extended by many subclasses**. It is **one-tomany** relationship.

**Overriding Methods:**

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

E.g.:

// Method overriding.

class A {

int i, j;

A(int a, int b) { i = a; j = b;

}

// display i and j void show() {

System.out.println("i and j: " + i + " " + j); }

}

class B extends A {

int k;

B(int a, int b, int c) { super(a, b); k = c;

}

// display k – this overrides show() in A void show() {

System.out.println("k: " + k); }

}

class Override { public static void main(String args[]) {

B subOb = new B(1, 2, 3);

subOb.show(); // this calls show() in B }

}

Output:

k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**.

class B extends A {

int k;

B(int a, int b, int c) { super(a, b); k = c;

} void show() { super.show(); // this calls A's show()

System.out.println("k: " + k); }

}

Output:

# 

i and j: 1 2

k: 3

**Abstract class:**

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, when a superclass is created that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

* Any class that contains one or more abstract methods must also be declared abstract.
* To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
* There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
* Abstract constructors, or abstract static methods cannot be declared.
* Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

// A Simple demonstration of abstract. abstract class A { abstract void callme();

// concrete methods are still allowed in abstract classes void callmetoo() {

System.out.println("This is a concrete method."); } } class B extends A { void callme() {

System.out.println("B's implementation of callme.");

}

}

class AbstractDemo { public static void main(String args[]) { B b = new B(); b.callme();

b.callmetoo();

}

}

**Final variables, methods and classes:**

**Using final with variables:**

The final keyword converts variable into constant whose value cannot be changed during the execution of the program.

Syntax: *final data\_type var\_name;*

**Using final with methods:**

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

E.g.:

class A { final void meth() {

System.out.println("This is a final method."); }

}

class B extends A { void meth() { // ERROR! Can't override.

System.out.println("Illegal!");

}

}

**Using final with class:**

To prevent a class from being inherited, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. E.g.:

final class A {

// ...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A

// ...

}

**Visibility Control:**

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Java addresses four categories of visibility for class members:

■ Subclasses in the same package

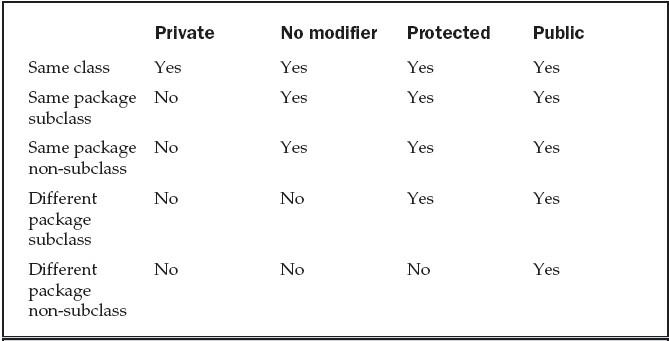
■ Non-subclasses in the same package

■ Subclasses in different packages

■ Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.



**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.

**One-Dimensional Array:**

To create an array, first an array variable of the desired type must be created. The general form of a one dimensional array declaration is:

*type var-name*[ ];

E.g.: int month\_days[];

Although this declaration establishes the fact that **month\_days** is an array variable, no array actually exists. In fact, the value of **month\_days** is set to **null**, which represents an array with no value. 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.

The general form of array initialization using **new** operator is given below:

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

E.g.:

month\_days = new int[12];

Arrays can be initialized when they are declared. 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**.

E.g.:

class AutoArray { 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;

// …. int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; System.out.println("April has " + month\_days[3] + " days.");

}

}

**Two-Dimensional Array:**

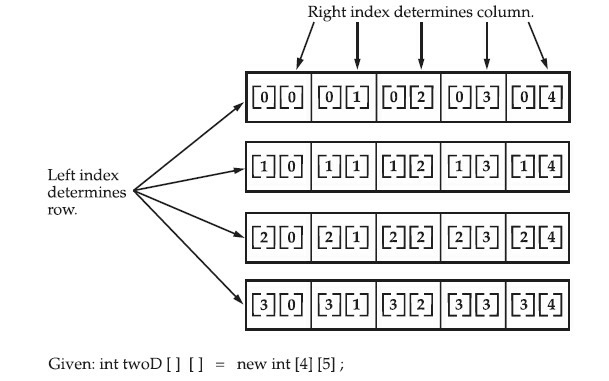
Syntax for declaring and initializing a two-dimensional array is given below:

t*ype var-name[][]=new type[size][size];*

E.g.:

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

This allocates a 4 by 5 array and assigns it to **twoD**. Conceptually, this array will look like the one shown below.



E.g.: // 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();

}

}

}

Output: 0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19 **String and String Buffer:**

In Java, a ***string***is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

When you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.

String Constructor:

The String class supports several constructors. To create an empty string, default constructor is used.

E.g.: String s= new String();

To create a **String** initialized by an array of characters,

E.g.: char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);

Another way of creating a String,

E.g.: String s=new String(“Hello”); //using new operator

Or

String s=“abc”; //using literals

**String Functions:**

**String Length:**

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

Syntax:

*int length( )*

**String Concatenation:**

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result.

E.g.: String age = "9";

String s = "He is " + age + " years old."; **charAt():**

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

Syntax:

*char charAt(int where)*

**getChars():**

To extract more than one character at a time, **getChars( )** method is used. It has this general form:

Syntax:

*void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)*

**equals and equalsIgnoreCase():**

To compare two strings for equality, use **equals( )**. It has this general form:

Syntax:

*boolean equals(Object str)* The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**.

Syntax:

*boolean equalsIgnoreCase(String str)*

**regionMatches():**

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.

Syntax:

*boolean regionMatches(int startIndex, String str2,int str2StartIndex, int numChars)*

*boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)*

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

**startsWith() and endsWith():**

The **startsWith( )** method determines whether a given **String** begins with a specified string.

Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

*boolean startsWith(String str)*

*boolean endsWith(String str)*

**equals() Vs ==**

The equals( ) method and the == operator perform two different operations. the **equals( )** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance. E.g.:

class EqualsNotEqualTo { public static void main(String args[]) {

String s1 = "Hello";

String s2 = new String(s1);

System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));

}

}

Output:

Hello equals Hello -> true

Hello == Hello -> false

**compareTo():**

To know which is *less than, equal to,* or *greater than* the next compareTo() function is used. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.

Syntax:

*int compareTo(String str)*

**Value Meaning**

Less than zero The invoking string is less than *str.*

Greater than zero The invoking string is greater than *str.*

Zero The two strings are equal.

**Vector:**

**Vector** implements a dynamic array. It is similar to **ArrayList.** Here are the **Vector** constructors:

*Vector( )*

*Vector(int size)*

*Vector(int size, int incr)*

*Vector(Collection c)*

The first form creates a default vector, which has an **initial size of 10**. The second form creates a vector whose initial capacity is specified by *size.* The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr.* The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection *c*. This constructor was added by Java 2.

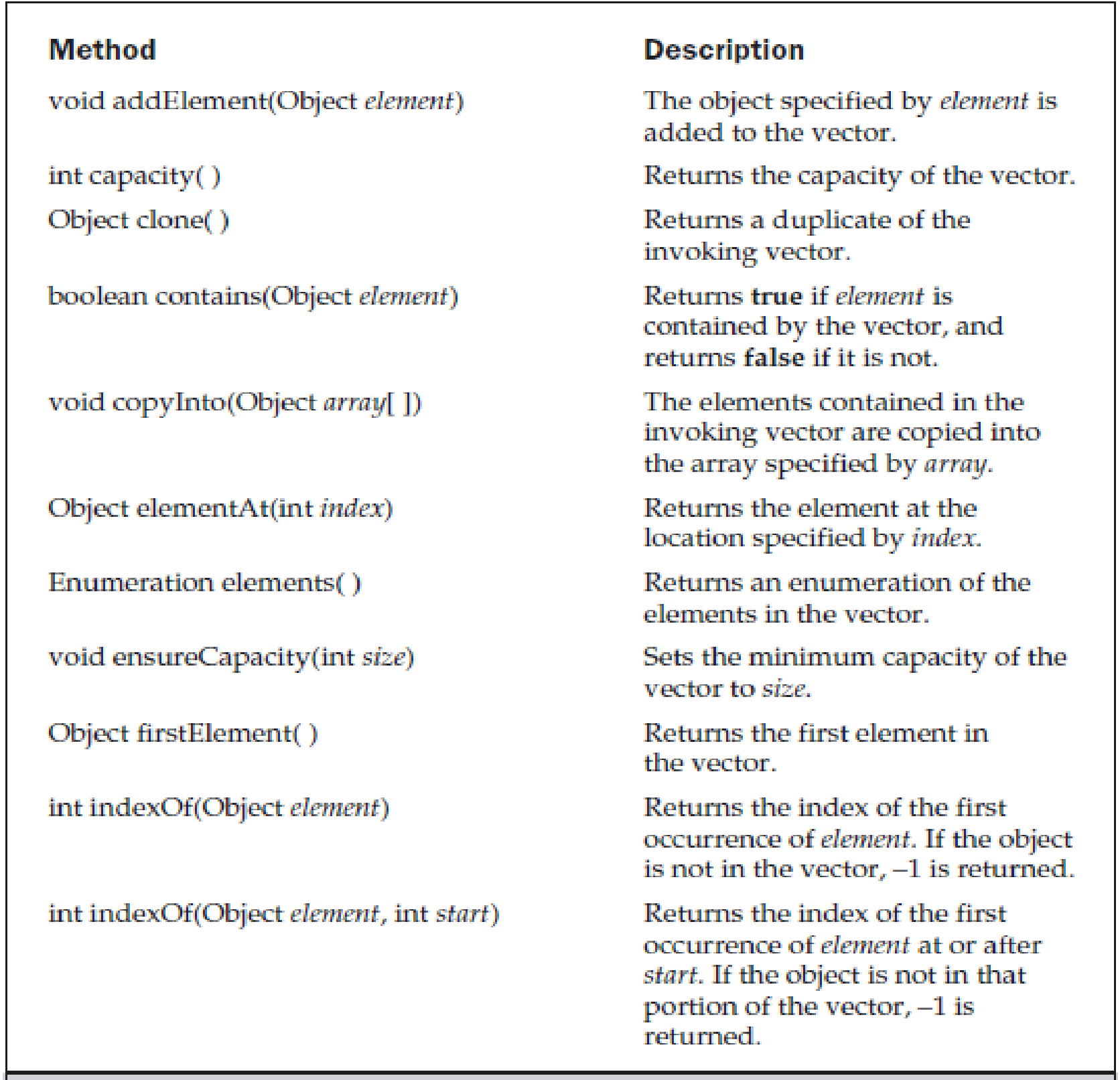
All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don’t specify an increment, the vector’s size is doubled by each allocation cycle.

