UNIT I
INTRODUCTION TO JAVA

THE HISTORY AND EVOLUTION OF JAVA

JAVA'S LINEAGE

Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past three decades. For these reasons, this section reviews the sequence of events and forces that led up to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

THE CREATION OF JAVA

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak" but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

HOW JAVA CHANGED THE INTERNET

The Internet helped catapult Java to the forefront of programming, and Java, in turn,has had a profound effect on the Internet. The reason for this is quite simple: Java expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. However, a second type of object can be transmitted to your computer: a dynamic, self-executing program. Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly the data that the server is sending. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to Java, cyberspace was effectively closed to half the entities that now live there. As you will see, Java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet.

JAVA'S MAGIC: THE BYTECODE

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine* (*JVM*). That is, in its standard form, the JVM is an *interpreter for bytecode.* This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted—mostly because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why. 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. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.

The fact that a Java program is interpreted also helps to make it secure. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect. Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your Java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same.

SERVLETS: JAVA ON THE SERVER SIDE

Java expands the universe of objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. However, a second

type of object can be transmitted to your computer: a dynamic, self-executing program. Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly the data that the server is sending. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to Java, cyberspace was effectively closed to half the entities that now live there. As you will see, Java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet.

JAVA FEATURES OR JAVA BUZZWORDS

Following are the features or buzzwords of Java language which made it popular:

1.Simple

2.Secure

3.Portable

4.Object-Oriented

5.Robust

6.Multithreaded

7.Architecture neutral

8.Interpreted

9.High Performance

10.Distributed

11.Dynamic

Simple:

- Java easy to learn
- It is easy to write programs using Java
- Expressiveness is more in Java.
- Most of the complex or confusing features in C++ are removed in Java like pointers etc..

Secure:

- Java provides data security through encapsulation.
- Also we can write applets in Java which provides security.
- An applet is a small program which can be downloaded from one computer to another automatically.
- There is no need to worry about applets accessing the system resources which may compromise security.
- Applets are run within the JVM which protects from unauthorized or illegal access to system resources.

Portable:

- Applications written using Java are portable in the sense that they can be executed on any kind of computer containing any CPU or any operating system.
- When an application written in Java is compiled, it generates an intermediate code file called as “bytecode”.
- Bytecode helps Java to achieve portability.
- This bytecode can be taken to any computer and executed directly.

Object - Oriented:

•Java follows object oriented model.

•So, it supports all the features of object oriented model like:

Encapsulation

Inheritance

Polymorphism

Abstraction

Robust:

•A program or an application is said to be robust(reliable) when it is able to give some response in any kind of context.

•Java's features help to make the programs robust. Some of those features are:

1. Type checking

2. Exception handling

Multithreaded:

•Java supports multithreading which is not supported by C and C++.

•A thread is a light weight process.

•Multithreading increases CPU efficiency.

•A program can be divided into several threads and each thread can be executed concurrently or in parallel with the other threads.

•Real world example for multithreading is computer. While we are listening to music, at the same time we can write in a word document or play a game.

Architecture - Neutral:

•Byte code helps Java to achieve portability.

•Byte code can be executed on computers having any kind of operating system or any kind of CPU.

•Since Java applications can run on any kind of CPU, Java is architecture – neutral.

Interpreted and High Performance:

•In Java 1.0 version there is an interpreter for executing the byte code. As interpreter is quite slow when compared to a compiler, java programs used to execute slowly.

•After Java 1.0 version the interpreter was replaced with JIT(Just-In-Time) compiler.

•JIT compiler uses Sun Micro system's Hot Spot technology.

•JIT compiler converts the byte code into machine code piece by piece and caches them for future use.

•This enhances the program performance means it executes rapidly.

Distributed:

•Java supports distributed computation using Remote Method Invocation (RMI) concept.

•The server and client(s) can communicate with another and the computations can be divided among several computers which makes the programs to execute rapidly.

• In distributed systems, resources are shared.

Dynamic:

• The Java Virtual Machine(JVM) maintains a lot of runtime information about the program and the objects in the program.

• Libraries are dynamically linked during runtime.

• So, even if you make dynamic changes to pieces of code, the program is not effected.

THE JAVA KEYWORDS:

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords const and goto are reserved, even though they are not currently used. true, false, and null might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

1. abstract
2. assert
3. boolean
4. break
5. byte
6. case
7. catch
8. char
9. class
10. const (not used)
11. continue
12. default
13. do
14. double
15. else
16. enum
17. extends
18. final
19. finally
20. float
21. for
22. goto (not used)
23. if
24. implements
25. import
26. instanceof
27. int
28. interface
29. long
30. native
31. new
32. package
33. private
34. protected
35. public
36. return
37. short
38. static
39. strictfp
40. super
41. switch
42. synchronized
43. this
44. throw
45. throws
46. transient
47. try
48. void
49. volatile
50. while

1. abstract: -

Java provides abstract keyword to signify something empty.

By empty we mean that it has no definition or implementation. Keyword abstract is used before the classes and methods. At the time of declaring a class we mark it as abstract. An abstract class cannot be instantiated. It means an abstract class cannot have Objects. By abstract method, we mean that method doesn't have any body. An abstract method doesn't have any statements to execute. Classes that contain abstract method must also be assigned as abstract class and implementation to their abstract method must be provided by extending those classes. If subclasses fail to provide implementation to super-class abstract methods, than they are also marked as abstract.

2. assert: -

Java provides us with a debugging keyword called as assert. In Java, assert is a normal Java statement which a programmer puts in the code, that he/she thinks will always be true. Generally, assert statements are being used by the programmer for the debugging purpose. Assertion statements can be turned on or off. While running a program if we enable assertion then wherever our assert statement gets failed Assertion Error is thrown by the program and whole program gets terminated. Generally, assertions are placed by programmer in the code where he believes that statement will be true. After enabling assertion if there is a failure, programmer gets an idea that in code there is some bug. The code fails with an Assertion Error.

3. boolean: -

A boolean keyword in Java is a data type for a variable which can store two values, true or false. Generally, boolean keyword is a data type which has size of 1 bit. In Java I am not sure about the size that whether it store 1 bit or not.

4. break: -

In Java, 'break' keyword is of quite high importance. As the name suggest, break statement breaks code from one point and transfer flow to other part of program to execute. Generally let say in if block you have placed a break statement. If your code enters if statement an encounters break statement it will stop executing further if block and come out of if block and continue to execute.

5. byte :-

In Java, byte keyword is used as a data type for storing 8 bits of information for an integer. If it's used before a method definition than that method, when called will always return a byte value.

6. case :-

In Java, case keyword is used within switch statements. A condition in switch statement is compared with the cases in switch statement. Whatever case matches with the switch expression that case is executed.

7. catch :-

In Java, catch keyword has a group of statements that catches exception, if there is any exception in try block proceeding to it. The statements have a way to deal with the exception or have a way to let

programmer know that something is needed to be correct.

8. char :-

In Java, char keyword is used as a data type which can store 16 bit Unicode character. If the keyword is placed before method declaration than the method upon execution returns a value which is a char.

9. class :-

In Java, class keyword is used to define a class. Generally a class acts as a blue print for an object. It defines implementation for the object. It has statements defining variables, methods, inner classes etc. An Object when instantiated for a particular class has all the physical implementation of what is defined in the class.

10. const :-

It is a reserved keyword in Java but it has no use in Java and Java has provided it no function to perform.

11. continue :-

In Java, the continue keyword is used when we want rest of the statement after continue keyword to get skipped and want program to continue with the next iteration.

12. default :-

In Java, default keyword is used within switch statement. It is used their optionally, if no Java case matches with the expression than the default case is executed by the program.

13. do :-

In Java, do keyword is used to implement do-while loop. Generally, do keyword is used when we want to loop a block of statement once. After that the boolean condition gets evaluated, if condition is yes the loop execute again, but if condition comes out to be false the loop exits.

14. double :-

In Java, double keyword is used as a data type for storing 64 bits of information for a float type. If it's used before a method definition than that method when called will always return a double value.

15. else :-

In Java, else keyword is used along with if keyword to create an if-else statement. Generally, a condition is evaluated in if block brackets. If the condition evaluates to true if block body gets executed. If the condition is evaluated to false else block body gets executed.

16. enum :-

It is used in Java language to declare a data type consisting of a set of named values. They are constants.

17. extends :-

In Java, extends keyword is used specify the super class of a subclass, also in interface declaration to specify one or more super interfaces. If we take an example say let say class A extends class B, here class A adds functionality to class B by adding fields or methods. It goes same with interfaces.

18. final :-

Generally, final keyword is used to make elements constant. That is once assigned cannot be changed. A final class cannot be sub-classed. A final variable cannot be assigned a new value after it has been assigned to a value. A final method cannot be overridden.

19. finally :-

In Java, finally keyword is used to define a block of statements after try to catch blocks. This block executes after try block, or after catch block, or before any return statement.

20. float :-

In Java, float keyword is used as a data type for storing 32 bits of information for a float type. If it's used before a method definition than that method when called will always return a float value.

21. for :-

In Java, for keyword is used to create looping statement that is for loop. In this for loop a counter variable is initialized, than a boolean condition is evaluated and compared with counter variable or any expression which turns out to either true or false. If the condition comes out to be true a block of statements following for keyword executes, if condition comes out to be false for loop terminates.

22. goto :-

In Java, a goto keyword is a reserved keyword which is not used in the Java language and has no function provided to it.

23. if :-

In Java, if keyword is used (optionally) along with else keyword to create an if-else statement. Generally, a condition is evaluated in if block brackets. If the condition evaluates to true if block body gets executed. If the condition is evaluated to false else block body gets executed.

24. implements :-

In Java, implements keywords is used in class declaration. Generally, implements keyword implements functionality of interfaces. Interfaces are abstract in nature; their functionality is implemented in the class which implements it.

25. import :-

In Java, import statement is used at the start of a Java file, just after package declaration. Generally, import statement gets those classes in a program whose functionality we want to use in the program. It is also used in importing static members.

26. instanceof :-

In Java, instanceof operator is a binary operator which is used to test an IS-A relationship between a class and object. Object is first operand and Class or Interface is a second operand.

27. int :-

In Java, int keyword is used as a data type for storing 32 bits of information for an integer type. If it's used before a method definition than that method when called will always return an int value.

28. interface :-

In Java, interface is a special type of structure which contains abstract methods, constant fields which are generally static or final.

29. long :-

In Java, long keyword is used as a data type for storing 64 bits of information for an integer type. If it's used before a method definition than that method when called will always return a long value.

30. native :-

This keyword is generally used in declaring method signifying that method's implementation is not in the same source file but in different source file and in different language too.

31. new :-

Java has a keyword called as new, which has been used to create a new instance of a class on heap. It is used is object creation and providing memory to it.

32. package :-

Packages have been created by the package keyword. In the package classes are kept which constitute a relation between each other.

33. private :-

In Java, private keyword can be used in declaration of methods, fields, inner class. This makes the methods, fields and inner class access to own class members.

34. protected :-

In Java, protected keyword can be used in declaration of methods, fields, inner class. This makes the methods, fields and inner class access to own class members, to sub class and classes of same package.

35. public :-

In Java, public keyword can be used before class, methods, fields. This makes the class , methods and fields accessible from any other class.

36. return :-

In Java, return keyword is used to pass a value from a method to caller method. It also signifies end of execution of a method.

37. short :-

In Java, short keyword is mainly used when we want to declare a field that can hold 16 bit of integer data. It is also placed before method declaration which signifies that method will return only short integer.

38. static :-

In Java, static keyword is used when you want to declare method, fields and inner class as class members rather object members, generally static members belong to a class rather than instance of that class. Therefore only one copy is made for them and used by the class as its own implementation.

39. strictfp :-

To ensure portability of floating point we ensure that they are platform independent.

40. super :-

In Java, super keywords is used to access overridden methods and hidden members of super class by its subclass. It is also used in constructor of a subclass to pass control over super class constructor.

41. switch :-

In Java switch keyword is used to have multi-decision making statements. Generally switch is mostly used with keywords case, break and default. This decision making block evaluates an expression first. After the expression is evaluated the value is compared with various case statements. Any value that matches with the expression, case associated with it gets executed. If no expression matches the default case gets executed.

42. synchronized :-

In java, synchronized keyword is used before the method declaration and before block of code to get mutual lock for an object. It means that the method or block of code gets locked by the object till it gets fully unlocked or gets fully executed by that object. Generally classes, fields and interfaces cannot be declared as synchronized. The static method gets code synchronization by class itself.

43. this :-

In Java, this keyword is used as a reference to currently executable object. Generally if we want to access class members we can use this keyword to access them. If we want to refer to a current object we use this keyword. We also use this keyword to transfer control from one constructor to another in the same class.

44. throw :-

In Java, when we want to throw a declared exception, the execution points to the catch block which has provided statements to overcome the exception. If no catch block is declared in the current method than the exception is passed to the calling method. It continues till it found exception handler. If no exception handler is found over the stack that exception is then transferred to the Uncaught Exception handler.

45. throws :-

In Java, we use throws keyword in the method declarations. If we get an exception in a method and method has no implementation for that exception, than method throws an exception. The exception thrown has been handled by the method who has called to current method in execution.

46. transient :-

Whenever we declare a variable as transient, that variable is not the part of Object serialization. When an object gets saved through default serialization all the non-transient variable retain their value after deserialization. If you want the variable should have default value after deserialization than make it transient.