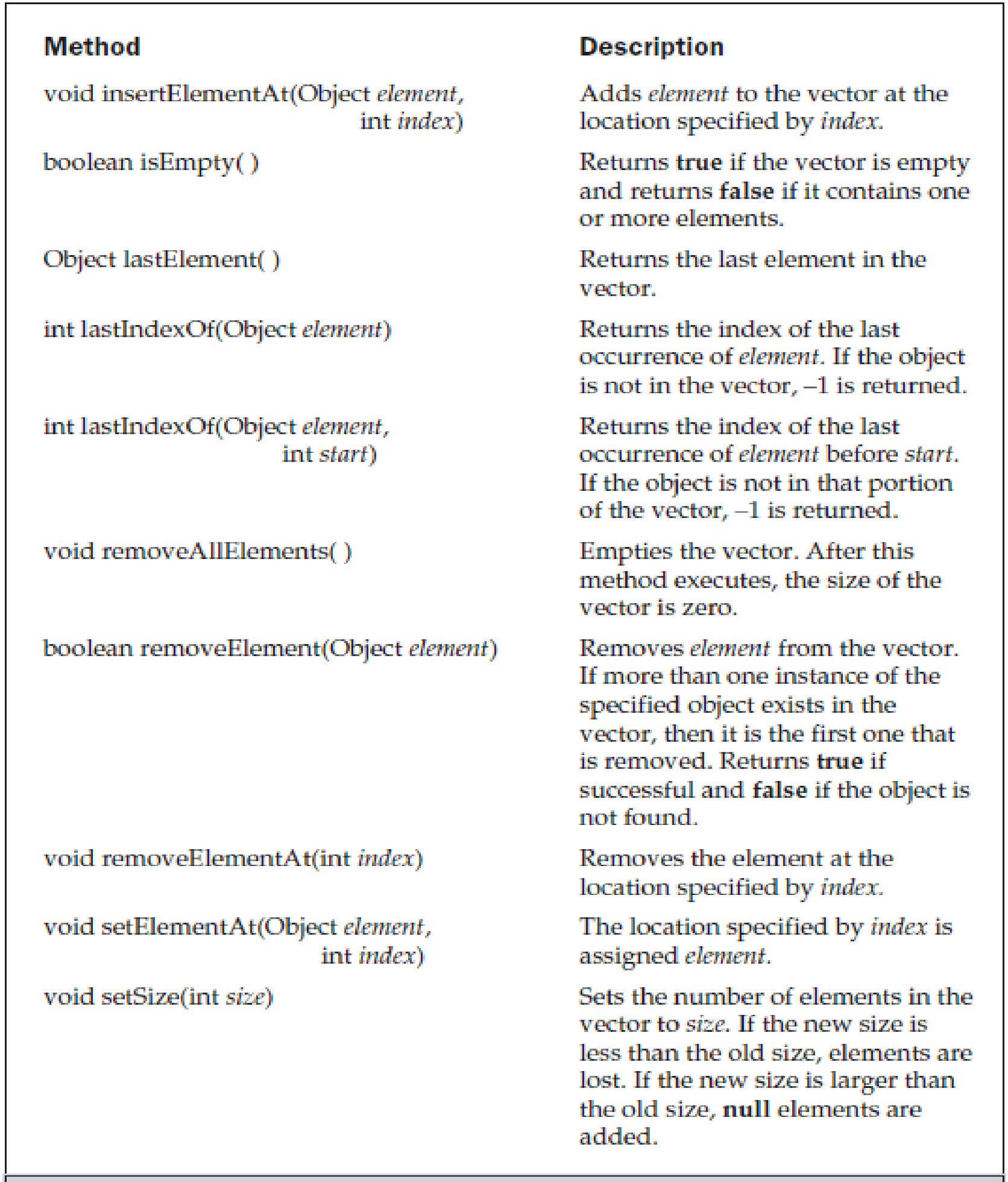
**Vector** defines these protected data members:

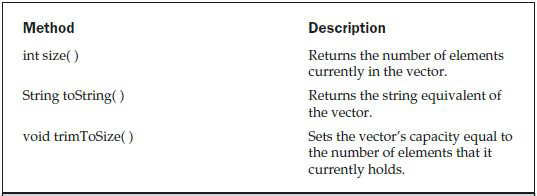
int capacityIncrement; int elementCount; Object elementData[ ];

The increment value is stored in **capacityIncrement**. The number of elements currently in the vector is stored in **elementCount**. The array that holds the vector is stored in **elementData**.

Following are the methods defined by Vector:







The following program demonstrates various vector operation:

import java.util.\*; class VectorDemo { public static void main(String args[]) {

// initial size is 3, increment is 2 Vector v = new Vector(3, 2);

System.out.println("Initial size: " + v.size()); System.out.println("Initial capacity: " + v.capacity());

v.addElement(new Integer(1));

v.addElement(new Integer(2));

v.addElement(new Integer(3));

v.addElement(new Integer(4));

System.out.println("Capacity after four additions: " + v.capacity());

v.addElement(new Double(5.45)); System.out.println("Current capacity: " + v.capacity());

v.addElement(new Double(6.08));

v.addElement(new Integer(7)); System.out.println("Current capacity: " + v.capacity());

v.addElement(new Float(9.4));

v.addElement(new Integer(10)); System.out.println("Current capacity: " + v.capacity());

v.addElement(new Integer(11));

v.addElement(new Integer(12)); System.out.println("First element: " +

(Integer)v.firstElement());

System.out.println("Last element: " +

(Integer)v.lastElement()); if(v.contains(new Integer(3))) System.out.println("Vector contains 3."); // enumerate the elements in the vector. Enumeration vEnum = v.elements(); System.out.println("\nElements in vector:"); while(vEnum.hasMoreElements()) System.out.print(vEnum.nextElement() + " ");

System.out.println();

}

}

Output:

Initial size: 0

Initial capacity: 3

Capacity after four additions: 5

Current capacity: 5

Current capacity: 7

Current capacity: 9

First element: 1 Last element: 12 Vector contains 3.

Elements in vector:

1 2 3 4 5.45 6.08 7 9.4 10 11 12

**Wrapper Classes:**

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For this conversion, the designers introduced **wrapper classes.**

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used.

Wrapper classes include methods to unwrap the object and give back the data type.

There is a wrapper class for every primitive date type in Java. This class encapsulates a single value for the primitive data type. For instance the wrapper class for int is Integer, for float is Float, and so on. Remember that the primitive name is simply the lowercase name of the wrapper except for char, which maps to Character, and int, which maps to Integer.

**The wrapper classes in the Java API serve two primary purposes:**

* To provide a mechanism to “wrap” primitive values in an object so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.
* To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

The wrapper object of a wrapper class can be created in one of two ways: by instantiating the wrapper class with the new operator or by invoking a static method on the wrapper class.

**Creating Wrapper Objects with the new Operator**

Before we can instantiate a wrapper class, we need to know its name and the arguments its constructor accepts. The name of the wrapper class corresponding to each primitive data type, along with the arguments its constructor accepts, is listed below:

Primitive datatype-->Wrapper Class-->Constructor arguments

* boolean--> Boolean--> boolean or String
* byte--> Byte--> byte or String
* char--> Character--> char
* short--> Short--> short or String
* int-->Integer--> int or String
* long--> Long--> long or String
* float-->Float--> float double or String
* double-->Double--> double or String

All the wrapper classes are declared final. That means you cannot derive a subclass from any of them. All the wrapper classes except Boolean and Character are subclasses of an abstract class called Number, whereas Boolean and Character are derived directly from the Object class. All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed.

e.g. Boolean wboo = new Boolean("false");

Integer wint = new Integer("16");

Integer yint = new Integer(16);

**Wrapping Primitives Using a static Method**

All wrapper classes offers static valueOf() methods which give you another approach to creating wrapper objects. Because it's a static method, it can be invoked directly on the class (without instantiating it), and will return the corresponding object that is wrapping what you passed in as an argument.

**Value() method:**

When you need to convert the value of a wrapped numeric to a primitive, use one of the many Value() methods. All of the methods in this family are no-arg methods. Each of the six numeric wrapper classes has six methods, so that any numeric wrapper can be converted to any primitive numeric type.

E.g.:

Integer i2 = new Integer(42); // make a new wrapper object byte b = i2.byteValue(); // convert i2's value to a byte

**Parsexxx(String) method:**

If you do not need to store a value in a wrapper but just want to perform a quick operation on it, such as converting the type, you can do it by using an appropriate static method of the appropriate wrapper class.