47. try :-

In Java, whenever we want to do exception handling we use try keyword. Generally try block is used where we are sure that exception can occur. In try block statements are executed normally and as soon as exception occur it is handled my catch keyword following try block. It must have at least one catch or finally block.

48. void :-

In Java, when we use void before any method declaration, we make sure that method doesn't return any value.

49. volatile :-

In Java, volatile keyword provides a lock free mechanism. If a variable is assigned as volatile than all the threads accessing it can modify its current value without keeping a separate copy. A volatile keyword is accessed by all threads simultaneously. They operate on current value of variable rather than cached value.

50. while :-

In Java, while keyword is used to create looping statement that is while loop. In this while loop a boolean condition is evaluated and compared with counter variable or any expression which turns out to either true or false. If the condition comes out to be true a block of statements following while keyword executes, if condition comes out to be false the while loop terminates.

51. false :-

In Java, false keyword is the literal for the data type Boolean. Expressions are compared by this literal value.

52. null :-

In Java, null keyword is the literal for the reference variable. Expressions are compared by this literal value. It is a reserved keyword.

53. true :-

In Java, true keyword is the literal for the data type Boolean. Expressions are compared by this literal value. - See more at: http://www.hubberspot.com/2012/03/top-50-java-keywords-complete-reference.html#sthash.Fwpuhrxd.dpuf

EVOLUTION OF JAVA

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial

than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled by applets, and reconfigured many features of the 1.0

library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added and subtracted attributes from its original specification.

AN OVERVIEW OF JAVA

THE KEY ATTRIBUTES OF OBJECT ORIENTED PROGRAMMING:

- Object
- Class
- Data Abstraction and Encapsulation
- Dynamic Binding
- Message Passing
- Inheritance
- Polymorphism

Object:

Object is a collection of number of entities. Object take up space in the memory. Objects are instances of classes. When a program is executed, the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having no details of each other's data or code.

Class:

Class is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.

Ex: banana, orange are the objects of class Fruit.

Fruit mango;

By above mango object is created for class Fruit.

Data Abstraction and Encapsulation:

Combining data and functions into a single unit called class and the process is known as Encapsulation. Data encapsulation is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those functions which are present in the class can access the data. The insulation of the data from direct access by the program is called data hiding or data abstraction. Hiding the complexity of program is called abstraction only essential features are represented. In short we can say that internal working is hidden.

Dynamic Binding:

Refers to linking of function call with function defination is called binding and when it is take place at run time called dynamic binding.

Message Passing:

The process by which one object can interact with other object is called message passing.

Inheritance:

It is the process by which object of one class aquire the properties or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without Modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes.
Example: **Parrot** is a part of the class flying bird which is again a part of the class bird.

Polymorphism:

A Greek term means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

Example:

- Operator Overloading

  You can redefine or overload most of the built in operators.
- Function Overloading

  You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the type and/or the number of arguments in the argument list but not only by return type.

SIMPLE JAVA PROGRAM

```
import java.io.*;
class Example
{
 public static void main(String args[])
{
 System.out.println("This is a simple java program");
}
}
```

1.5 IDENTIFIERS IN JAVA:

Identifiers are the names of variables, methods, classes, packages and interfaces. Unlike literals they are not the things themselves, just ways of referring to them. In the HelloWorld program, HelloWorld, String, args, main and println are identifiers.

Identifiers must be composed of letters, numbers, the underscore _ and the dollar sign $. Identifiers may only begin with a letter, the underscore or a dollar sign.

Each variable has a name by which it is identified in the program. It's a good idea to give your variables mnemonic names that are closely related to the values they hold. Variable names can include any alphabetic character or digit and the underscore _. The main restriction on the names you can give your variables is that they cannot contain any white space. You cannot begin a variable name with a number. It is

important to note that as in C but not as in Fortran or Basic, all variable names are case-sensitive. MyVariable is not the same as myVariable. There is no limit to the length of a Java variable name. The following are legal variable names:

- MyVariable
- myvariable
- MYVARIABLE
- x
- i
- _myvariable
- $myvariable
- _9pins
- andros

TWO CONTROL STATEMENTS

The if Statement

The Java **if** statement works much like the IF statement in any other language. Further, it is syntactically identical to the **if** statements in C, C++, and C#. Its simplest form is shown here:

if(*condition*) *statement*;

Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed. If *condition* is false, then the statement is bypassed. Here is an example:

```
if(num < 100) println("num is less than 100");
```

In this case, if **num** contains a value that is less than 100, the conditional expression is true, and **println( )** will execute. If **num** contains a value greater than or equal to 100, then the **println( )** method is bypassed.

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

Notice that the test for equality is the double equal sign.

The for Loop

As you may know from your previous programming experience, loop statements are an important part of nearly any programming language. Java is no exception. Perhaps the most versatile is the **for** loop. If you are familiar with C, C++, or C#, then you will be pleased to know that the **for** loop in Java works the same way it does in those languages. If you don't know C/C++/C#, the **for** loop is still easy to use.

The simplest form of the **for** loop is shown here:

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

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. Here is a short program that illustrates the **for** loop:

USING BLOCKS OF CODE

Java allows two or more statements to be grouped into *blocks of code,* also called *code blocks.* This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target for Java's **if** and **for** statements. Consider this **if** statement:

```
if(x < y) { // begin a block
x = y;
y = 0;
} // end of block
```

Here, if **x** is less than **y**, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block. Let's look at another example. The following program uses a block of code as the target of a **for** loop.

```
/*
Demonstrate a block of code.
Call this file "BlockTest.java"
*/
class BlockTest {
public static void main(String args[]) {
int x, y;
y = 20;
// the target of this loop is a block
for(x = 0; x<10; x++) {
System.out.println("This is x: " + x);
System.out.println("This is y: " + y);
y = y - 2;
}
}
}
```

The output generated by this program is shown here:

```
This is x: 0
```

```
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

In this case, the target of the **for** loop is a block of code and not just a single statement. Thus, each time the loop iterates, the three statements inside the block will be executed. This fact is, of course, evidenced by the output generated by the program.

As you will see later in this book, blocks of code have additional properties and uses. However, the main reason for their existence is to create logically inseparable units of code.

LEXICAL ISSUES

Now that you have seen several short Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifiers, comments, literals, operators, separators, and keywords. The operators are described in the next chapter. The others are described next.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of valid identifiers are:

Avg Temp count a4 $test this_is_ok

Invalid variable names include:

2count high-temp Not/ok

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment.*

This type of comment is used to produce an HTML file that documents your program.

The documentation comment begins with a **/**** and ends with a ***/**. Documentation comments are explained in Appendix A.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol Name Purpose

( ) Parentheses Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.

{ } Braces Used to contain the values of

Automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.

[ ] Brackets Used to declare array types. Also used when dereferencing array values.

; Semicolon Terminates statements.

, Comma Separates consecutive identifiers

in a variable declaration. Also used to chain statements together inside a **for** statement.

. Period Used to separate package names

from subpackages and classes. Also used to separate a variable or method from a reference variable.

THE JAVA CLASS LIBRARIES

The sample programs shown in this chapter make use of two of Java's built-in methods: **println( )** and **print( )**. As mentioned, these methods are members of the **System** class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes. Throughout Part I of this book, various elements of the standard library classes and methods are described as needed. In Part II, the class libraries are described in detail.

DATA TYPES:

As with all modern programming languages, Java supports several types of data. You may use these types to declare variables and to create arrays. As you will see, Java's approach to these items is clean, efficient, and cohesive.

Java Is a Strongly Typed Language

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

The Primitive Types:

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

- **Integers**: This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers**: This group includes float and double, which represent numbers with fractional precision.
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java isotherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency.Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range.

For example, an int is always 32 bits,

Regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

Integers

Java defines four integer types: byte, short, int, and long.

All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

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

Let's look at each type of integer.

byte:

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127.

Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

Byte variables are declared by use of the byte keyword.

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

byte b, c;

short:

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are **some examples** of short variable declarations:

short s;

short t;

int:

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

Long:

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

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

```
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

This program generates the following **output:**

```
In 1000 days light will travel about 16070400000000 miles.
```

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

Floating-Point Types:

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

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

Float:

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

For example, float can be useful when representing dollars and cents.

Here are **some example** float variable declarations:

```
float hightemp, lowtemp;
```

double:

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

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

```
// Compute the area of a circle.
class Area {
public static void main(String args[]) {
```

```
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Characters

In Java, the data type used to store characters is char.

Note: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. 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 chars.

The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits.

But such is the price that must be paid for global portability.

Here is a program that demonstrates char variables:

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[]) {
char ch1, ch2;
ch1 = 88;  // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

This program displays the following

output:

```
ch1 and ch2: X Y
```

Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you may have used with characters in other languages will work in Java, too.

Although char is designed to hold Unicode characters, it can also be thought of as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

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

The **output** generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

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

Booleans:

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

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statementChapter 3: Data Types, Variables, and Arrays 39
```

```
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The **output** generated by this program is shown here:

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

There are three interesting things to notice about this program.

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

```
if(b == true) ...
```

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

Escape Sequence: Java supports all escape sequence which is supported by C/ C++. A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

VARIABLES:

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

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable

declaration is shown here:

type identifier [= value][, identifier [= value] ...] ;

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

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

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

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, sqrt( ), which is a member of theMath class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of themain( )method. However, Java allows variables to be declared within any block.

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. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: **global and local.**

However, these traditional scopes do not fit well with Java's strict, object-oriented model.

In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred, when classes are described.

For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

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.**

Scopes can be nested.

For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// 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);
}
}
```

As the comments indicate, the variable x is declared at the start of main( )'s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

For example, this fragment is invalid because count cannot be used prior to its declaration:

```
// This fragment is wrong!
count = 100; // oops!  cannot use count before it is declared!
int count;
```

Here is another important point to remember:

variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program.

```
// Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, y is reinitialized to –1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
public static void main(String args[]) {
 int bar = 1;
{              // creates a new scope
int bar = 2; // Compile-time error – bar already defined!
}
}
}
```

Type Conversion and Casting:

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.

For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between in compatible types. To do so, you must use a cast, which performs **an explicit conversion** between incompatible types.

Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- **The two types are compatible.**
- **The destination type is larger than the source type.**

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

For **widening conversions**, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long,or char.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target-type) value;

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

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

Adifferent type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some **type conversions that require casts:**

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

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

This program generates the following **output:**

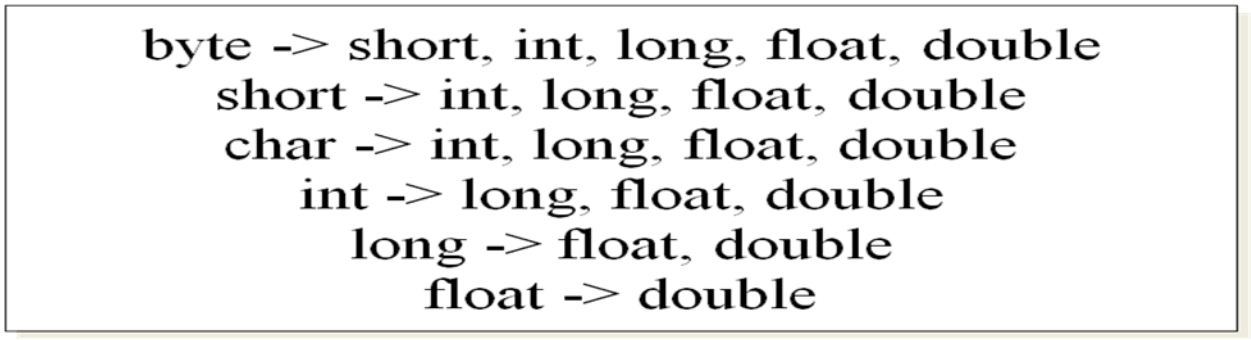
```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case.

When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

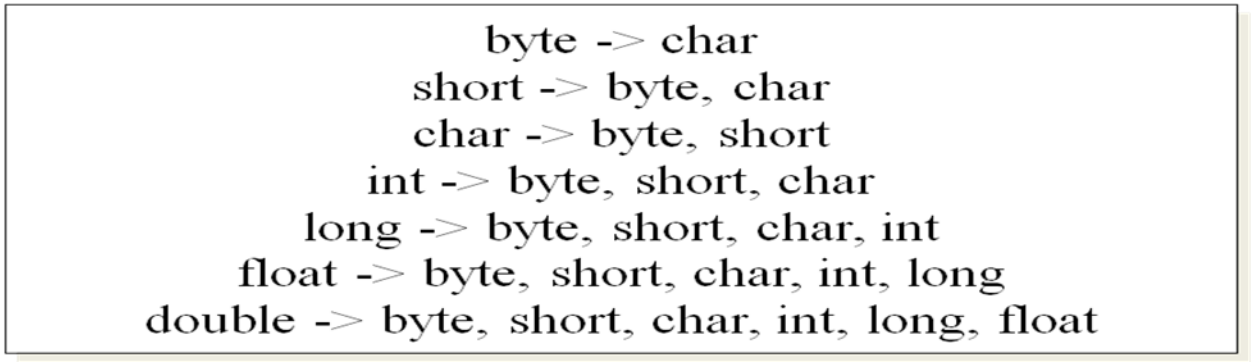
Conclusion of Type Conversion:

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting
- Widening Type Converstion
  - Implicit conversion by compiler automatically



- Narrowing Type Conversion

– Programmer should describe the conversion explicitly



- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double
- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

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. Aspecific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

On which memory, arrays are created in java?

Arrays are created on dynamic memory by JVM. There is no question of static memory in Java; everything (variable, array, object etc.) is created on dynamic memory only.

Arrays: An array represents a group of elements of same data type. Arrays are generally categorized into two types:

- Single Dimensional arrays (or 1 Dimensional arrays)
- Multi-Dimensional arrays (or 2 Dimensional arrays, 3 Dimensional arrays, ...)

Single Dimensional Arrays:

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

type var-name[ ];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.

A one dimensional array or single dimensional array represents a row or a column of elements. For example, the marks obtained by a student in 5 different subjects can be represented by a 1D array.

- We can declare a one dimensional array and directly store elements at the time of its declaration, as*: int marks[] = {50, 60, 55, 67, 70};*
- We can create a 1D array by declaring the array first and then allocate memory for it by using new operator, as:

array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero.

int marks[]; //declare marks array

marks = new int[5]; //allot memory for storing 5 elements

These two statements also can be written as:

int marks [] = new int [5];

- We can pass the values from keyboard to the array by using a loop, as given here

```
for(int i=0;i<5;i++)
{
//read integer values from keyboard and store into marks[i]
BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
Marks[i]=Integer.parseInt(br.readLine());
        }
```

Let us examine some more examples for 1D array:

```
float salary[]={5670.55f,12000f};
```

float[] salary={5670.55f,12000f};

string names[]=new string[10];

string[] names={'v','r'};

Let's review: Obtaining an array is a two-step process.

First, you must declare a variable of the desired array type.

Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

The advantage of using arrays is that they simplify programming by replacing a lot of statements by just one or two statements. In C/C++, by default, arrays are created on static memory unless pointers are used to create them. In java, arrays are created on dynamic memory i.e., allotted at runtime by JVM.

Program : Write a program to accept elements into an array and display the same.

```
// program to accept elements into an array and display the same.
import java.io.*;
class ArrayDemo1
{
   public static void main (String args[]) throws IOException
  {  //Create a BufferedReader class object (br)
     BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
     System.out.println ("How many elements: " );
int n = Integer.parseInt (br.readLine ());
//create a 1D array with size n
     int a[] = new int[n];
     System.out.print ("Enter elements into array : ");
for (int i = 0; i<n;i++)
       a [i] = Integer.parseInt ( br.readLine ());
System.out.print ("The entered elements in the array are: ");
     for (int i =0; i < n; i++)
       System.out.print (a[i] + "\t");
  }
}
```

Output:

```
D:/prakash>javac ArrayDemo1.java
D:/prakash>java ArrayDemo1
How many elements:5
```

Enter elements into arry:10 20 30 40 50

The entered elements in the array are:10 20 30 40 50

D:/prakash>

Multi-Dimensional Arrays (2D, 3D ... arrays):

A two dimensional array is a combination of two or more (1D) one dimensional arrays. A three dimensional array is a combination of two or more (2D) two dimensional arrays.

- **Two Dimensional Arrays (2d array):**

A two dimensional array represents several rows and columns of data. To represent a two dimensional array, we should use two pairs of square braces [ ] [ ] after the array name. For example, the marks obtained by a group of students in five different subjects can be represented by a 2D array.

o We can declare a two dimensional array and directly store elements at the time of its declaration, as:

int marks[] [] = {{50, 60, 55, 67, 70},{62, 65, 70, 70, 81}, {72, 66, 77, 80, 69} };

o We can create a two dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

int marks[ ] [ ]; //declare marks array

marks = new int[3][5]; //allot memory for storing 15 elements.

These two statements also can be written as*: int marks [ ][ ] = new int[3][5];*

Program : Write a program to take a 2D array and display its elements in the form of a matrix.

```
//Displaying a 2D array as a matrix
class Matrix
{  public static void main(String args[])
  {  //take a 2D array
    int x[ ][ ] = {{1, 2, 3}, {4, 5, 6} };
    // display the array elements
    for (int i = 0 ; i < 2 ; i++)
    {  System.out.println ();
      for (int j = 0 ; j < 3 ; j++)
        System.out.print(x[i][j] + "\t");
    }
  }
}
```

Output:

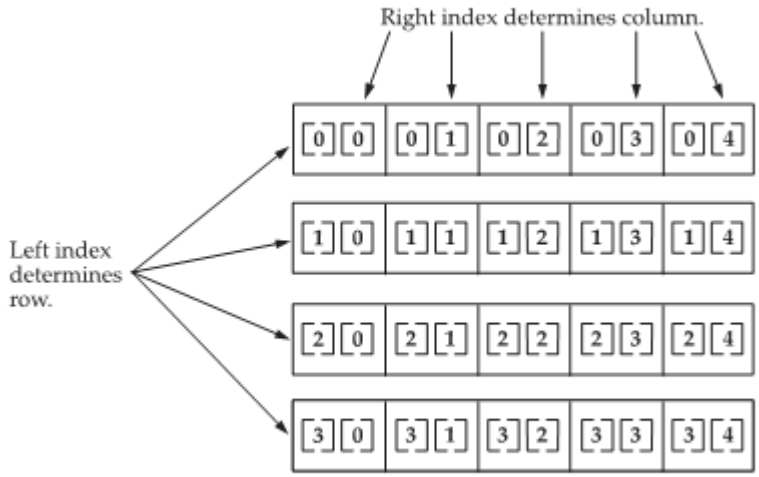
D:/prakash>javac Matrix.java

D:/prakash>java Matrix

1 2 3

4 5 6

D:/prakash>



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

- **Three Dimensional arrays (3D arrays):**

We can consider a three dimensional array as a combination of several two dimensional arrays. To represent a three dimensional array, we should use three pairs of square braces [ ] [ ] after the array name.

o We can declare a three dimensional array and directly store elements at the time of its declaration, as:

int arr[ ] [ ] [ ] = {{{50, 51, 52},{60, 61, 62}}, {{70, 71, 72}, {80, 81, 82}}};

o We can create a three dimensional array by declaring the array first and then we can allot memory for it by using new operator as:

int arr[ ] [ ] = new int[2][2][3]; //allot memory for storing 15 elements.

```
//example for 3-D array
class ThreeD
{
        public static void main(String args[])
        {
                int dept, student, marks, tot=0;
int arr[][][]={{{50,51,52},{60,61,62}},{{70,71,72},{80,81,82}},{{65,66,67},{75,76,77}}};
                for(dept=0;dept<3;dept++)
                {
                        System.out.println("dept"+(dept+1)+":");
```

```
                    for(student=0;student<2;student++)
                    {
                            System.out.print("student"+(student+1)+"marks:");
                            for(marks=0;marks<3;marks++)
                            {
                                    System.out.print(arr[dept][student][marks]+" ");
                                    tot+=arr[dept][student][marks];
                            }
                            System.out.println("total:"+tot);
                            tot=0;
                    }
                    System.out.println();
            }
        }
}
```

arrayname.length:

If we want to know the size of any array, we can use the property 'length' of an array. In case of 2D, 3D length property gives the number of rows of the array.

Alternative Array Declaration Syntax

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

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

int al[] = new int[3];

int[] a2 = new int[3];

The following declarations are also equivalent:

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

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

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

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

creates three array variables of type int. It is the same as writing

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

The alternative declaration form is also useful when specifying an array as a return type for a method.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy they are stored as strings in a String array passed to the args parameter of main( ).

The first command-line argument is stored at args[0], the second at args[1], and so on.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.

class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine
this is a test 100 -1
```

When you do, you will see the following **output:**

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

REMEMBER All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.

1.10 STRINGS

Strings:

A String represents group of characters. Strings are represented as String objects in java.

The String class is defined in the java.lang package and hence is implicitly available to all the programs in Java. The String class is declared as final, which means that it cannot be subclassed. It extends the Object class and implements the Serializable, Comparable, and CharSequence interfaces.

Java implements strings as objects of type String. A string is a sequence of characters. Unlike most of the other languages, Java treats a string as a single value rather than as an array of characters.

Creating Strings:

- We can declare a String variable and directly store a String literal using assignment operator.

  String str = "Hello";

- We can create String object using new operator with some data.

  String s1 = new String ("Java");

- We can create a String by using character array also.

  char arr[] = { 'p','r','o','g','r','a','m'};

- We can create a String by passing array name to it, as:

  String s2 = new String (arr);

- We can create a String by passing array name and specifying which characters we need:

  String s3 = new String (str, 2, 3);

Here starting from 2nd character a total of 3 characters are copied into String s3.

1.11 STRING HANDLING IN JAVA :

String Class Methods:

Method	Description
String concat (String str)	Concatenates calling String with str. **Note:** + also used to do the same
int length ()	Returns length of a String
char charAt (int index)	Returns the character at specified location (from 0)
int compareTo (String str)	Returns a negative value if calling String is less than str, a positive value if calling String is greater than str or 0 if Strings are equal.
boolean equals (String str)	Returns true if calling String equals str. **Note:** == operator compares the references of the string objects. It does not compare the contents of the objects. equals () method compares the contents. While comparing the strings, equals () method should be used as it yields the correct result.
boolean equalsIgnoreCase (String str)	Same as above but ignores the case
boolean startsWith (String prefix)	Returns true if calling String starts with prefix
boolean endsWith (String suffix)	Returns true if calling String ends with suffix
int indexOf (String str)	Returns first occurrence of str in String.
int lastIndexOf(String str)	Returns last occurrence of str in the String. **Note:** Both the above methods return negative value, if str not found in calling String. Counting starts from 0.
String replace (char oldchar, char newchar)	returns a new String that is obtained by replacing all characters oldchar in String with newchar.
String substring (int beginIndex)	returns a new String consisting of all characters from beginIndex until the end of the String
String substring (int beginIndex, int endIndex)	returns a new String consisting of all characters from beginIndex until the endIndex.
String toLowerCase ()	converts all characters into lowercase
String toUpperCase ()	converts all characters into uppercase
String trim ()	eliminates all leading and trailing spaces

String represents a sequence of characters. It has fixed length of character sequence. Once a string object has been created than we can't change the character that comprise that string. It is immutable. This allows String to be shared. String object can be instantiated like any other object

String str = new String ("Stanford “);

str += "Lost!!";

Accessor methods:

length(), charAt(i), getBytes(), getChars(istart,iend,gtarget[],itargstart), split(string,delim), toCharArray(), valueOf(g,iradix), substring(iStart [,iEndIndex)]) [returns up to but not including iEndIndex]

Modifier methods:

concat(g), replace(cWhich, cReplacement), toLowerCase(), toUpperCase(), trim().

Boolean test methods:

contentEquals(g), endsWith(g), equals(g), equalsIgnoreCase(g), matches(g), regionMatches(i1,g2,i3,i4), regionMatches(bIgnoreCase,i1,g2,i3,i4), startsWith(g)

Integer test methods:

compareTo(g) [returns 0 if object equals parameter, -1 if object is before parameter in sort order, +1 if otherwise], indexOf(g) [returns position of first occurrence of substring g in the string, -1 if not found], lastIndexOf(g) [returns position of last occurrence of substring g in the string, -1 if not found], length().

Constructors defined in the String class

The String class defines several constructors. The most common constructor of the String class is the one given below:

public String(String value)

This constructor constructs a new String object initialized with the same sequence of the characters passed as the argument. In other words, the newly created String object is the copy of the string passed as an argument to the constructor.

Other constructors defined in the String class are as follows:

public String()

This constructor creates an empty String object. However, the use of this constructor is unnecessary because String objects are immutable.

public String(char[] value)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the array that is passed as the argument to it.

public String(char[] value, int startindex, int len)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the subarray. This subarray is derived from the character array and the two integer values that are passed as arguments to the constructor. The int variable startindex represents the index value of the

starting character of the subarray, and the int variable len represents the number of characters to be used to form the new String object.

public String(StringBuffer sbf)

This constructor creates a new String object that contains the same sequence of characters currently contained in the string buffer argument.

public String(byte[] asciichars)

The array of bytes that is passed as an argument to the constructor contains the ASCII character set. Therefore, this array of bytes is first decoded using the default charset of the platform. Then the constructor creates a new String object initialized with same sequence of characters obtained after decoding the array.

public String(byte[] asciiChars, int startindex, int len)

This constructor creates the String object after decoding the array of bytes and by using the subarray of bytes.

Special String Operations:

Finding the length of string

The String class defines the length() method that determines the length of a string. The length of a string is the number of characters contained in the string. The signature of the length() method is given below:

public int length()

String Concatenation using the + operator

The + operator is used to concatenate two strings, producing a new String object as the result. For example,

```
String sale = "500";
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars".

The + operator may also be used to concatenate a string with other data types. For example,

```
int sale = 500;
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars". In this case, the variable sale is declared as int rather than String, but the output produced is the same. This is because the int value contained in the variable sale is automatically converted to String type, and then the + operator concatenates the two strings.