E.g.: String s = "123"; int i = Integer.parseInt(s);

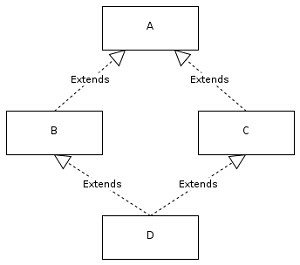
Chapter 3: Interfaces and Packages

**Multiple Inheritance:**

Multiple Inheritance is a feature of object oriented programming languages in which a class can inherit characteristics and features from more than one superclass. But Java does not support Multiple Inheritance.

**Why Java doesn’t support Multiple Inheritance?**

**Simplicity:** In order to enforce simplicity should be the main reason for omitting multiple inheritance. For e.g. consider **diamond problem of multiple inheritance.**



We have two classes B and C inheriting from A. Assume that B and C are [overriding](http://javapapers.com/core-java/overloading-and-overriding/) an inherited method and they provide their own implementation. Now D inherits from both B and C doing multiple inheritance. D should inherit that overridden method, which overridden method will be used? Will it be from B or C? Here we have an ambiguity.

Chapter 4: Multithreaded Programming and Exception

# Handling

**4.1: Multithreading**

Unlike most other programming languages, Java provides built-in support for *multithreaded programming*.A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

There are two distinct types of multitasking: process-based and thread-based. Processbased multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

**The Thread Class and Runnable Interface:**

Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. A new thread can be created by either extending **Thread** or implementing the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Some of them are listed below:

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| getName | Obtain a thread’s name. |
| getPriority | Obtain a thread’s priority. |
| isAlive | Determine if a thread is still running. |
| join | Wait for a thread to terminate. |
| run | Entry point for the thread. |
| sleep | Suspend a thread for a period of time. |
| start | Start a thread by calling its run method. |

Used for interprocess communication:

Wait tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.

notify wakes up the first thread that called **wait( )** on the same object.

notifyAll wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

**The Main Thread:**

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of the program, because it is the one that is executed when program begins. The main thread is important for two reasons:

■ It is the thread from which other “child” threads will be spawned.

■ Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown below:

*static Thread currentThread( )*

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

**Creating a Thread:**

A Thread can be created by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

■ You can implement the **Runnable** interface.

■ You can extend the **Thread** class, itself.

**Implementing Runnable:**

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. One can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**,

*public void run( )*

Inside **run( )**, the code that constitutes the new thread can be defined. **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within main program. This thread will end when **run( )** returns.

After a class that implements **Runnable** is created, an object of type **Thread** will be instantiated from within that class. **Thread** defines several constructors. One type is shown

below:

*Thread(Runnable threadOb, String threadName)*

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until its **start( )** method is called, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here: *void start( )*

Following is an example that creates a new thread and starts it running:

// Create a second thread.

class NewThread implements Runnable {

Thread t;

NewThread() {

// Create a new, second thread t = new Thread(this, "Demo Thread"); System.out.println("Child thread: " + t); t.start(); // Start the thread

}

// This is the entry point for the second thread.

public void run() { try {

for(int i = 5; i > 0; i--) {

System.out.println("Child Thread: " + i); Thread.sleep(500);

}

}

catch (InterruptedException e) {

System.out.println("Child interrupted."); }

System.out.println("Exiting child thread."); }

}

class ThreadDemo { public static void main(String args[]) { new NewThread(); // create a new thread try {

for(int i = 5; i > 0; i--) {

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e) {

System.out.println("Main thread interrupted."); }

System.out.println("Main thread exiting."); }

}

Inside **NewThread**’s constructor, a new **Thread** object is created by the following statement:

*t = new Thread(this, "Demo Thread");*

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread of executionsbeginning at the **run( )** method. After calling **start( )**, **NewThread**’s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows:

Output:

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1 Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

**Extending a Thread:**

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Here is the preceding program rewritten to extend Thread:

// Create a second thread by extending Thread

class NewThread extends Thread {

NewThread() {

// Create a new, second thread super("Demo Thread");

System.out.println("Child thread: " + this); start(); // Start the thread

}

// This is the entry point for the second thread. public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Child Thread: " + i); Thread.sleep(500);

}

}

catch (InterruptedException e) {

System.out.println("Child interrupted.");

}

System.out.println("Exiting child thread."); }

}

class ExtendThread { public static void main(String args[]) { new NewThread(); // create a new thread try {

for(int i = 5; i > 0; i--) {

System.out.println("Main Thread: " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e) {

System.out.println("Main thread interrupted."); }

System.out.println("Main thread exiting."); }

}

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

*public Thread(String threadName)*

Here, *threadName* specifies the name of the thread.

**Stopping and Blocking a Thread:**

As mentioned, the main thread must be to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?

Fortunately, **Thread** provides a means by which one can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown below:

*final boolean isAlive( )*

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive( )** is occasionally useful, the method that is more commonly used to wait for a thread to finish is called **join( )**, shown here:

*final void join( ) throws InterruptedException*

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.

// Using join() to wait for threads to finish.

class NewThread implements Runnable {

String name; // name of thread

Thread t;

NewThread(String threadname) { name = threadname; t = new Thread(this, name);

System.out.println("New thread: " + t); t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() { try {

for(int i = 5; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

}

catch (InterruptedException e) {

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting."); }

}

class DemoJoin { public static void main(String args[]) {

NewThread ob1 = new NewThread("One");

NewThread ob2 = new NewThread("Two");

NewThread ob3 = new NewThread("Three");

System.out.println("Thread One is alive: "+ ob1.t.isAlive());

System.out.println("Thread Two is alive: "+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "+ ob3.t.isAlive());

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish."); ob1.t.join();

ob2.t.join(); ob3.t.join();

}

catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Thread One is alive: "+ ob1.t.isAlive());

System.out.println("Thread Two is alive: "+ ob2.t.isAlive());

System.out.println("Thread Three is alive: "+ ob3.t.isAlive());

System.out.println("Main thread exiting."); }

}

Output:

Sample output from this program is shown here:

New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

Thread One is alive: true

Thread Two is alive: true Thread Three is alive: true Waiting for threads to finish.

One: 5

Two: 5

Three: 5

One: 4

Two: 4

Three: 4

One: 3

Two: 3

Three: 3

One: 2

Two: 2

Three: 2

One: 1

Two: 1

Three: 1 Two exiting.

Three exiting.

One exiting.

Thread One is alive: false

Thread Two is alive: false Thread Three is alive: false Main thread exiting.

**Stopping and Blocking a Thread:**

A program can use **suspend( )** and **resume( )**, which are methods defined by **Thread**, to pause (block) and restart the execution of a thread. They have the form shown below:

*final void suspend( )*

*final void resume( )*

The following program demonstrates these methods:

// Using suspend() and resume().

class NewThread implements Runnable {

String name; // name of thread

Thread t;

NewThread(String threadname) { name = threadname; t = new Thread(this, name);

System.out.println("New thread: " + t); t.start(); // Start the thread

}

// This is the entry point for thread.

public void run() { try {

for(int i = 15; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(200);

}

}

catch (InterruptedException e) {

System.out.println(name + " interrupted.");

}

System.out.println(name + " exiting."); }

}

class SuspendResume { public static void main(String args[]) {

NewThread ob1 = new NewThread("One"); NewThread ob2 = new NewThread("Two"); try {

Thread.sleep(1000); ob1.t.suspend();

System.out.println("Suspending thread One");

Thread.sleep(1000); ob1.t.resume();

System.out.println("Resuming thread One"); ob2.t.suspend();

System.out.println("Suspending thread Two");

Thread.sleep(1000); ob2.t.resume(); System.out.println("Resuming thread Two");

}

catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

// wait for threads to finish

try {

System.out.println("Waiting for threads to finish."); ob1.t.join(); ob2.t.join();

}

catch (InterruptedException e) {

System.out.println("Main thread Interrupted");

}

System.out.println("Main thread exiting."); }

}

Output:

Sample output from this program is shown here:

New thread: Thread[One,5,main]

One: 15

New thread: Thread[Two,5,main]

Two: 15

One: 14

Two: 14

One: 13

Two: 13

One: 12

Two: 12

One: 11

Two: 11

Suspending thread One

Two: 10

Two: 9

Two: 8

Two: 7

# Two: 6

Resuming thread One

Suspending thread Two

One: 10

One: 9

# One: 8

One: 7

One: 6

Resuming thread Two

Waiting for threads to finish.

Two: 5

One: 5

Two: 4

One: 4

Two: 3

One: 3

Two: 2

One: 2

Two: 1 One: 1

Two exiting.

One exiting.

Main thread exiting.