String Comparison

The String class defines various methods that are used to compare strings or substrings within strings. Each of them is discussed in the following sections:

Note: Since strings are stored as a memory address, the == operator can't be used for comparisons. Use equals() and equalsIgnoreCase() to do comparisons. A simple example is:

equals()

The equals() method is used to check whether the Object that is passed as the argument to the method is equal to the String object that invokes the method. It returns true if and only if the argument is a String object that represents the same sequence of characters as represented by the invoking object. The signature of the equals() method is as follows:

boolean i=str1.equals(str2)

equalsIgnoreCase()

The equalsIgnoreCase() method is used to check the equality of the two String objects without taking into consideration the case of the characters contained in the two strings. It returns true if the two strings are of the same length and if the corresponding characters in the two strings are the same ignoring case. The signature of the equalsIgnoreCase() method is:

boolean i=str1.equalsIgnoreCase(str2)

compareTo()

The compareTo() method is used in conditions where a Programmer wants to sort a list of strings in a predetermined order. The compareTo() method checks whether the string passed as an argument to the method is less than, greater than, or equal to the invoking string. A string is considered less than another string if it comes before it in alphabetical order. The signature of the compareTo() method is as follows:

int i=str1. compareTo(str2)

where, str is the String being compared to the invoking String. The compareTo() method returns an int value as the result of String comparison. The meaning of these values are given in the following table:

compareToIgnoreCase()

The String class also has the compareToIgnoreCase() method that compares two strings without taking into consideration their case difference. The signature of the method is given below:

int i=str1.compareToIgnoreCase(str2)

regionMatches()

The regionMatches() method is used to check the equality of two string regions where the two string regions belong to two different strings. The signature of the method is given below:

public boolean regionMatches(int startindex, String str2, int startindex2, int len)

There is also an overloaded version of the method that tests the equality of the substring ignoring the case of characters in the substring. Its signature is given below:

public boolean regionMatches(boolean ignoreCase, int startindex, String str2, int startindex2, int len)

In both signatures of the method, startindex specifies the starting index of the substring within the invoking string. The str2 argument specifies the string to be compared. The startindex2 specifies the starting index of the substring within the string to be compared. The len argument specifies the length of the substring being compared. However, in the latter signature of the method, the comparison is done ignoring the case of the characters in the substring only if the ignoreCase argument is true.

startsWith()

The startsWith() method is used to check whether the invoking string starts with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean startsWith(String prefix)

There is also an overloaded version of the startsWith() method with the following signature:

public boolean startsWith(String prefix, int startindex)

In both signatures of the method given above, the prefix denotes the substring to be matched within the invoking string. However, in the second version, the startindex denotes the starting index into the invoking string at which the search operation will commence.

endsWith()

The endsWith() method is used to check whether the invoking string ends with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean endsWith(String prefix)

Modifying a String

The String objects are immutable. Therefore, it is not possible to change the original contents of a string. However, the following String methods can be used to create a new copy of the string with the required modification:

substring()

The substring() method creates a new string that is the substring of the string that invokes the method. The

method has two forms:

public String substring(int startindex)

public String substring(int startindex, int endindex)

where, startindex specifies the index at which the substring will begin and endindex specifies the index at which the substring will end. In the first form where the endindex is not present, the substring begins at startindex and runs till the end of the invoking string.

Concat()

The concat() method creates a new string after concatenating the argument string to the end of the invoking string. The signature of the method is given below:

public String concat(String str)

replace()

The replace() method creates a new string after replacing all the occurrences of a particular character in the string with another character. The string that invokes this method remains unchanged. The general form of the method is given below:

public String replace(char old_char, char new_char)

trim()

The trim() method creates a new copy of the string after removing any leading and trailing whitespace. The signature of the method is given below:

public String trim(String str)

toUpperCase()

The toUpperCase() method creates a new copy of a string after converting all the lowercase letters in the invoking string to uppercase. The signature of the method is given below:

public String toUpperCase()

toLowerCase()

The toLowerCase() method creates a new copy of a string after converting all the uppercase letters in the invoking string to lowercase. The signature of the method is given below:

public String toLowerCase()

Searching Strings

The String class defines two methods that facilitate in searching a particular character or sequence of characters in a string. They are as follows:

IndexOf()

The indexOf() method searches for the first occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring first appears. Otherwise, it returns -1.

The indexOf() method has the following signatures:

public int indexOf(int ch)
public int indexOf(int ch, int startindex)
public int indexOf(String str)
public int indexOf(String str, int startindex)

lastIndexOf()

The lastIndexOf() method searches for the last occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring last appears. Otherwise, it returns –1.

The lastIndexOf() method has the following signatures:

public int lastIndexOf(int ch)
public int lastIndexOf (int ch, int startindex)
public int lastIndexOf (String str)
public int lastIndexOf (String str, int startindex)

Program : Write a program using some important methods of String class.

```
// program using String class methods
class StrOps
{   public static void main(String args [])
{  String str1 = "When it comes to Web programming, Java is #1.";
String str2 = new String (str1);
String str3 = "Java strings are powerful.";
int result, idx;    char ch;
System.out.println ("Length of str1: " + str1.length ());
 // display str1, one char at a time.
for(int i=0; i < str1.length(); i++)
System.out.print (str1.charAt (i));
```

```
System.out.println ();
if (str1.equals (str2) )
System.out.println ("str1 equals str2");
else
System.out.println ("str1 does not equal str2");
if (str1.equals (str3) )
 System.out.println ("str1 equals str3");
else
System.out.println ("str1 does not equal str3");
result = str1.compareTo (str3);
if(result == 0)
System.out.println ("str1 and str3 are equal");
else if(result < 0)
System.out.println ("str1 is less than str3");
else
System.out.println ("str1 is greater than str3");
str2 = "One Two Three One";     // assign a new string to str2
idx = str2.indexOf ("One");
System.out.println ("Index of first occurrence of One: " + idx);
idx = str2.lastIndexOf("One");
System.out.println ("Index of last occurrence of One: " + idx);
}
}
```

Output:

```
D:/prakash>javac StrOPs.java
D:/prakash>java StrOps
Length of str1:46
When it comes to web programming, Java is #1.
Str1 equals str2
Str1 does not equal str3
Str1 is greater than str3
Index of first occurrence of one:0
Index of last occurrence of one:14
D:/prakash>
```

StringBuffer:

StringBuffer:

StringBuffer objects are mutable, so they can be modified. The methods that directly manipulate data of the object are available in StringBuffer class.

Creating StringBuffer:

- We can create a StringBuffer object by using new operator and pass the string to the object, as: StringBuffer sb = new StringBuffer ("Kiran");
- We can create a StringBuffer object by first allotting memory to the StringBuffer object using new operator and later storing the String into it as:

StringBuffer sb = new StringBuffer (30);

In general a StringBuffer object will be created with a default capacity of 16 characters. Here, StringBuffer object is created as an empty object with a capacity for storing 30 characters. Even if we declare the capacity as 30, it is possible to store more than 30 characters into StringBuffer.

To store characters, we can use append () method as:

Sb.append ("Kiran");

This represents growable and writeable character sequence. It is mutable in nature. StringBuffer are safe to be used by multiple thread as they are synchronized but this brings performance penalty.

It defines 3-constructor:

- StringBuffer(); //initial capacity of 16 characters
- StringBuffer(int size); //The initial size
- StringBuffer(String str);

StringBuffer str = new StringBuffer ("Stanford ");

str.append("Lost!!");

StringBuffer Class Methods:

Method	Description
StringBuffer append (x)	x may be int, float, double, char, String or StringBuffer. It will be appended to calling StringBuffer
StringBuffer insert (int offset, x)	x may be int, float, double, char, String or StringBuffer. It will be inserted into the StringBuffer at offset.
StringBuffer delete (int start, int end)	Removes characters from start to end
StringBuffer reverse ()	Reverses character sequence in the StringBuffer
String toString ()	Converts StringBuffer into a String
int length ()	Returns length of the StringBuffer

Program : Write a program using some important methods of StringBuffer class.

```
// program using StringBuffer class methods
import java.io.*;
class Mutable
{  public static void main(String[] args) throws IOException
  {  // to accept data from keyboard
```

```
BufferedReader br=new BufferedReader (new InputStreamReader (System.in));
  System.out.print ("Enter sur name : ");
  String sur=br.readLine ( );
  System.out.print ("Enter mid name : ");
  String mid=br.readLine ( );
  System.out.print ("Enter last name : ");
  String last=br.readLine ( );
  // create String Buffer object
  StringBuffer sb=new StringBuffer ( );
  // append sur, last to sb
  sb.append (sur);
  sb.append (last);
  // insert mid after sur
  int n=sur.length ( );
  sb.insert (n, mid);
  // display full name
  System.out.println ("Full name = "+sb);
  System.out.println ("In reverse ="+sb.reverse ( ));   }}
```

Output:

```
D:/prakash>javac Mutable.java
D:/prakash>java Mutable
Enter sur name:kranthi
Enter mid name:prakash
Enter last name:A
Full name=kranthi prakash A
In reverse=A ramuk ihtnark
D:/prakash>
```

Accessor methods:

capacity(), charAt(i), length(), substring(iStart [,iEndIndex)])

Modifier methods:

append(g), delete(i1, i2), deleteCharAt(i), ensureCapacity(), getChars(srcBeg, srcEnd, target[], targetBeg), insert(iPosn, g), replace(i1,i2,gvalue), reverse(), setCharAt(iposn, c), setLength(),toString(g)

So the basic differences are.....

1. String is immutable but StringBuffer is not.
2. String is not threadsafe but StringBuffer is thread safe
3. String has concat() for append character but StringBuffer has append() method

4. while you create String like String str = new String(); it create 2 object 1 on heap and 1 on String Constant pool and that refered by str but in StringBuffer it Create 1 object on heap

StringBuilder

StringBuilder class is introduced in Java 5.0 version. This class is an alternative to the existing StringBuffer class. If you look into the operations of the both the classes, there is no difference. The only difference between StringBuilder and StringBuffer is that StringBuilder class is not synchronized so it gives better performance. Whenever there are no threading issues, its preferable to use StringBuilder. StringBuffer class can be replaced by StringBuilder with a simple search and replace with no compilation issue.

Accessor methods: capacity(), length(), charAt(i), indexOf(g), lastIndexOf(g)

Modifier methods: append(g), delete(i1, i2), insert(iPosn, g), getChars(i), setCharAt(iposn, c), substring(), replace(i1,i2,gvalue), reverse(), trimToSize(g), toString(g)

String Handling in Java :

The String class is defined in the java.lang package and hence is implicitly available to all the programs in Java. The String class is declared as final, which means that it cannot be subclassed. It extends the Object class and implements the Serializable, Comparable, and CharSequence interfaces.

Java implements strings as objects of type String. A string is a sequence of characters. Unlike most of the other languages, Java treats a string as a single value rather than as an array of characters.

The String objects are immutable, i.e., once an object of the String class is created, the string it contains cannot be changed. In other words, once a String object is created, the characters that comprise the string cannot be changed. Whenever any operation is performed on a String object, a new String object will be created while the original contents of the object will remain unchanged. However, at any time, a variable declared as a String reference can be changed to point to some other String object.

Why String is immutable in Java

Though there could be many possible answer for this question and only designer of String class can answer this, I think below three does make sense

1) Imagine StringPool facility without making string immutable, its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables , so if any one of them change the value others will be automatically gets affected i.e. lets say

String A = "Test"

String B = "Test"

Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.

2) String has been widely used as parameter for many java classes e.g. for opening network connection you can pass hostname and port number as stirng , you can pass database URL as string for opening database connection, you can open any file by passing name of file as argument to File I/O classes.
In case if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization and then can change the file name either deliberately or accidentally and gain access of those file.
3) Since String is immutable it can safely shared between many threads, which is very important for multithreaded programming.
String Vs StringBuffer and StringBuilder

String

Strings: A String represents group of characters. Strings are represented as String objects in java.

Creating Strings:

- We can declare a String variable and directly store a String literal using assignment operator.

  String str = "Hello";

- We can create String object using new operator with some data.

String s1 = new String ("Java");

- We can create a String by using character array also.

  char arr[] = { 'p','r','o','g','r','a','m'};

- We can create a String by passing array name to it, as:

  String s2 = new String (arr);

- We can create a String by passing array name and specifying which characters we need:

String s3 = new String (str, 2, 3);

Here starting from 2nd character a total of 3 characters are copied into String s3.

String Class Methods:

Method	Description
String concat (String str)	Concatenates calling String with str. **Note:** + also used to do the same
int length ()	Returns length of a String
char charAt (int index)	Returns the character at specified location (from 0)
int compareTo (String str)	Returns a negative value if calling String is less than str, a positive value if calling String is greater than str or 0 if Strings are equal.
boolean equals (String str)	Returns true if calling String equals str. **Note:** == operator compares the references of the string objects. It does not compare the contents of the objects. equals () method compares the contents. While comparing the strings, equals () method should be used as it yields the correct result.
boolean equalsIgnoreCase (String str)	Same as above but ignores the case
boolean startsWith (String prefix)	Returns true if calling String starts with prefix
boolean endsWith (String suffix)	Returns true if calling String ends with suffix
int indexOf (String str)	Returns first occurrence of str in String.
int lastIndexOf(String str)	Returns last occurrence of str in the String. **Note:** Both the above methods return negative value, if str not found in calling String. Counting starts from 0.
String replace (char oldchar, char newchar)	returns a new String that is obtained by replacing all characters oldchar in String with newchar.
String substring (int beginIndex)	returns a new String consisting of all characters from beginIndex until the end of the String
String substring (int beginIndex, int endIndex)	returns a new String consisting of all characters from beginIndex until the endIndex.
String toLowerCase ()	converts all characters into lowercase
String toUpperCase ()	converts all characters into uppercase
String trim ()	eliminates all leading and trailing spaces

String represents a sequence of characters. It has fixed length of character sequence. Once a string object has been created than we can't change the character that comprise that string. It is immutable. This allows String to be shared. String object can be instantiated like any other object

String str = new String ("Stanford “);

str += "Lost!!";

Accessor methods: length(), charAt(i), getBytes(), getChars(istart,iend,gtarget[],itargstart), split(string,delim), toCharArray(), valueOf(g,iradix), substring(iStart [,iEndIndex)]) [returns up to but not including iEndIndex]

Modifier methods: concat(g), replace(cWhich, cReplacement), toLowerCase(), toUpperCase(), trim().