The **Thread** class also defines a method called **stop( )** that stops a thread. Its signature is shown here: *final void stop( )*

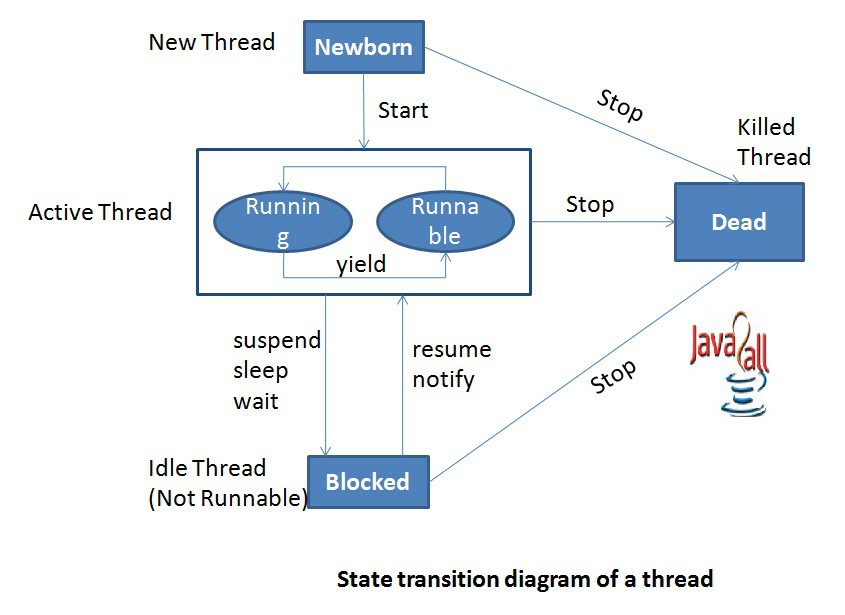
Once a thread has been stopped, it cannot be restarted using **resume( )**.

**Life Cycle of Thread:**

A Thread has many different states through its life:

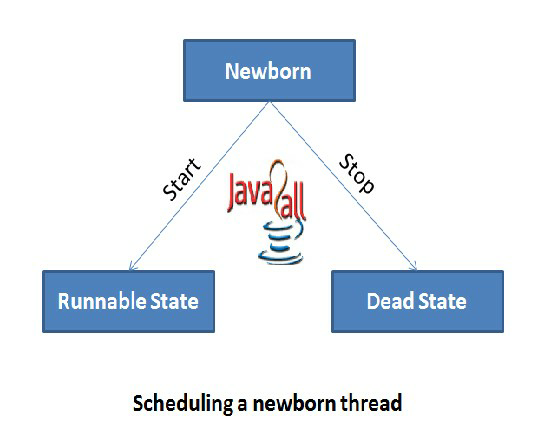
1. Newborn State
2. Runnable State
3. Running State
4. Blocked State
5. Dead State

Thread should be in any one state of above and it can move from one state to another by different methods and ways.



**Newborn State:**

When a thread is created, it is in newborn state. In this state a thread is just created and still not running. A thread can be moved to runnable state by invoking start() method and it can be moved to Dead state by invoking stop() method.

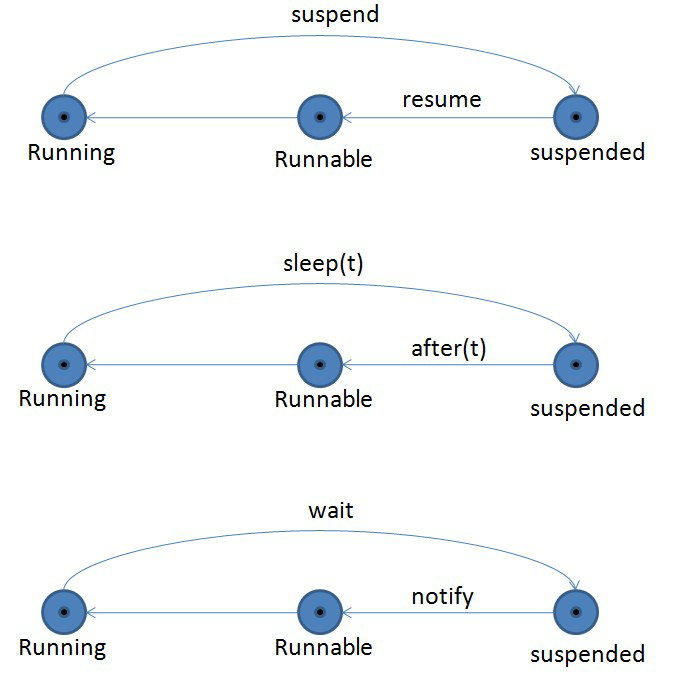


**Runnable State:**

When a thread is in this state it is ready for running and its waiting for the control. Control can be moved to another thread by yield() method.

**Running State:**

In this state, a thread is in execution mode because the control of cpu is given to that particular thread. A thread can move in three different situations from running mode.



**Blocked State:**

A thread is called in Blocked state when it is not allowed to enter runnable or running state. It happens when thread is in waiting, suspended or sleeping mode.

**Dead State:**

When a thread completes executing its run() method, the of that particular thread ends. A thread can be killed by invoking stop() method.

**Thread Priorities:**

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

To set a thread’s priority, **setPriority( )** method is used, which is a member of **Thread**. This is its general form:

*final void setPriority(int level)*

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN\_PRIORITY** and **MAX\_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM\_PRIORITY**, which is currently 5. These priorities are defined as **final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here: *final int getPriority( )*

E.g.: // Demonstrate thread priorities.

class clicker implements Runnable {

int click = 0; Thread t;

private volatile boolean running = true; public clicker(int p) { t = new Thread(this); t.setPriority(p);

t.start(); }

public void run() { while (running) { click++;

}

}

public void stop() { running = false;

}

}

class HiLoPri { public static void main(String args[]) {

Thread.currentThread().setPriority(Thread.MAX\_PRIORITY); clicker hi = new clicker(Thread.NORM\_PRIORITY + 2); clicker lo = new clicker(Thread.NORM\_PRIORITY - 2); try {

Thread.sleep(10000);

}

catch (InterruptedException e) {

System.out.println("Main thread interrupted.");

} lo.stop(); hi.stop();

// Wait for child threads to terminate. try { hi.t.join(); lo.t.join();

}

catch (InterruptedException e) {

System.out.println("InterruptedException caught");

}

System.out.println("Low-priority thread: " + lo.click);

System.out.println("High-priority thread: " + hi.click);

}

}

Output:

Low-priority thread: 4408112

High-priority thread: 589626904

Notice that **running** is preceded by the keyword **volatile**. it is used here to ensure that the value of **running** is examined each time the loop iterates. Without the use of **volatile**, Java is free to optimize the loop in such a way that a local copy of **running** is created. The use of **volatile** prevents this optimization, telling Java that **running** may change in ways not directly apparent in the immediate code.

**Synchronization:**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*

Synchronization in Java can be achieved through two idioms: *synchronized methods* and *synchronized statement*.

**Synchronized Method:**

To protect a method from being access by more than one thread at the same time, precede the method’s definition with the keyword *synchronized*.

Syntax: *synchronized return\_type method\_name(parameter\_list)*

E.g.: In the following program multiple threads access a shared method (without synchronized method), which results in unpredicted output.

// This program is not synchronized.

class Callme { void call(String msg) {

System.out.print("[" + msg); try {

Thread.sleep(1000);

}

catch(InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println("]");

}

}

class Caller implements Runnable {

String msg;

Callme target; Thread t;

public Caller(Callme targ, String s) { target = targ; msg = s; t = new Thread(this);

t.start(); }

public void run() { target.call(msg);

}

}

class Synch { public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Hello");

Caller ob2 = new Caller(target, "Synchronized");

Caller ob3 = new Caller(target, "World");

// wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join();

}

catch(InterruptedException e) {

System.out.println("Interrupted");

}

}

}

Output:

Hello[Synchronized[World]

]

]

To fix the above program precede the method call()’s definition with the keyword **synchronized** as shown below:

synchronized void call()

{…

**Synchronized Statement:**

While creating a synchronized method within classes is an easy and effective way to achieve synchronization, it will not work in all cases. For example, to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by the programmer, but by a third party and the programmer do not have access to the source code. Thus, you can’t add **synchronized** to the appropriate methods within the class.

Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block. This is the general form of the **synchronized** statement:

synchronized(*object*) {

// statements to be synchronized }

Here, *object* is a reference to the object being synchronized.

The above program can be rewritten using synchronized blocks as follows:

// This program uses a synchronized block.

class Callme { void call(String msg) {

System.out.print("[" + msg); try {

Thread.sleep(1000);

}

catch (InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println("]");

}

}

class Caller implements Runnable {

String msg;

Callme target;

Thread t;

public Caller(Callme targ, String s) {

target = targ; msg = s;

t = new Thread(this);

t.start();

}

// synchronize calls to call() public void run() { synchronized(target) { // synchronized block target.call(msg);

}

}

}

class Synch1 { public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Hello");

Caller ob2 = new Caller(target, "Synchronized");

Caller ob3 = new Caller(target, "World");

// wait for threads to end try { ob1.t.join(); ob2.t.join(); ob3.t.join();

}

catch(InterruptedException e) {

System.out.println("Interrupted"); }

}

}

**Exception Handling:**

**Managing Errors and Exception:**

**Exception:**

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by the code.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**.

Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

This is the general form of an exception-handling block:

try {

// block of code to monitor for errors

}

catch (*ExceptionType1 exOb*) { // exception handler for *ExceptionType1*

}

catch (*ExceptionType2 exOb*) { // exception handler for *ExceptionType2*

}

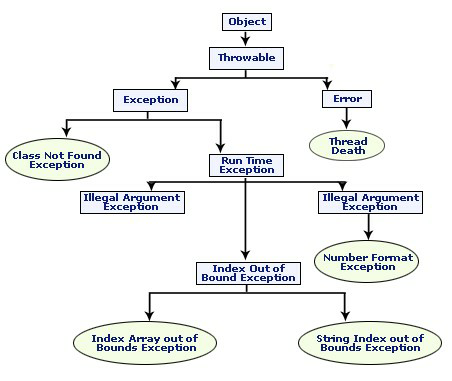
// ...

finally {

// block of code to be executed before try block ends }

**Types of Error (Exceptions):**

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.