Boolean test methods: contentEquals(g), endsWith(g), equals(g), equalsIgnoreCase(g), matches(g), regionMatches(i1,g2,i3,i4), regionMatches(bIgnoreCase,i1,g2,i3,i4), startsWith(g)

Integer test methods: compareTo(g) [returns 0 if object equals parameter, -1 if object is before parameter in sort order, +1 if otherwise], indexOf(g) [returns position of first occurrence of substring g in the string, -1 if not found], lastIndexOf(g) [returns position of last occurrence of substring g in the string, -1 if not found], length().

Constructors defined in the String class

The String class defines several constructors. The most common constructor of the String class is the one given below:

public String(String value)

This constructor constructs a new String object initialized with the same sequence of the characters passed as the argument. In other words, the newly created String object is the copy of the string passed as an argument to the constructor.

Other constructors defined in the String class are as follows:

public String()

This constructor creates an empty String object. However, the use of this constructor is unnecessary because String objects are immutable.

public String(char[] value)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the array that is passed as the argument to it.

public String(char[] value, int startindex, int len)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the subarray. This subarray is derived from the character array and the two integer values that are passed as arguments to the constructor. The int variable startindex represents the index value of the starting character of the subarray, and the int variable len represents the number of characters to be used to form the new String object.

public String(StringBuffer sbf)

This constructor creates a new String object that contains the same sequence of characters currently contained in the string buffer argument.

public String(byte[] asciichars)

The array of bytes that is passed as an argument to the constructor contains the ASCII character set. Therefore, this array of bytes is first decoded using the default charset of the platform. Then the constructor creates a new String object initialized with same sequence of characters obtained after decoding the array.

public String(byte[] asciiChars, int startindex, int len)

This constructor creates the String object after decoding the array of bytes and by using the subarray of bytes.

Special String Operations

Finding the length of string

The String class defines the length() method that determines the length of a string. The length of a string is the number of characters contained in the string. The signature of the length() method is given below:

public int length()

String Concatenation using the + operator

The + operator is used to concatenate two strings, producing a new String object as the result. For example,

```
String sale = "500";
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars".

The + operator may also be used to concatenate a string with other data types. For example,

```
int sale = 500;
String s = "Our daily sale is" + sale + "dollars";
System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars". In this case, the variable sale is declared as int rather than String, but the output produced is the same. This is because the int value contained in the variable sale is automatically converted to String type, and then the + operator concatenates the two strings.

String Comparison

The String class defines various methods that are used to compare strings or substrings within strings. Each of them is discussed in the following sections:

Note: Since strings are stored as a memory address, the == operator can't be used for comparisons. Use equals() and equalsIgnoreCase() to do comparisons. A simple example is:

equals()

The equals() method is used to check whether the Object that is passed as the argument to the method is equal to the String object that invokes the method. It returns true if and only if the argument is a String object that represents the same sequence of characters as represented by the invoking object. The signature of the equals() method is as follows:

public boolean equals(Object str)

equalsIgnoreCase()

The equalsIgnoreCase() method is used to check the equality of the two String objects without taking into consideration the case of the characters contained in the two strings. It returns true if the two strings are of the same length and if the corresponding characters in the two strings are the same ignoring case. The

signature of the equalsIgnoreCase() method is:
public boolean equalsIgnoreCase(Object str)

compareTo()

The compareTo() method is used in conditions where a Programmer wants to sort a list of strings in a predetermined order. The compareTo() method checks whether the string passed as an argument to the method is less than, greater than, or equal to the invoking string. A string is considered less than another string if it comes before it in alphabetical order. The signature of the compareTo() method is as follows:

public int compareTo(String str)

where, str is the String being compared to the invoking String. The compareTo() method returns an int value as the result of String comparison. The meaning of these values are given in the following table:

The String class also has the compareToIgnoreCase() method that compares two strings without taking into consideration their case difference. The signature of the method is given below:

public int compareToIgnoreCase(String str)

regionMatches()

The regionMatches() method is used to check the equality of two string regions where the two string regions belong to two different strings. The signature of the method is given below:

public boolean regionMatches(int startindex, String str2, int startindex2, int len)

There is also an overloaded version of the method that tests the equality of the substring ignoring the case of characters in the substring. Its signature is given below:

public boolean regionMatches(boolean ignoreCase, int startindex, String str2, int startindex2, int len)

In both signatures of the method, startindex specifies the starting index of the substring within the invoking string. The str2 argument specifies the string to be compared. The startindex2 specifies the starting index of

the substring within the string to be compared. The len argument specifies the length of the substring being compared. However, in the latter signature of the method, the comparison is done ignoring the case of the characters in the substring only if the ignoreCase argument is true.

startsWith()

The startsWith() method is used to check whether the invoking string starts with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean startsWith(String prefix)

There is also an overloaded version of the startsWith() method with the following signature:

public boolean startsWith(String prefix, int startindex)

In both signatures of the method given above, the prefix denotes the substring to be matched within the invoking string. However, in the second version, the startindex denotes the starting index into the invoking string at which the search operation will commence.

endsWith()

The endsWith() method is used to check whether the invoking string ends with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean endsWith(String prefix)

Modifying a String

The String objects are immutable. Therefore, it is not possible to change the original contents of a string. However, the following String methods can be used to create a new copy of the string with the required modification:

substring()

The substring() method creates a new string that is the substring of the string that invokes the method. The method has two forms:

public String substring(int startindex)

public String substring(int startindex, int endindex)

where, startindex specifies the index at which the substring will begin and endindex specifies the index at

which the substring will end. In the first form where the endindex is not present, the substring begins at startindex and runs till the end of the invoking string.

Concat()

The concat() method creates a new string after concatenating the argument string to the end of the invoking string. The signature of the method is given below:

public String concat(String str)

replace()

The replace() method creates a new string after replacing all the occurrences of a particular character in the string with another character. The string that invokes this method remains unchanged. The general form of the method is given below:

public String replace(char old_char, char new_char)

trim()

The trim() method creates a new copy of the string after removing any leading and trailing whitespace. The signature of the method is given below:

public String trim(String str)

toUpperCase()

The toUpperCase() method creates a new copy of a string after converting all the lowercase letters in the invoking string to uppercase. The signature of the method is given below:

public String toUpperCase()

toLowerCase()

The toLowerCase() method creates a new copy of a string after converting all the uppercase letters in the invoking string to lowercase. The signature of the method is given below:

public String toLowerCase()

Searching Strings

The String class defines two methods that facilitate in searching a particular character or sequence of characters in a string. They are as follows:

IndexOf()

The indexOf() method searches for the first occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring first appears. Otherwise, it returns -1.

The indexOf() method has the following signatures:

public int indexOf(int ch)
public int indexOf(int ch, int startindex)
public int indexOf(String str)
public int indexOf(String str, int startindex)

lastIndexOf()

The lastIndexOf() method searches for the last occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring last appears. Otherwise, it returns –1.

The lastIndexOf() method has the following signatures:

public int lastIndexOf(int ch)
public int lastIndexOf (int ch, int startindex)
public int lastIndexOf (String str)
public int lastIndexOf (String str, int startindex)

Program : Write a program using some important methods of String class.

```
// program using String class methods

class StrOps
```

```
{   public static void main(String args [])
{  String str1 = "When it comes to Web programming, Java is #1.";
String str2 = new String (str1);
String str3 = "Java strings are powerful.";
int result, idx;    char ch;
System.out.println ("Length of str1: " + str1.length ());
 // display str1, one char at a time.
for(int i=0; i < str1.length(); i++)
System.out.print (str1.charAt (i));
System.out.println ();
if (str1.equals (str2) )
System.out.println ("str1 equals str2");
else
System.out.println ("str1 does not equal str2");
if (str1.equals (str3) )
 System.out.println ("str1 equals str3");
else
System.out.println ("str1 does not equal str3");
result = str1.compareTo (str3);
if(result == 0)
System.out.println ("str1 and str3 are equal");
```

```
else if(result < 0)

System.out.println ("str1 is less than str3");

else

System.out.println ("str1 is greater than str3");

str2 = "One Two Three One";     // assign a new string to str2

idx = str2.indexOf ("One");

System.out.println ("Index of first occurrence of One: " + idx);

idx = str2.lastIndexOf("One");

System.out.println ("Index of last occurrence of One: " + idx);

}

}
```

Output:

StringBuffer

StringBuffer: StringBuffer objects are mutable, so they can be modified. The methods that directly manipulate data of the object are available in StringBuffer class.

Creating StringBuffer:

- We can create a StringBuffer object by using new operator and pass the string to the object, as: StringBuffer sb = new StringBuffer ("Kiran");
- We can create a StringBuffer object by first allotting memory to the StringBuffer object using new operator and later storing the String into it as:

StringBuffer sb = new StringBuffer (30);

In general a StringBuffer object will be created with a default capacity of 16 characters. Here, StringBuffer object is created as an empty object with a capacity for storing 30 characters. Even if we declare the capacity as 30, it is possible to store more than 30 characters into StringBuffer.

To store characters, we can use append () method as:

Sb.append ("Kiran");

This represents growable and writeable character sequence. It is mutable in nature. StringBuffer are safe to be used by multiple thread as they are synchronized but this brings performance penalty.
It defines 3-constructor:

- StringBuffer(); //initial capacity of 16 characters
- StringBuffer(int size); //The initial size
- StringBuffer(String str);

StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");

StringBuffer Class Methods:

Method	Description
StringBuffer append (x)	x may be int, float, double, char, String or StringBuffer. It will be appended to calling StringBuffer
StringBuffer insert (int offset, x)	x may be int, float, double, char, String or StringBuffer. It will be inserted into the StringBuffer at offset.
StringBuffer delete (int start, int end)	Removes characters from start to end
StringBuffer reverse ()	Reverses character sequence in the StringBuffer
String toString ()	Converts StringBuffer into a String
int length ()	Returns length of the StringBuffer

Program : Write a program using some important methods of StringBuffer class.

```
// program using StringBuffer class methods
import java.io.*;
class Mutable
{  public static void main(String[] args) throws IOException
  {  // to accept data from keyboard
  BufferedReader br=new BufferedReader (new InputStreamReader (System.in));
    System.out.print ("Enter sur name : ");
    String sur=br.readLine ( );
    System.out.print ("Enter mid name : ");
    String mid=br.readLine ( );
    System.out.print ("Enter last name : ");
    String last=br.readLine ( );
    // create String Buffer object
    StringBuffer sb=new StringBuffer ( );
    // append sur, last to sb
    sb.append (sur);
    sb.append (last);
    // insert mid after sur
    int n=sur.length ( );
    sb.insert (n, mid);
    // display full name
```

```
    System.out.println ("Full name = "+sb);

    System.out.println ("In reverse ="+sb.reverse ( ));

  }

}
```

Output:

UNIT-II

OPERATORS:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

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

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand,	(A > B) is not true.

	if yes then condition becomes true.	
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

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

Bitwise operator works on bits and performs bit-by-bit operation. Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a

		signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A |B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

There are following assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A

*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C 2

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Following is the example:

```
public class Test {

  public static void main(String args[]){
    int a , b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );

    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
```

```
   }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

```
public class Test {

   public static void main(String args[]){
      String name = "James";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR	|	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

CONTROL STATEMENTS:

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. In Java control statements are categorized into selection control statements, iteration control statements and jump control statements.

- **Java's Selection Statements:**

Java supports two selection statements: if and switch. These statements allow us to control the flow of program execution based on condition.

if Statement:

if statement performs a task depending on whether a condition is true or false.

Syntax: if (condition)

```
    statement1;
  else
    statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

Program : Write a program to find biggest of three numbers.

```
//Biggest of three numbers
class BiggestNo
{  public static void main(String args[])
  {  int a=5,b=7,c=6;
    if ( a > b && a>c)
      System.out.println ("a is big");
    else if ( b > c)
      System.out.println ("b is big");
    else
      System.out.println ("c is big");
  }
}
```

Output:

```
D:/prakash>javac BiggestNo.java
D:/prakash>java BiggestNo
b is big
D:/prakash>
```

Switch Statement:

When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax: switch (expression)

```
  {  case value1:  //statement sequence
       break;
    case value2:  //statement sequence
       break;
     .....
    case valueN:  //statement sequence
       break;
    default:  //default statement sequence
```

```
   }
```

Here, depending on the value of the expression, a particular corresponding case will be executed.

Program : Write a program for using the switch statement to execute a particular task depending on color value.

```
//To display a color name depending on color value
class ColorDemo
{  public static void main(String args[])
  {  char color = 'r';
    switch (color)
    {  case 'r': System.out.println ("red");    break;
      case 'g': System.out.println ("green");  break;
      case 'b': System.out.println ("blue");    break;
      case 'y': System.out.println ("yellow");  break;
      case 'w': System.out.println ("white");  break;
      default: System.out.println ("No Color Selected");
    }
  }
}
```

Output:

```
D:/prakash>javac ColorDemo.java
D:/prakash>java ColorDemo
red
D:/prakash>
```

Java's Iteration Statements:

Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

while Loop:

while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.

Syntax: while (condition)

```
   {
     statements;
```

```
   }
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i=1;
    while (i <= 20)
    {  System.out.print (i + "\t");
      i++;
    }
  }
}
```

Output:

```
D:/prakash>javac Natural.java
D:/prakash>java Natural
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
D:/prakash>
```

do-while Loop:

do...while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do...while loop is also called as exit control loop.

Syntax: do

```
   {
     statements;
   } while (condition);
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural1
{  public static void main(String args[])
  {  int i=1;
    do
    {  System.out.print (i + "\t");
      i++;
```

```
   } while (i <= 20);
  }
}
```

Output:

```
D:/prakash>javac Natural1.java
D:/prakash>java Natural1
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
D:/prakash>
```

for Loop:

The for loop is also same as do...while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)

```
   {   statements;
   }
```

Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural2
{  public static void main(String args[])
  {  int i;
    for (i=1; i<=20; i++)
      System.out.print (i + "\t");
  }
}
```

Output:

```
D:/prakash>javac Natural2.java
D:/prakash>java Natural2
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
D:/prakash>
```

Jump Statements:

Java supports three jump statements: break, continue and return.

These statements transfer control to another part of the program.

break:

Ø break can be used inside a loop to come out of it.

Ø break can be used inside the switch block to come out of the switch block.

Ø break can be used in nested blocks to go to the end of a block. Nested blocks represent a block written within another block.

Syntax: break; (or) break label;//here label represents the name of the block.

Program : Write a program to use break as a civilized form of goto.

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

Output:

```
D:/prakash>javac BreakDemo.java
D:/prakash>java BreakDemo
Before the break
This is after second block
D:/prakash>
```

continue:

This statement is useful to continue the next repetition of a loop/ iteration. When continue is executed, subsequent statements inside the loop are not executed.

Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main (String args[])
  {  int i=1;
    while (true)
    {  System.out.print (i + "\t");
      i++;
      if (i <= 20 )
        continue;
      else
        break;
    }
  }
}
```

Output:

```
D:/prakash>javac Natural.java
D:/prakash>java Natural
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
D:/prakash>
```

return statement:

Ø return statement is useful to terminate a method and come back to the calling method.

Ø return statement in main method terminates the application.

Ø return statement can be used to return some value from a method to a calling method.

Syntax: return; (or)

return value; // value may be of any type

Program : Write a program to demonstrate return statement.

```
//Demonstrate return
class ReturnDemo
{  public static void main(String args[])
```

```
  {  boolean t = true;
    System.out.println ("Before the return");
    if (t)
      return;
    System.out.println ("This won't execute");
  }
}
```

Output:

```
D:/prakash>javac ReternDemo.java
D:/prakash>java ReturnDemo
Before the return
D:/prakash>
```

Note: goto statement is not available in java, because it leads to confusion and forms infinite loops.

CLASSES

Concepts of classes, objects:

The class is at the core of Java. **It is the logical construct** upon which the entire Java language is built because it defines the shape and nature of an object.

As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java.

Class Fundamentals:

The classes created in the preceding chapters primarily exist simply to encapsulate the main( )method, which has been used to demonstrate the basics of the Java syntax.

Perhaps the most important thing to understand about a class is that it defines a new **data type,** defined, this new type can be used to **create objects of that type**.

Thus, **a class is a template for an object**, and **an object is an instance of a class**.

Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The General Form of a Class:

When you define a class, you declare its exact form and nature. You do this 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. As you will see, a class' code defines the interface to its data.

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

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

The data, or variables, defined within a class are called **instance variables**. The code is contained within methods. Collectively, the methods and variables defined within a class are called **members of the class.** In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public.

Notice that the general form of a class does not specify a main( )method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, applets don't require a main( ) method at all.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.

- Variables in method declarations—these are called *parameters*.
-

The Bicycle class uses the following lines of code to define its fields:

public int cadence;
public int gear;
public int speed;

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

A Simple Class:

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth.

```
// This program declares two Box objects.
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
```

```
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

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

- As you can see, mybox1's data is completely separate from the data contained in mybox2.
- As stated, a class defines a new type of data. In this case, the new data type is called Box.

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

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

- After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.
- To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

For example, to assign the width variable of mybox the value 100, you would use the following statement:

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

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

- In general, you use the dot operator to access both the instance variables and the methods within an object.
- You should call the file that contains this program BoxDemo2.java, because the main() method is in the class called BoxDemo2, not the class called Box.
- When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo2.
- The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo2 class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo2.java, respectively.

To run this program, you must execute BoxDemo2.class. When you do, you will see the following output:

Volume is 3000.0

Volume is 162.0

- As stated earlier, each object has its own copies of the instance variables.

This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

Declaring Objects

An object is an instance of a class. An object is known by a name and every object contains a state. The state is determined by the values of attributes (variables). The state of an object can be changed by calling methods on it. The sequence of state changes represents the behavior of the object.

An object is a software entity (unit) that combines a set of data with a set of operations to manipulate that data.

As just explained, **when you create a class, you are creating a new data type**. You can use this type 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. You can do this using the new operator.

- **Declaration**: The code set in variable declarations that associate a variable name with an object type.
- **Instantiation**: The new keyword is a Java operator that creates the object.
- **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

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

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

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

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.

Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted in Figure.

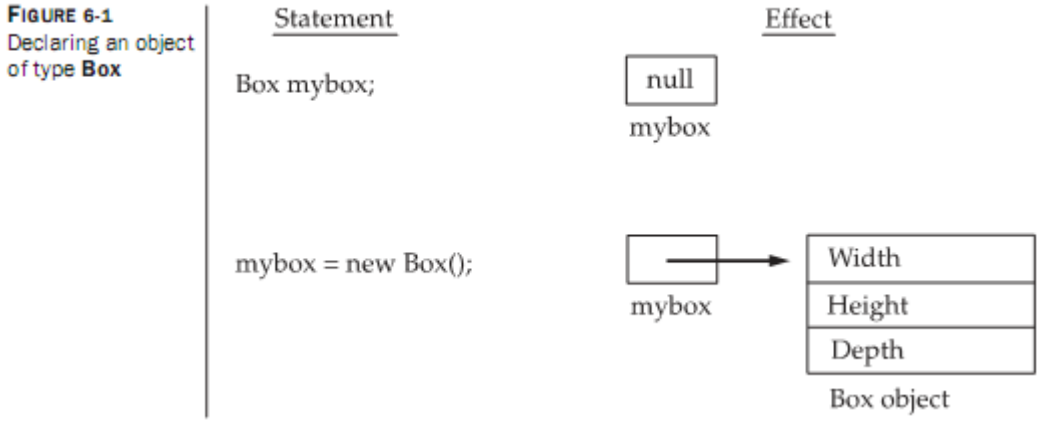


FIGURE 6-1 Declaring an object of type **Box**

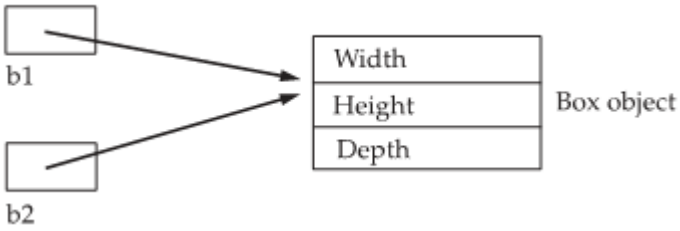
Assigning Object Reference Variables:

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

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

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although b1 and b2 both refer to the same object, they are not linked in any other way.

For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2. For example:

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

Box b2 = b1;

// ...

b1 = null;

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

REMEMBER: When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Access Modifiers:

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

1.private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

2.default:

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

3.protected:

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

4.public:

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Methods :

Here is an example of a typical method declaration:

```
public double calculateAnswer(double width, int numberofitems,
                              double length, double height) {
   //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.

4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

calculateAnswer(double, int, double, double)

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized.

Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

return;

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

returnValue;

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The getArea() method in the Rectangle Rectangle class that was discussed in the sections on objects returns an integer:

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

This method returns the integer that the expression width*height evaluates to.

Parameters:

Parameters refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

parameter passing:

In general, **there are two ways that a computer language can pass an argument to a subroutine.**

The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

As you will see, Java uses both approaches, depending upon what is passed.

In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
```

```
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

The output from this program is shown here:

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

As you can see, the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.

Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

For example, consider the following program:

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

```
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Note: When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
```

```
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Constructors:

It can be tedious to initialize all of the variables in a class each time an instance is created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.

You can rework the Box example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace setDim( ) with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the dimensions of a box.*/
```

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
```

```
double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}
```

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

```
Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0
```

As you can see, both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both mybox1 and mybox2 will have the same volume. The println( ) statement inside Box( ) is for the sake of illustration only.

Parameterized Constructors

While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor.

As you can probably guess, this makes them much more useful. For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.

```
/* Here, Box uses a parameterized constructor to initialize the dimensions of a box.*/

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// compute and return volume

double volume() {

return width * height * depth;

}

}

class BoxDemo7 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

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

Box mybox2 = new Box(3, 6, 9);
```

```
double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}
```

The output from this program is shown here:

```
Volume is 3000.0

Volume is 162.0
```

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

For example, in the following line,

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

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

Overloading Constructors:

Since Box( ) requires three arguments, it's an error to call it without them.

This raises some important questions.

What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?

As the Box class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described. Here is a program that contains an improved version of Box that does just that:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1;  // use -1 to indicate
height = -1; // an uninitialized
depth = -1;  // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
```

```
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

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

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

Recursion:

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

A method that calls itself is said to be recursive.

The classic **example** of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
```

```
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output from this program is shown here:

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically.

Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By annonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();

e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();

Employee e2=new Employee();

 e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By annonymous object:

```
new Employee();
```

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

```
protected void finalize(){}
```

UNDERSTANDING STATIC:

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

why java main method is static?

Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

Example of static method, variable:

```
//Program of changing the common property of all objects(static field).

class Student9{
  int rollno;
  String name;
  static String college = "ITS";

  static void change(){
  college = "BBDIT";
  }

  Student9(int r, String n){
  rollno = r;
  name = n;
  }

  void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student9.change();
```

```
        Student9 s1 = new Student9 (111,"Karan");
        Student9 s2 = new Student9 (222,"Aryan");
        Student9 s3 = new Student9 (333,"Sonoo");

        s1.display();
        s2.display();
        s3.display();
        }
    }
```

Test it Now

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:static block is invoked

Hello main

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
   ...
   class NestedClass {
      ...
```

```
    }
}
```

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

STATIC NESTED CLASSES:

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Non Static Nested Classes(Inner class):

Non static nested class is also known as inner class. It has access to all variables and methods of outer class and may refer to them directly. But the reverse is not true, that is outer class cannot directly access members of inner class. One more thing is an inner class must created and instantiated within the scope of outer class only.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.*

By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or

indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

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

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

Stack Class

```
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
```

```
else
return stck[tos--];
}
}

class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
```

16
15
14
13
12
11
10

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

UNIT III

INHERITANCE:

• Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.

• In the terminology of Java, a class that is inherited is called a “superclass”.

• The class that does the inheriting is called a “subclass”.

• Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance Basics:

• To inherit a class, you simply incorporate the definition of one class into another by using the “extends” keyword.

PROGRAM 14

```
// A simple example of inheritance.
// 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();
}
}
```

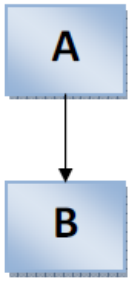
Types of Inheritance:

1. Single
2. Multi level
3. Hierarchal
4. Hybrid

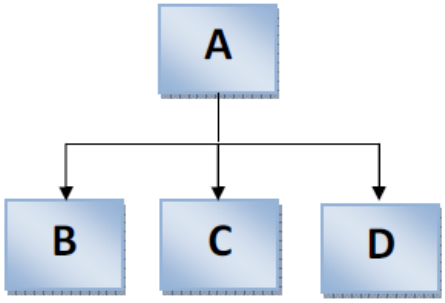
1.Single 3.Hierarchal

2. Multi level 4.Hybrid

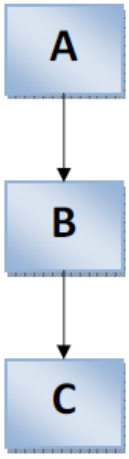
1.Single



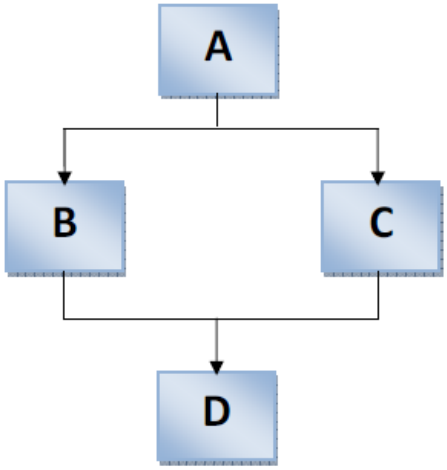
3.Hierarchal



2. Multi level



4.Hybrid



MEMBER ACCESS AND INHERITANCE:

The subclass cannot access its parent's private members directly. If we try to access them the JVM generates compile time error. To overcome this drawback we have an alternative method to access the private members of parent in child class or subclass. The following example elucidates

```
// Create a superclass.
class A
{
int i; // public by default
private int j; // private to A
void setij(int x, int y)
{
i = x;
j = y;
}
}
```

```
// A's j is not accessible here.
class B extends A
{
int total;
void sum()
{
total = i + j; // ERROR, j is not accessible here
}
}
class Access
{
public static void main(String args[])
{
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

ACCESSING PRIVATE MEMBERS OF SUPER CLASS BY METHODS:

```
class TwoDshape
{
private double width;
private double height;
double getwidth(){



return width;}
double getheight(){
return height;}
void setwidth(double w){width=w;}
void setheight(double h){height=h;}
void showDim()
{
System.out.println("width and height are"+width+"and"+height);
}
}
```

```
class Triangle extends TwoDshape
{
String style;
double area()
{
return (getwidth()*getheight())/2;
}
void showStyle()
{
System.out.println("Triangle is "+style);
}
}
class shapes
{
public static void main(String args[])
{
Triangle t1=new Triangle();
Triangle t2=new Triangle();
t1.setwidth(4.0);
t1.setheight(4.0);
t1.style="filled";
t2.setwidth(8.0);
t2.setheight(12.0);
t2.style="outlined";
System.out.println("Information for t1:");
t1.showStyle();
t1.showDim();
System.out.println("Area is "+t1.area());
System.out.println("Information for t2:");
t2.showStyle();
t2.showDim();
System.out.println("Area is "+t2.area());
}
}
```

"A SUPERCLASS VARIABLE CAN REFERENCE A SUBCLASS OBJECT"

• A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations.

```
class RefDemo
{
public static void main(String args[])
{
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

CREATING A MULTILEVEL HIERARCHY:

You can build hierarchies that contain as many layers of inheritance as you like.