The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

The most general of these exceptions are subclasses of the standard type

RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

In Java, there are two types of exception:

1. **Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

|  |  |  |
| --- | --- | --- |
| **Exception** | **Description** |  |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |
|  | | |

1. **Unchecked** are the exceptions that are not checked at compiled time. In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

Following is the list of Java Checked Exceptions Defined in java.lang.

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | | **Description** | |
| ClassNotFoundException | | Class not found. | |
| CloneNotSupportedException | | Attempt to clone an object that does not implement the Cloneable interface. | |
| IllegalAccessException | Access to a class is denied. | |  | |
| InstantiationException | Attempt to create an object of an abstract class or interface. | |
| InterruptedException | One thread has been interrupted by another thread. | |
| NoSuchFieldException | A requested field does not exist. | |
| NoSuchMethodException | A requested method does not exist. | |
|  |  | | | |

**Using try and catch:**

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch.

class Exc2 { public static void main(String args[]) { int d, a; try { // monitor a block of code.

d = 0; a = 42 / d;

System.out.println("This will not be printed.");

}

catch (ArithmeticException e) { // catch divide-by-zero error System.out.println("Division by zero.");

}

System.out.println("After catch statement.");

}

} Output:

This program generates the following output:

Division by zero.

After catch statement.

For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

// Handle an exception and move on. import java.util.Random; class HandleError { public static void main(String args[]) {

int a=0, b=0, c=0;

Random r = new Random(); for(int i=0; i<32000; i++) { try { b = r.nextInt(); c = r.nextInt(); a = 12345 / (b/c);

}

catch (ArithmeticException e) {

System.out.println("Division by zero."); a = 0; // set a to zero and continue }

System.out.println("a: " + a);

}

}

}

**Multiple catch:**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.

E.g.:

// Demonstrate multiple catch statements.

class MultiCatch { public static void main(String args[]) { try { int a = args.length;

System.out.println("a = " + a);

int b = 42 / a; int c[] = { 1 }; c[42] = 99;

}

catch(ArithmeticException e) {

System.out.println("Divide by 0: " + e);

}

catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Array index oob: " + e);

}

System.out.println("After try/catch blocks.");

}

}

**Using throw:**

In Java, it is possible for a program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferee.d to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

e.g.:

// Demonstrate throw.

class ThrowDemo { static void demoproc() { try { throw new NullPointerException("demo");

}

catch(NullPointerException e) {

System.out.println("Caught inside demoproc."); throw e; // rethrow the exception }

}

public static void main(String args[]) { try { demoproc();

}

catch(NullPointerException e) {

System.out.println("Recaught: " + e); }

}

}

Output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

**Using throws:**

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method’s declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a **throws** clause:

*type method-name(parameter-list)* throws *exception-list*

{

// body of method

}

Here, *exception*-*list* is a comma-separated list of the exceptions that a method can throw.

e.g.:

// This is now correct.

class ThrowsDemo { static void throwOne() throws IllegalAccessException { System.out.println("Inside throwOne."); throw new IllegalAccessException("demo");

}

public static void main(String args[]) { try { throwOne();

}

catch (IllegalAccessException e) {

System.out.println("Caught " + e); }

}

}

Output:

inside throwOne

caught java.lang.IllegalAccessException: demo

**Using finally:**

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then the programmer will not want the code that closes the file to be bypassed by the exceptionhandling mechanism. The **finally** keyword is designed to address this contingency.

**finally** creates a block of code that will be executed after a **try**/**catch** block has complete d

and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. The **finally** clause is optional.

e.g.:

// Demonstrate finally.

class FinallyDemo {

// Through an exception out of the method.

static void procA() { try {

System.out.println("inside procA");

throw new RuntimeException("demo");

} finally {

System.out.println("procA's finally"); }

}

// Return from within a try block.

static void procB() { try {

System.out.println("inside procB"); return;

} finally {

System.out.println("procB's finally"); }

}

// Execute a try block normally.

static void procC() { try {

System.out.println("inside procC");

} finally {

System.out.println("procC's finally"); }

}

public static void main(String args[]) { try { procA();

}

catch (Exception e) {

System.out.println("Exception caught");

} procB(); procC();

}

}

Output:

inside procA

procA’s finally Exception caught inside procB procB’s finally inside procC

procC’s finally

\*\*\*\*\*\*\*\*\*

Chapter 5: Java Applets and Graphics Programming

**5.1: Applet Programming:**

**Applet Basics:**

*Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document. After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Compiling an Applet program in done in the same way a normal java program is compiled.

However, running an applet involves a different process. There are two ways to run an applet:

* Executing an Applet within a Java-Compatible Web Browser.
* Using an applet viewer, such as the standard SDK tool, appletviewer. An appletviewer executes an applet in a window. This is generally the fastest and easiest way to run an applet.

**Local and Remote Applets:**

Web pages can contain two types of Applets:

* 1. Local Applet.
  2. Remote Applet.

**Local Applets:**

A local applet is the one that is stored on the client machine itself. When the web page must find a local applet, it does not need to retrieve the information from the Internet; in fact the browser doesn’t even need to be connected to the Internet at that time. A local applet is specified by a path name and a file name. E.g.:

<applet codebase=”tictactoe” code=”tictactoe.class” width=120 height=120>

</applet>

**Remote Applet:**

A remote applet is the one that is located on another computer system (remote system). This system may be located in the building next door or it may be on the other side of the world, it makes no difference to the java compatible browser. No matter where the remote applet is located, it is downloaded onto the client’s computer via Internet. The browser must of course be connected to the Internet at the time it needs to display the remote applet.

To reference a remote applet in the web page, the programmer must know the applet’s URL, and any attributes and parameters that is needed to be supplied in order to display the applet correctly. E.g.:

<applet codebase=”[http://www.myconnect.com/applets/”](http://www.myconnect.com/applets/)

Code=”tictactoe.class”

Width=120

Height=120>

</applet>

**How Applet differ from Application:**

|  |  |
| --- | --- |
| Applet | Application |
| They are small programs. | They are large programs. |
| They are executed on client browser. | They are executed on standalone computer system. |
| It is portable and can be executed on any java supported web browser. | Need JDK, JRE, JVM installed on client machine. |
| Applet applications are executed in restricted environment. | Application can access all the resources of the computer system. |
| Applets are created by extending the Applet class. | Applications are created by writing P.S.V.M. method. |

Applet applications have 5 methods which will Application has single start point, which main be automatically invoked on occurrence of method.

specific event.

Exa

mple:

import java.awt.\*;

import java.applet.\*;

public class Myclass extends Applet

{

public void init() { }

public void start() { }

public void stop() {}

public void destroy() {}

public void paint(Graphics g) {}

}

Example:

public class MyClass

{

public static void main(String args[]) {}

}

**Preparing to write applet code:**

A simple applet program is shown below:

import java.awt.\*;

import java.applet.\*;

public class SimpleApplet extends Applet { public void paint(Graphics g) {

g.drawString("A Simple Applet", 20, 20); }

}

The applet begins with two important statements. The first imports the Abstract Window

Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the consolebased I/O classes. The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that is created must be a subclass of **Applet**.

The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program. Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

*void drawString(String message, int x, int y)*

Here, *message* is the string to be output beginning at *x*,*y.* In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.

After entering the source code for **SimpleApplet**, it is compiled in the same way similar to normal java applications.

Now to execute the applet in a web browser, a short HTML text file is needed to be written that contains the appropriate APPLET tag. The HTML file is shown below:

<applet code="SimpleApplet" width=200 height=60> </applet>

The **width** and **height** statements specify the dimensions of the display area used by the applet. After creating the HTML file, the browser can be executed and then load this file, which causes **SimpleApplet** to be executed.

To execute **SimpleApplet** with an applet viewer, Simply include a comment at the head of the Java source code file that contains the APPLET tag. By doing so, the code is documented with a prototype of the necessary HTML statements, and the compiled applet can be tested merely by starting the applet viewer with your Java source code file. If this method is used, the **SimpleApplet** source file looks like this: import java.awt.\*;

import java.applet.\*; /\*

<applet code="SimpleApplet" width=200 height=60> </applet>

\*/

public class SimpleApplet extends Applet { public void paint(Graphics g) {

g.drawString("A Simple Applet", 20, 20); }

}

Output:



■ Applets do not need a **main( )** method.

■ Applets must be run under an applet viewer or a Java-compatible browser.

■ User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by the AWT.

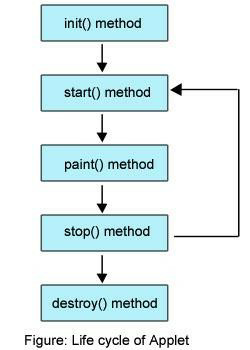
**Applet Life Cycle:**

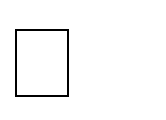
Various states, an applet, undergoes between its object creation and object removal (when the job is over) is known as **life cycle**. Each state is represented by a method. There exist 5 states represented by 5 methods. That is, in its life of execution, the applet exists (lives) in one of these 5 states. These methods are known as "callback methods" as they are called automatically by the browser whenever required for the smooth execution of the applet. Programmers just write the methods with some code but never call.

Following are the methods:

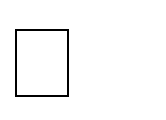
1. init()
2. start()
3. paint()
4. stop()
5. destroy()

These methods are known as **life cycle methods**. These methods are defined in **java.applet. Applet** class except **paint()** method. The paint() method is defined in **java.awt.Component** class, an indirect super class of Applet. Even though, the methods are called automatically by the browser, the programmer should know well when they are called.