```
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
```

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



}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Add weight.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
```

```
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
// Add shipping costs
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();


cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}
class DemoShipment {
public static void main(String args[]) {
```

```
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}
```

WHEN CONSTRUCTORS ARE CALLED?

• When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?

• The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.

• Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.

```
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
```

```
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

DYNAMIC METHOD DISPATCH:

• Method overriding forms the basis for one of Java's most powerful concepts: Dynamic method dispatch.

• Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Program

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
```

```
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object


r.callme(); // calls C's version of callme
}
}
```

SUPER KEYWORD:

"super" has two general forms.

1.The first form calls the super class constructor.

2.The second is used to access a member of a super class that has been hidden by a member of subclass

Case 1: To call the super class constructor the very first statement in subclass constructor should be "super(parameter_list);". The following example demonstrates the first form of "super".

Program:

```
class parent
{
int width,length;
parent()
{
width=4;
length=5;
}
parent(int i,int j)
{
width=i;
length=j;
}
}
```

```
class child extends parent{
child()
{
super(5,6);
}
public int area()
{
return(width*length);
}
}
class sup1
{
public static void main(String args[])
{
child c=new child();
System.out.println("Area is:"+c.area());



}
}
```

Output:

Area is:30

Case 2: The second use of super keyword is to access members of parent class when naming collision occurs.

Program:

```
class parent
{
int width,length;
parent()
{
width=4;
length=5;
}
parent(int i,int j)
{
width=i;
length=j;
```

```
}
}
class child extends parent{
int width=10,length=20;
child()
{
super();
}
public int area()
{
return (super.width*super.length);
}
}
class sup2
{
public static void main(String args[])
{
child c=new child();
System.out.println("Area is:"+c.area());
}
}
```

Output:

Area is:20

FINAL KEYWORD

Final has two important usages in inheritance

1. If final is kept in fromt of a class, that class cannot be inherited further.

2. If final is kept in front of a method, that method cannot be overridden further.

Program:

```
final class fig
{
double wid,len;
fig()
{
wid=2.5;
len=3.0;
```

```
}
fig(double i,double j)
{
wid=i;
len=j;
}
final public void area() //This method cannot be overridden as final is used
{
System.out.println("Area cannot be computed for fig class");
}
}
class tri extends fig //This is invalid as final is used for fig class
{
tri()
{
super(4.5,5.2);
}
public void area() //this returns error since this cannot override final area above
{
System.out.println("area of triangle is "+((wid*len)/2));
}
}
class rec extends fig //This is invalid as final is used for fig class
{
rec()
{
super(4.2,5.0);
}
public void area() //this returns error since this cannot override final area above
{


System.out.println("area of rectangle is:"+(wid*len));
}
}
class findemo1
{
```

```
public static void main(String args[])
{
tri t=new tri();
t.area();
rec r=new rec();
r.area();
}
}
```

Output:

```
findemo1.java:19: error: cannot inherit from final fig
class tri extends fig //This is invalid as final is used for fig class
^
findemo1.java:30: error: cannot inherit from final fig
class rec extends fig //This is invalid as final is used for fig class
^
findemo1.java:25: error: area() in tri cannot override area() in fig
public void area() //this returns error since this cannot override
final area above
^
overridden method is final
findemo1.java:36: error: area() in rec cannot override area() in fig
public void area() //this returns error since this cannot override
final area above
^
overridden method is final
4 errors
```

METHOD OVERRIDING:

If same method with same name, type, parameter lists and access modifiers resides in all classes with respect to inheritance then by making use of the concerned class object will access that method. In this case the method with same name, type, parameter lists and access modifiers is said to be overridden. This concept is called method overriding.

Program 1:

```
class fig
{
double wid,len;
fig()
```

```
{
wid=2.5;
len=3.0;
}
fig(double i,double j)
{
wid=i;
len=j;
}
public void area()
{
System.out.println("Area cannot be computed for fig class");
}
}
class tri extends fig
{
tri()
{
super(4.5,5.2);
}
public void area()
{
System.out.println("area of triangle is "+((wid*len)/2));
}
}
class rec extends fig
{
rec()
{
super(4.2,5.0);
}
public void area()
{
System.out.println("area of rectangle is:"+(wid*len));
}
}
```

```
class demo1
{
public static void main(String args[])
{
tri t=new tri();
t.area();
rec r=new rec();


r.area();
}
}
```

Output:

area of triangle is 11.700000000000001

area of rectangle is:21.0

Program 2(to pass through object):

```
class fig
{
double wid,len;
fig()
{
wid=2.5;
len=3.0;
}
fig(double i,double j)
{
wid=i;
len=j;
}
public void area()
{
System.out.println("Area cannot be computed for fig class");
}
}
class tri extends fig
{
tri(double i,double j)
```

```
{
super(i,j);
}
public void area()
{
System.out.println("area of triangle is "+((wid*len)/2));
}
}
class rec extends fig
{
rec(double i,double j)
{
super(i,j);


}
public void area()
{
System.out.println("area of rectangle is:"+(wid*len));
}
}
class demo2
{
public static void main(String args[])
{
tri t=new tri(4.0,5.0);
t.area();
rec r=new rec(3.0,4.0);
r.area();
}
}
```

Output:

area of triangle is 10.0

area of rectangle is:12.0

ABSTRACT KEYWORD:

There are some situations in which the super class is not in a position to provide implementation for its own methods. For example in the above program called find areas the super class “figure” has a method

called area. Here there is no proper implementation for this method. In such case java provides abstract keyword which helps to declare its non implementable methods as abstract.

Syntax: abstract type methodname();

If a class has one or more abstract methods, then it should declare itself as abstract class.

Generally an abstract class has

1. Abstract methods(without body or skeleton)

2. The abstract methods of parent class or superclass must be implemented by its subclass

A subclass which is not in a position to implement its parent's abstract methods should declare itself as abstract.

3. Abstract classes cannot be instantiated, means objects cannot be created for these classes.

4. Abstract classes can have constructors. These constructors should be called by its subclasses at the time of their object's creation.

```
abstract class figure //abstract class
{
double dim1,dim2;
figure(double a,double b)
{
dim1=a;
dim2=b;
}
abstract double area(); //abstract method
}


class rectangle extends figure
{
rectangle(double a,double b)
{
super(a,b);
}
//override area for rectangle
double area()
{
return (dim1*dim2);
}
}
class triangle extends figure
```

```
{
triangle(double a,double b)
{
super(a,b);
}
//override area for triangle
double area()
{
return (dim1*dim2)/2;
}
}
class areas
{
public static void main(String args[])
{
rectangle r=new rectangle(2,3);
triangle t=new triangle(4,5);
System.out.println("the area of rectangle is "+r.area());
System.out.println("the area of triangle is "+t.area());
}
}
```

INTERFACES

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- In practice, this means that you can define interfaces which don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- However, each class is free to determine the details of its own implementation.

- By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple

methods" aspect of polymorphism.

DEFINING AN INTERFACE

• An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

• Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.

• When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier.

• Notice that the methods which are declared have no bodies.

• They end with a semicolon after the parameter list.

• They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

• Variables can be declared inside of interface declarations.

• They are implicitly final and static, meaning they cannot be changed by the implementing class.

• They must also be initialized with a constant value.

• All methods and variables are implicitly public if the interface, itself, is declared as public.

Example:

```
interface Callback
{
void callback(int param);
}
```

IMPLEMENTING INTERFACES

• Once an interface has been defined, one or more classes can implement that interface.

• To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

• The general form of a class that includes the implements

```
access class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

Example-1

```
class Client implements Callback
{
// Implement Callback's interface
public void callback(int p)
{
System.out.println("callback called with " + p);



}
}
```

Note: When you implement an interface method, it must be declared as public.

Example-2

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

PARTIAL IMPLEMENTATIONS

• If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

For example:

```
abstract class Incomplete implements Callback
{
int a, b;
void show() {
System.out.println(a + " " + b);
}
```

}

PACKAGES

• Packages and interfaces are two of the basic components of a Java program.

• Packages are containers for classes that are used to keep the class name space "compartmentalized".

• Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

• In general, a Java source file can contain any (or all) of the following four internal parts:

■ A single package statement (optional)

■ Any number of import statements (optional)

■ A single public class declaration (required)

■ Any number of classes private to the package (optional).

• Java provides a mechanism for partitioning the class name space into more "manageable chunks".

• This mechanism is the "package".

• The package is both a "naming" and a "visibility" control mechanism.

• You can define classes inside a package that are not accessible by code outside that package.

• You can also define class members that are only exposed to other members of the same package.

• This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

DEFINING A PACKAGE

• To create a package is quite easy: simply include a package command as the first statement in a Java source file.

• Any classes declared within that file will belong to the specified package.

• This is the general form of the package statement:

package p1;

• Here, p1 is the name of the package.

HIERARCHY OF PACKAGES

• You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

• The general form of a multileveled package statement is shown here:

Syntax : package pkg1[.pkg2[.pkg3]];

Ex : package java.awt.image;

Simple Example

```
// A simple package
package MyPack;
class Balance
{
String name;
double bal;
Balance(String n, double b)
{
name = n;
bal = b;
}
void show()
{
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance
{
public static void main(String args[])
{
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

ACCESS PROTECTION

• 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

IMPORTING PACKAGES

• Java includes the import statement to bring certain classes, or entire packages, into visibility.

• Once imported, a class can be referred to directly, using only its name.

• In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.

• This is the general form of the import statement:

import pkg1[.pkg2].(classname|*);

• Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).

• There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

Ex:

```
import java.util.Date;
import java.io.*;
```

• If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.

• In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

• Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy.

• For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

• The same example without the import statement looks like this:

```
class MyDate extends java.util.Date {
}
```

• when a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

Program

```
package MyPack;
public class Balance
{
String name;
double bal;
public Balance(String n, double b)
{
name = n;
bal = b;
}
public void show()
{
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
import MyPack.*;
class TestBalance
{
public static void main(String args[])
{
Balance test = new Balance("cse", 99.88);
test.show(); // you may also call show()
}
}
```

EXCEPTION HANDLING

Error

- These are errors related to that are beyond your control such as those occur in JVM.

Exception

- Errors that result from program activity are represented by sub classes of Exception.
- Ex: Division by zero , Array Index Out Of Bounds , IO Errors....etc

Note: Important Subclass of Exception is “RuntimeException”.

Exception Fundamentals

- Exceptions are managed by five important keywords.

• They are 1. Try

2. Catch

3. Throw

4. Throws

5. 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.

Syntax:

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
}
```

2

Uncaught Exceptions

Example 1:

```
class Example
{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
```

```
}
}
```

Error:

java.lang.ArithmeticException: / by zero at Example.main(Example.java:6)

Example 2

```
class Ex
{
static void subroutine()
{
int d = 0;
int a = 10 / d;
}
public static void main(String args[])
{
Ex.subroutine();
}
}
```

Error/Exception handled by JVM

java.lang.ArithmeticException: / by zero at Ex.subroutine(Ex.java:6)

at Ex.main(Ex.java:10)

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.

Example:

```
class Ex1
{
public static void main(String args[])
{

3
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.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Important Note:

- The println( ) inside the try block is never executed.
- Once an exception is thrown, program control transfers out of the try block into the catch block.
- Put differently, catch is not "called," so execution never "returns" to the try block from a catch.
- Thus, the line "This will not be printed." is not displayed.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

Ex:

```
class Exc2
{
public static void main(String args[])
{
int d, a;
try
{
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{
```

```
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

4

This program generates the following output:

```
Division by zero.
After catch statement.
```

Example....

```
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);
}
}
}
```

Displaying a Description of an Exception

- You can display this description in a println( ) statement by simply passing the exception as an

argument.

• Ex:

```
catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

Error:

Exception: java.lang.ArithmeticException: / by zero

5

Multiple catch Clauses

• 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.

Program 1:

```
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.");
}
}
```

A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.

Program 2:

```
class SuperSubCatch
{
public static void main(String args[])
{
try
{
int a = 0;
int b = 42 / a;
}
catch(Exception e)
{
System.out.println("Generic Exception catch.");
}

6
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e)
{ // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

Nested try Statements

- The try statement can be nested.
- That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is

unwound and the next try statement's catch handlers are inspected for a match.

• This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.

• If no catch statement matches, then the Java run-time system will handle the exception.

Program 3:

```
class NestTry
{
public static void main(String args[])
{
try
{
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try
{ // nested try block
if(a==1) a = a/(a-a); // division by zero
if(a==2)
{
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

7

Program 4:

```
class MethNestTry
{
static void nesttry(int a)
{
try
{ // nested try block
f(a==1) a = a/(a-a); // division by zero
if(a==2)
{
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[])
{
try
{
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

Throw

• So far, you have only been catching exceptions that are thrown by the Java run-time system.

• However, it is possible for your program to throw an exception explicitly, using the throw statement.