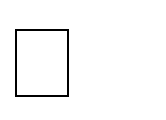


**init():** The applet's voyage starts here. In this method, the applet object is created by the 

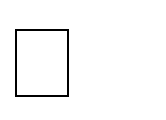
browser. Because this method is called before all the other methods, programmer can utilize this method to instantiate objects, initialize variables, setting background and foreground colors in GUI etc.; the place of a constructor in an application. It is equivalent to [**born state**](http://way2java.com/multithreading/life-cycle-of-thread/) of a thread.

**start():** In init() method, even through applet object is created, it is in **inactive** state. An 

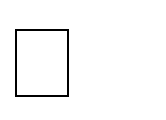
inactive applet is not eligible for microprocessor time even though the microprocessor is idle. To make the applet active, the init() method calls start() method. In start() method, applet becomes active and thereby eligible for processor time.

**paint():** This method takes a **java.awt.Graphics** object as parameter. This class includes many 

methods of drawing necessary to draw on the applet window. This is the place where the programmer can write his code of what he expects from applet like animation etc. This is equivalent to [**runnable state**](http://way2java.com/multithreading/life-cycle-of-thread/) of thread.

**stop():** In this method the applet becomes temporarily inactive. An applet can come any 

number of times into this method in its life cycle and can go back to the active state (paint() method) whenever would like. It is the best place to have cleanup code. It is equivalent to the [**blocked state**](http://way2java.com/multithreading/life-cycle-of-thread/) of the thread.

**destroy():** This method is called just before an applet object is garbage collected. This is the 

end of the life cycle of applet. It is the best place to have cleanup code. It is equivalent to the [**dead state**](http://way2java.com/multithreading/life-cycle-of-thread/) of the thread.

E.g.: A simple applet program that sets the background color, foreground color and outputs a string.

import java.awt.\*; import java.applet.\*;

/\*

<applet code="Sample" width=300 height=50>

</applet> \*/

public class Sample extends Applet{

String msg;

// set the foreground and background colors.

public void init() { setBackground(Color.cyan); setForeground(Color.red); msg = "Inside init( ) --";

}

// Initialize the string to be displayed. public void start() { msg += " Inside start( ) --";

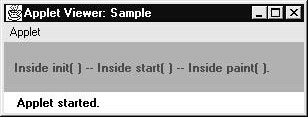
}

// Display msg in applet window. public void paint(Graphics g) { msg += " Inside paint( ).";

g.drawString(msg, 10, 30); }

}

Output:



**Applet Tag:**

The APPLET tag is used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page. The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

< APPLET

[CODEBASE = *codebaseURL*]

CODE = *appletFile*

[ALT = *alternateText*]

[NAME = *appletInstanceName*]

WIDTH = *pixels* HEIGHT = *pixels*

[ALIGN = *alignment*]

[VSPACE = *pixels*] [HSPACE = *pixels*]

>

[< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>] [< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]

. . .

[*HTML Displayed in the absence of Java*]

</APPLET>

**CODEBASE:** CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet’s executable

class file. The HTML document’s URL directory is used as the CODEBASE if this attribute is not specified.

**CODE:** CODE is a required attribute that gives the name of the file containing your applet’s compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

**ALT:** The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can’t currently run Java applets.

**NAME:** NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

**WIDTH AND HEIGHT:** WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

**ALIGN:** ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

**VSPACE AND HSPACE:** These attributes are optional*.* VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet.

**PARAM NAME AND VALUE:** The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

**Passing Parameter to the Applet:**

The APPLET tag in HTML allows passing parameters to the applet. To retrieve a parameter, the **getParameter( )** method is used. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **Boolean** values need to be converted their string representations into their internal formats. E.g.:

// Use Parameters

import java.awt.\*; import java.applet.\*;

/\*

<applet code="ParamDemo" width=300 height=80>

<param name=fontName value=Courier>

<param name=fontSize value=14>

<param name=leading value=2>

<param name=accountEnabled value=true>

</applet>

\*/

public class ParamDemo extends Applet{

String fontName; int fontSize; float leading; boolean active;

// Initialize the string to be displayed.

public void start() { String param;

fontName = getParameter("fontName");

if(fontName == null)

fontName = "Not Found";

param = getParameter("fontSize"); try { if(param != null) // if not found fontSize = Integer.parseInt(param); else fontSize = 0;

}

catch(NumberFormatException e) { fontSize = -1;

}

param = getParameter("leading"); try { if(param != null) // if not found leading = Float.valueOf(param).floatValue();

else leading = 0;

}

catch(NumberFormatException e) { leading = -1;

}

param = getParameter("accountEnabled");

if(param != null) active = Boolean.valueOf(param).booleanValue();

}

// Display parameters.

public void paint(Graphics g) {

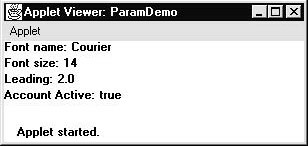
g.drawString("Font name: " + fontName, 0, 10);

g.drawString("Font size: " + fontSize, 0, 26);

g.drawString("Leading: " + leading, 0, 42);

g.drawString("Account Active: " + active, 0, 58); }

} Output:



**Graphics Programming:**

**The Graphics Class:**

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

■ It is passed to an applet when one of its various methods, such as **paint( )** or **update( )**, is called.

■ It is returned by the **getGraphics( )** method of **Component**.

The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped.

**Lines and Rectangles:**

**Drawing Lines:**

Lines are drawn by means of the **drawLine( )** method, shown below:

*void drawLine(int startX, int startY, int endX, int endY)*

**drawLine( )** displays a line in the current drawing color that begins at *startX*,*startY* and ends at *endX*,*endY*.

E.g.:

// Draw lines

import java.awt.\*; import java.applet.\*;

/\*

<applet code="Lines" width=300 height=200> </applet>

\*/

public class Lines extends Applet { public void paint(Graphics g) {

g.drawLine(0, 0, 100, 100);

g.drawLine(0, 100, 100, 0);

g.drawLine(40, 25, 250, 180);

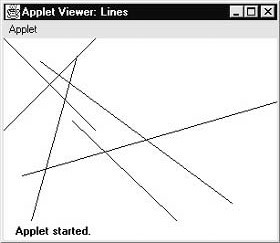
g.drawLine(75, 90, 400, 400);

g.drawLine(20, 150, 400, 40);

g.drawLine(5, 290, 80, 19); }

}

Output:



**Drawing Rectangles:**

The **drawRect( )** and **fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown below:

*void drawRect(int top, int left, int width, int height) void fillRect(int top, int left, int width, int height)*

The upper-left corner of the rectangle is at *top*,*left.* The dimensions of the rectangle are specified by *width* and *height.*

To draw a rounded rectangle, use **drawRoundRect( )** or **fillRoundRect( )**, both shown here:

*void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)*

*void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)*

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top*,*left.* The dimensions of the rectangle are specified by *width* and *height.* The diameter of the rounding arc along the X axis is specified by *xDiam.* The diameter of the rounding arc along the Y axis is specified by *yDiam.* // Draw rectangles import java.awt.\*; import java.applet.\*;

/\*

<applet code="Rectangles" width=300 height=200> </applet>

\*/

public class Rectangles extends Applet { public void paint(Graphics g) {

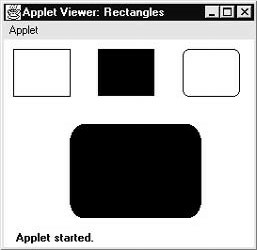
g.drawRect(10, 10, 60, 50);

g.fillRect(100, 10, 60, 50);

g.drawRoundRect(190, 10, 60, 50, 15, 15);

g.fillRoundRect(70, 90, 140, 100, 30, 40); }

} Output:



**Drawing Ellipses and Circles:**

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown below:

*void drawOval(int top, int left, int width, int height) void fillOval(int top, int left, int width, int height)*

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height.* To draw a circle, specify a square as the bounding rectangle. E.g.:

// Draw Ellipses import java.awt.\*; import java.applet.\*;

/\*

<applet code="Ellipses" width=300 height=200> </applet>

\*/

public class Ellipses extends Applet { public void paint(Graphics g) {

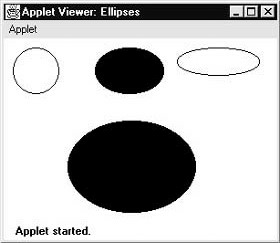
g.drawOval(10, 10, 50, 50);

g.fillOval(100, 10, 75, 50);

g.drawOval(190, 10, 90, 30);

g.fillOval(70, 90, 140, 100); }

}



**Drawing Arcs:**

Arcs can be drawn with **drawArc( )** and **fillArc( )**, shown below:

*void drawArc(int top, int left, int width, int height, int startAngle,int sweepAngle) void fillArc(int top, int left, int width, int height, int startAngle,int sweepAngle)*

The arc is bounded by the rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height.* The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle.* Angles are specified in degrees. Zero degrees is on the horizontal, at the three o’clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. E.g.:

// Draw Arcs import java.awt.\*; import java.applet.\*;

/\*

<applet code="Arcs" width=300 height=200> </applet>

\*/

public class Arcs extends Applet { public void paint(Graphics g) {

g.drawArc(10, 40, 70, 70, 0, 75);

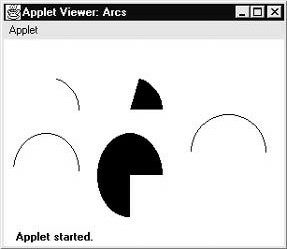
g.fillArc(100, 40, 70, 70, 0, 75);

g.drawArc(10, 100, 70, 80, 0, 175);

g.fillArc(100, 100, 70, 90, 0, 270);

g.drawArc(200, 80, 80, 80, 0, 180); }

}



**Drawing Polygons:**

It is possible to draw arbitrarily shaped figures using **drawPolygon( )** and **fillPolygon( )**, shown below:

*void drawPolygon(int x[ ], int y[ ], int numPoints) void fillPolygon(int x[ ], int y[ ], int numPoints)*

The polygon’s endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by *x* and *y* is specified by *numPoints.* There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.

// Draw Polygon import java.awt.\*; import java.applet.\*;

/\*

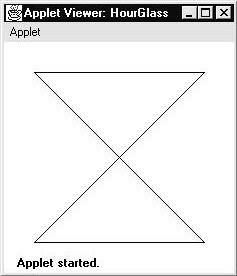
<applet code="HourGlass" width=230 height=210> </applet>

\*/

public class HourGlass extends Applet { public void paint(Graphics g) { int xpoints[] = {30, 200, 30, 200, 30}; int ypoints[] = {30, 30, 200, 200, 30}; int num = 5;

g.drawPolygon(xpoints, ypoints, num); }

}



Chapter 6: Streams and File I/O

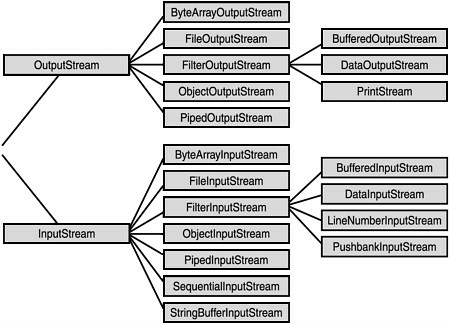
**6.1: Stream Classes**

A stream can be defined as sequence of data. Java’s stream based I/O is built upon four abstract classes: **InputStream, OutputStream, Reader and Writer. InputStream** and

**OutputStream** are designed for byte streams. The InputStream is used to read data from a source and OutputStream is used to write data to a destination. **Reader** and **Writer** are designed for character stream. One should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

**6.2: Byte Streams**

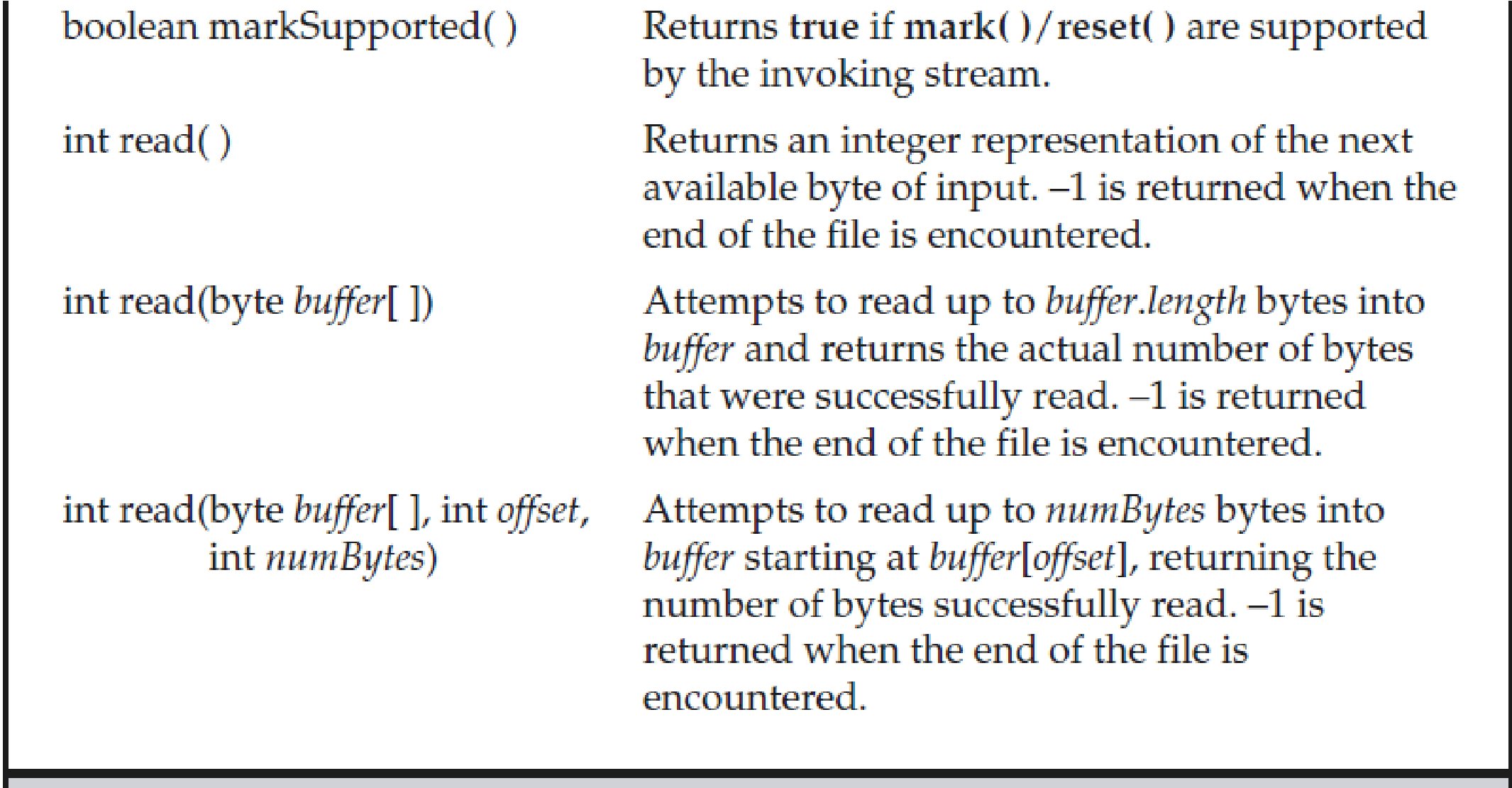
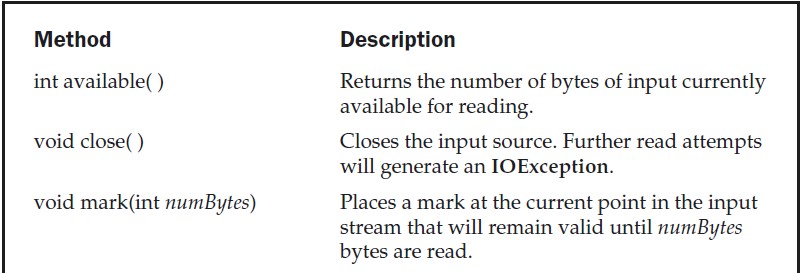
The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. The byte stream classes are topped by **InputStream** and **OutputStream**.

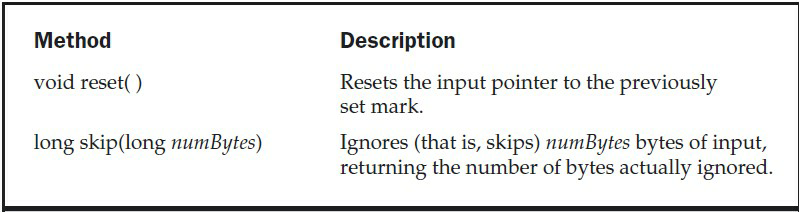


**InputStream:**

**InputStream** is an abstract class that defines Java’s model of streaming byte input. All of the methods in this class will throw an **IOException** on error conditions.

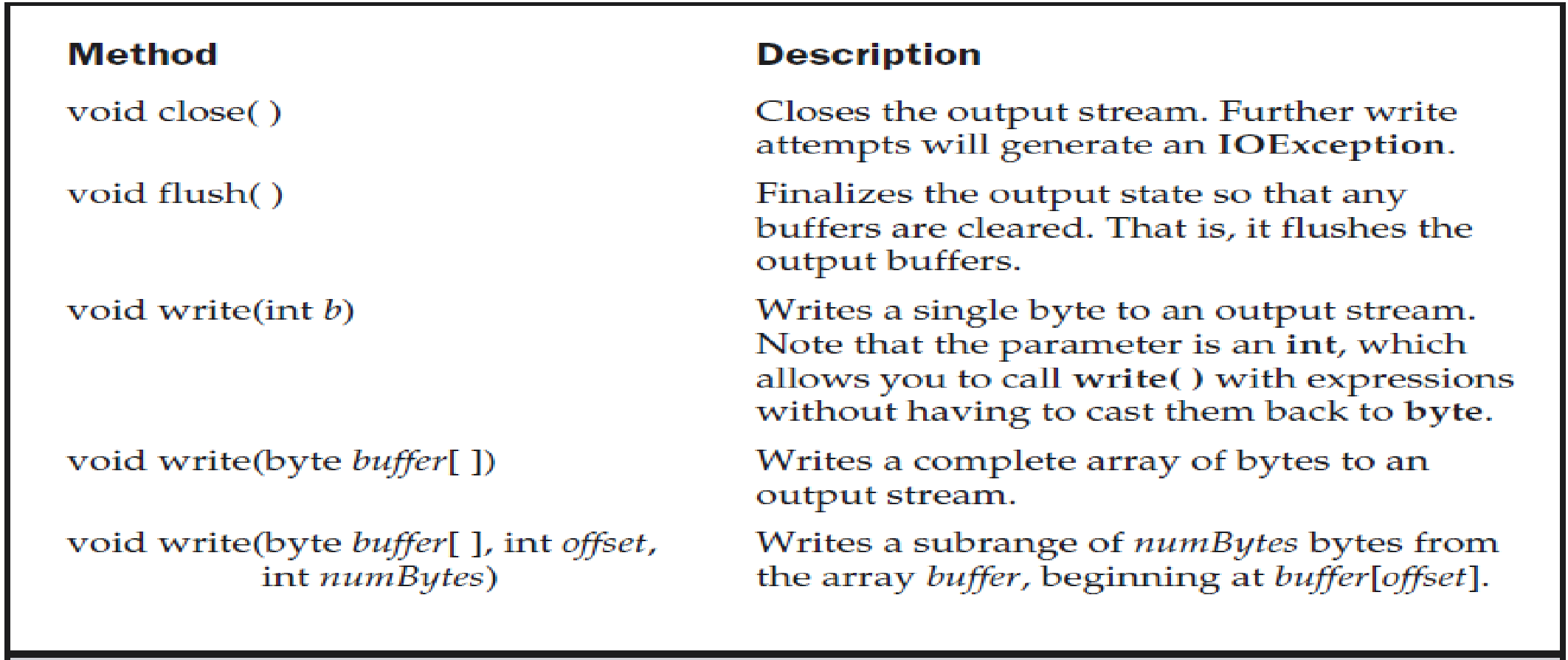
Following are the methods of InputStream:





**OutputStream:**

**OutputStream** is an abstract class that defines streaming byte output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors. Following are the methods in OutputStream:



**FileInputStream:**

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown below:

FileInputStream(String filepath)

FileInputStream(File fileObj)

Either can throw a **FileNotFoundException**. Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

E.g.: FileInputStream f0 = new FileInputStream("/autoexec.bat")

File f = new File("/autoexec.bat");

FileInputStream f1 = new FileInputStream(f);

**FileOutputStream:**

FileOutputStream creates an OutputStream that you can use to write bytes to a file. Its most commonly used constructors are shown below:

FileOutputStream(String *filePath*)

FileOutputStream(File *fileObj*)

FileOutputStream(String *filePath*, boolean *append*)

FileOutputStream(File *fileObj*, boolean *append*)

They can throw a **FileNotFoundException** or a **SecurityException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode. The fourth constructor was added by Java 2, version 1.4.

Creation of a **FileOutputStream** is not dependent on the file already existing.

**FileOutputStream** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

E.g.:

import java.io.\*; public class fileStreamTest{ public static void main(String args[]){

try{

byte bWrite [] = {11,21,3,40,5};

OutputStream os = new FileOutputStream("test.txt"); for(int x=0; x < bWrite.length ; x++){ os.write( bWrite[x] ); // writes the bytes

}

os.close();

InputStream is = new FileInputStream("test.txt");

int size = is.available(); for(int i=0; i< size; i++){

System.out.print((char)is.read() + " ");

}

is.close();

}

catch(IOException e){

System.out.print("Exception");

}

}

}

**The Character Streams:**

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.

**FileReader:**

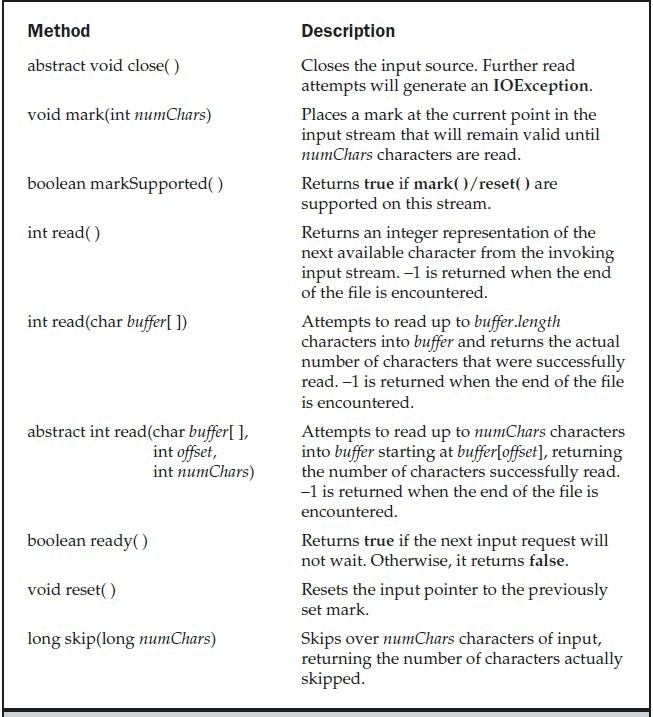
The FileReader creates a **Reader** that can be used to read the content of a file. Following are the two most commonly used constructors for this class:

FileReader(String *filePath*)

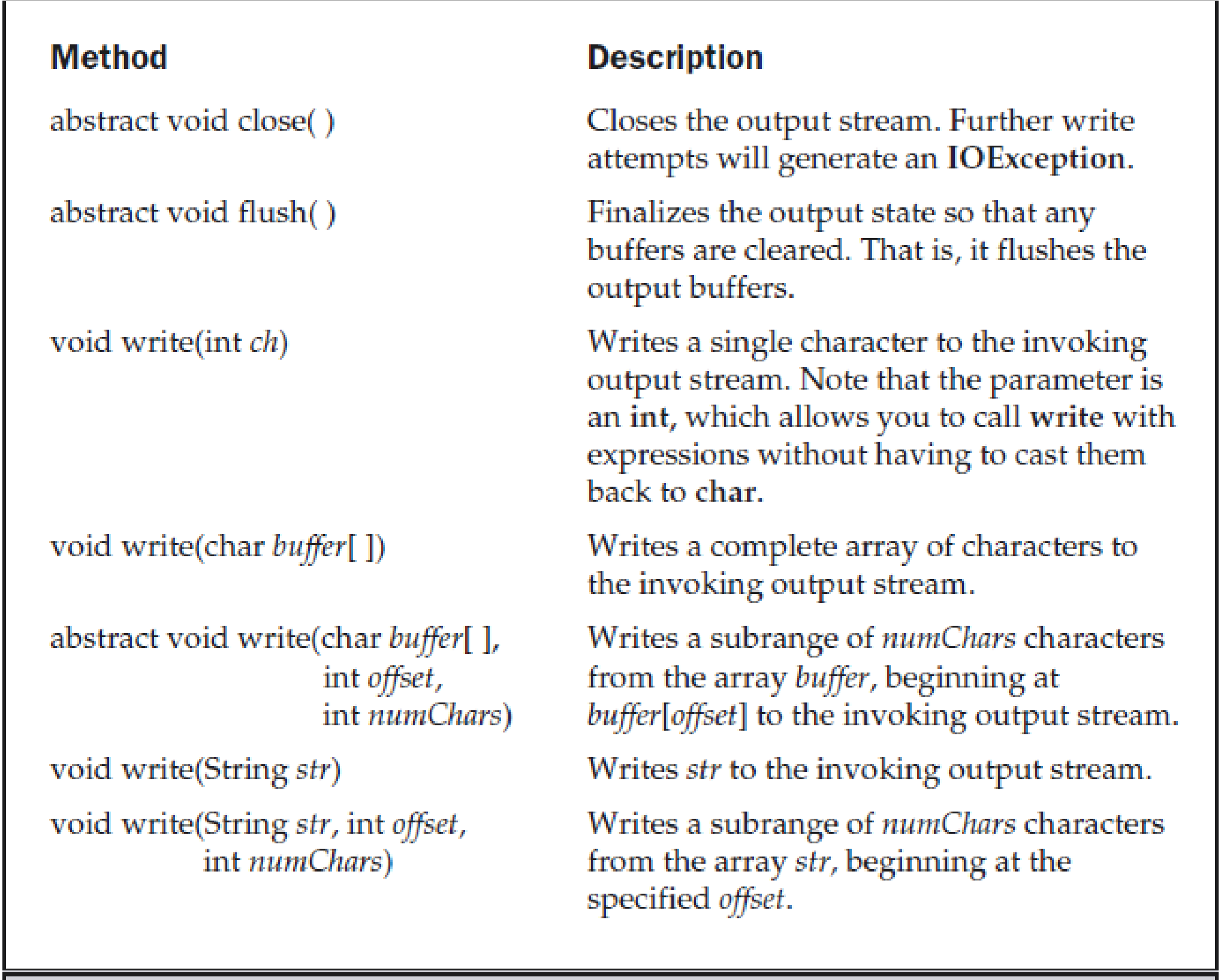
FileReader(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

Following are the methods defined by Reader class:



Following are the methods defined by Writer class:



The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory.

// Demonstrate FileReader. import java.io.\*; class FileReaderDemo { public static void main(String args[]) throws Exception {

FileReader fr = new FileReader("FileReaderDemo.java");

BufferedReader br = new BufferedReader(fr);

String s;

while((s = br.readLine()) != null) { System.out.println(s);

}

fr.close();

}

}

**FileWriter:**

**FileWriter** creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String *filePath*)

FileWriter(String *filePath,* boolean *append)*

FileWriter(File *fileObj*)

FileWriter(File *fileObj*, boolean *append*)

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file. The fourth constructor was added by Java 2, version 1.4. Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output

when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

E.g.:

// Demonstrate FileWriter. import java.io.\*; class FileWriterDemo { public static void main(String args[]) throws Exception {

String source = "Now is the time for all good men\n" + " to come to the aid of their country\n" + " and pay their due taxes."; char buffer[] = new char[source.length()]; source.getChars(0, source.length(), buffer, 0); FileWriter f0 = new FileWriter("file1.txt"); for (int i=0; i < buffer.length; i += 2) { f0.write(buffer[i]);

}

f0.close();

FileWriter f1 = new FileWriter("file2.txt"); f1.write(buffer); f1.close();

}

}

**Serialization:**

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

**Serializable:**

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.