```
throw ThrowableInstance;
Program 5:
class ThrowDemo
{
static void demoproc()
{
try
{
throw new NullPointerException("demo");

8
}
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);
}
}
}
```

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.

Syntax:

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

Program 6:

```
// This program contains an error and will not compile.
class ThrowsDemo
{
static void throwOne()
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
throwOne();
}
```

9

```
}
```

Program 7:

```
// 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);
}
}
}
```

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.

• finally creates a block of code that will be executed after a try/catch block has completed 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.

• The finally clause is optional.

• However, each try statement requires at least one catch or a finally clause.

Program 8:

```
class FinallyDemo
{
static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally

10
{
System.out.println("procA's finally");
}
}
```

```
static void procB()
{
try
{
System.out.println("inside procB");
return;
}
finally
{
System.out.println("procB's finally");
}
}
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();
}
}
```

Java's Built-in Exceptions

- Inside the standard package java.lang.
- Java defines several exception classes.(unchecked Exceptions)

Unchecked Exceptions

Exception	Meaning
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.
IndexOutOfBoudsException	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.

Checked Exceptions

Exception	Meaning
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.

Creating our Own Exception Subclasses

- We can create our own exception types to handle situations specific to our applications.
- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).

• The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable.

• Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

Method	Description
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
Void printStackTrace()	Displays the stack trace.
Void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
Void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString()	Returs a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.

Program 9

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return “MyException[“ + detail + “]”;
}
}
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println(“Called compute(“ + a + “)”);
if(a > 10)
throw new MyException(a);
System.out.println(“Normal exit”);
}
public static void main(String args[]) {
try {
compute(1);
compute(20);}
catch (MyException e){
System.out.println(“Caught “ + e);
}
```

}
}

UNIT IV

Multi-Threading

• 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.

Types of Multitasking

There are two distinct types of multitasking:

_Process-based Multitasking

_Thread-based Multitasking

Process-based Multitasking

• 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.

• Ex: Process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.

Thread-based Multitasking

• 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.

• Ex: 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.

Multitasking Threads Multitasking Processes

Light weight tasks Heavyweight tasks

Context switching- Not Costly Context switching – Costly

Share the same address space Share the different address space

Less Idle time in CPU More Idle time in CPU

The Java Thread Model

• The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

• Java uses threads to enable the entire environment to be asynchronous.

• This helps reduce inefficiency by preventing the waste of CPU cycles.

Single-Threaded system

• Single-threaded systems use an approach called an event loop with polling.

• In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.

• Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.

• Until this event handler returns, nothing else can happen in the system.

• This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed.

• In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

Java's Multithreading System

• The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated.

• One thread can pause without stopping other parts of your program.

• For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.

• Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.

• When a thread blocks in a Java program, only the single thread that is blocked pauses.

• All other threads continue to run.

Threads

Threads exist in several states.

_ Running

_ Ready to Run

_ Suspended

_ Resumed

_ Blocked

_ Terminated

• A thread can be running.

• It can be ready to run as soon as it gets CPU time.

• A running thread can be suspended, which temporarily suspends its activity.

• A suspended thread can then be resumed, allowing it to pick up where it left off.

• A thread can be blocked when waiting for a resource.

• At any time, a thread can be terminated, which halts its execution immediately.

• Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface

• Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.

• To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. They are...

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.

The Main Thread

• When a Java program starts up, one thread begins running immediately.

• This is usually called the main thread of your program, because it is the one that is executed when your 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 your 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 here:

static Thread currentThread( )

Program 1

```
class CurrentThreadDemo
{
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
t.setName("My Thread");
System.out.println("After name change: " + t);
try
{
```

```
for(int n = 5; n > 0; n--)
{
System.out.println(n);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
}
}
```

Sleep()

• The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

Its general form is shown here:

static void sleep(long milliseconds) throws InterruptedException

Creating a Thread

• You create a thread 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.

• You can construct a thread on any object that implements Runnable.

• To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run( )

• run( ) establishes the entry point for another, concurrent thread of execution within your program.

• This thread will end when run( ) returns.

_ Thread defines several constructors.

The one that we will use is:

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 you call its start( ) method, which is declared within Thread.

• In essence, start( ) executes a call to run( ).

• The start( ) method is shown here:

void start( )

Program 2

```
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.");
}
}
```

Extending 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

Program 3

```
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.");
}
}
```

Creating Multiple Threads

- Till now we have seen a Main Thread & one Child Thread.
- It is also possible to create multiple Threads.
- The following program demonstrates creation of multi-Threads.

Program 4

```
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 MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Using isAlive( ) and join( )

- How can one thread know when another thread has ended?
- Fortunately, Thread provides a means by which you can answer this question.
- Two ways exist to determine whether a thread has finished.

Alive()

- First, you can call isAlive( ) on the thread.
- This method is defined by Thread, and its general form is shown here:

final boolean isAlive( )

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

Join()

- The method that you will more commonly use 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.

Program 5

```
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.");
}
}
```

Thread Priorities

• Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

• A higher-priority thread can also preempt a lower-priority one.

• For Example, 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, use the setPriority( ) method, which is a member of Thread. This is its general form:

final void setPriority(int level)

• 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.

Program 6

```
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);
}
```

```
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
}
}
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);
lo.start();
hi.start();
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);
}
}
```

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.

• Key to synchronization is the concept of the monitor (also called a semaphore).

Monitor

• A monitor is an object that is used as a mutually exclusive lock, or mutex.

• Only one thread can own a monitor at a given time.

• When a thread acquires a lock, it is said to have entered the monitor.

• All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

• These other threads are said to be waiting for the monitor.

• A thread that owns a monitor can reenter the same monitor if it so desires.

Java implements synchronization through language elements in either one of the two ways.

1. Using Synchronized Methods
2. The synchronized Statement

Using Synchronized Methods

Program 7(unsynchronized Thread)

```
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");
}
}
}
```

The synchronized Statement

- Second solution is to call the methods defined by the class inside a synchronized block.
- This is the general form of the synchronized statement:

```
synchronized(object)
{
// statements to be synchronized
}
```

Program 8

```
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");
```

```
}
}
}
```

Interthread Communication

• To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.

_ 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.

These methods are declared within Object, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Program 9 (Correct Implementation of Producer & Consumer Problem)

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);


valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {
wait();
```

```
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
```

```
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Deadlock

• A special type of error that you need to avoid that relates specifically to multitasking is “deadlock”.

• It occurs when two threads have a circular dependency on a pair of synchronized objects.

_Deadlock is a difficult error to debug for two reasons:

■ In general, it occurs only rarely, when the two threads time-slice in just the right way.

■ It may involve more than two threads and two synchronized objects.

Suspending, Resuming, and Stopping Threads

• suspend( ) and resume( ), which are methods defined by Thread, to pause and restart the execution of a thread. They have the form shown below:

```
final void suspend( )
final void resume( )
```

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

```
final void stop( )
```

I/O Basics

• In fact, aside from print( ) and println( ), none of the I/O methods have been used significantly.

• The reason is simple: most real applications of Java are not text-based, console programs.

• Rather, they are graphically oriented applets that rely upon Java’s Abstract Window Toolkit (AWT) for interaction with the user.

Streams

• Java programs perform I/O through streams.

• A stream is an abstraction that either produces or consumes information.

• A stream is linked to a physical device by the Java I/O system.

• All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.

• This means that an input stream can abstract many different kinds of input: from a disk file, a

keyboard, or a network socket.

• Likewise, an output stream may refer to the console, a disk file, or a network connection.

• Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

• Java 2 defines two types of streams:

_Byte &

_Character.

• Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

• Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

The Byte Stream Classes

• Byte streams are defined by using two class hierarchies.

• At the top are two abstract classes:

_ InputStream &

_ OutputStream.

• Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An inut stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a compination of two or more input streams that will be read sequentially, one after the other.

The Character Stream Classes

• Character streams are defined by using two class hierarchies.

• At the top are two abstract classes,

_Reader &

_Writer.

• These abstract classes handle Unicode character streams.

• The abstract classes Reader and Writer define several key methods that the other stream classes implement.

• Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively.

• These methods are overridden by derived stream classes.

CHARACTER STREAM CLASSES

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print() and println()
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Predefined Streams

• All Java programs automatically import the java.lang package.

• This package defines a class called System, which encapsulates several aspects of the run-time environment.

• System also contains three predefined stream variables,

_ in,

_ out, &

_ err.

• These fields are declared as public and static within System.

• This means that they can be used by any other part of your program and without reference to a specific System object.

• System.out refers to the standard output stream. By default, this is the console.

• System.in refers to standard input, which is the keyboard by default.

• System.err refers to the standard error stream, which also is the console by default.

• However, these streams may be redirected to any compatible I/O device.

• System.in is an object of type InputStream;

• System.out and System.err are objects of type PrintStream.

• These are byte streams, even though they typically are used to read and write characters from and

to the console.

Using Byte Streams

• Reading Console Input:

• In Java, console input is accomplished by reading from System.in.

• To obtain a character-based stream that is attached to the console, you wrap System.in in a BufferedReader object, to create a character stream.

• BuffereredReader supports a buffered input stream.

• Its most commonly used constructor is shown here:

_BufferedReader(Reader inputReader)

• Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created.

• Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters.

• To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

_InputStreamReader(InputStream inputStream)

• Because System.in refers to an object of type InputStream, it can be used for inputStream.

• Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

_BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

• After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters:

• To read a character from a BufferedReader, use read( ).

• The version of read( ) that we will be using is

_int read( ) throws IOException

• Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value.

• It returns –1 when the end of the stream is encountered.

Program 1

```
import java.io.*;
class BRRead {
public static void main(String args[])
throws IOException
{
char c;
BufferedReader br = new
```

```
BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
c = (char) br.read();
System.out.println(c);
} while(c != 'q');
}
}
```

Reading Strings:

• To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is

_String readLine( ) throws IOException

Program 2

```
import java.io.*;
class BRReadLines {
public static void main(String args[])
throws IOException
{
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
System.out.println("Enter lines of text.");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

Writing Console Output

• Console output is most easily accomplished with print( ) and println( ). These methods are defined by the class PrintStream.

• PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ).

• Thus, write( ) can be used to write to the console.

• The simplest form of write() defined by PrintStream is

_void write(int byteval)

Program 3

```
class WriteDemo {
public static void main(String args[]) {
int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
}
}
```

PrintWriter Class

• PrintWriter is one of the character-based classes.

• Using a character-based class for console output makes it easier to internationalize your program.

• PrintWriter defines several constructors. The one we will use is shown here:

_PrintWriter(OutputStream outputStream, boolean flushOnNewline)

• outputStream is an object of type OutputStream, and flushOnNewline controls whether Java flushes the output stream every time a println( ) method is called.

• If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.

Program 4

```
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d);
}
}
```

Reading and Writing Files using Byte Streams

• Java provides a number of classes and methods that allow you to read and write files.

• In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

• However, Java allows you to wrap a byte-oriented file stream within a character-based object.

• Two of the most often-used stream classes are

_FileInputStream &

_FileOutputStream,

• which create an object of one of these classes, specifying the name of the file as an argument to

• the constructor.

• both classes support additional, overridden constructors, the following are the forms that we will be using:

FileInputStream(String filename)throws FileNotFoundException

FileOutputStram(String filename throws FileNotFoundException

• When you are done with a file, you should close it by calling close( ).

• It is defined by both FileInputStream and FileOutputStream, as shown here:

_void close( ) throws IOException

• To read from a file, you can use a version of read( ) that is defined within FileInputStream.

_int read( ) throws IOException

To Show a Text file

Program 5

```
import java.io.*;
class ShowFile {
public static void main(String args[])
throws IOException
{
int i;
FileInputStream fin;
try {
fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("File Not Found");
return;
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Usage: ShowFile File");
return;
}
// read characters until EOF is encountered
do {
i = fin.read();
if(i != -1) System.out.print((char) i);
```

```
} while(i != -1);
fin.close();
}
}
```

To Copy a Text file

Program 6

```
import java.io.*;
class CopyFile {
public static void main(String args[])
throws IOException
{
int i;
FileInputStream fin;
FileOutputStream fout;
try {
// open input file
try {
fin = new FileInputStream(args[0]);
} catch(FileNotFoundException e) {
System.out.println("Input File Not Found");
return;
}
// open output file
try {
fout = new FileOutputStream(args[1]);
} catch(FileNotFoundException e) {
System.out.println("Error Opening Output File");
return;
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Usage: CopyFile From To");
return;
}
// Copy File
try {
do {
i = fin.read();
```

```
if(i != -1) fout.write(i);
} while(i != -1);
} catch(IOException e) {
System.out.println("File Error");
}
fin.close();
fout.close();
}
}
```

File I/O using character streams

FileReader:

- The FileReader class creates a Reader that you can use to read the contents of a file.
- Its two most commonly used constructors are shown here:

_FileReader(String filePath)

_FileReader(File fileObj)

Program 7

```
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)

Program 8

```
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();
FileWriter f2 = new FileWriter("file3.txt");
f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
f2.close();
}
}
```

APPLETS

Applet

• Definition : 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.

Applet Example

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("A Simple Applet", 20, 20);
}
}
```

• An Applet begins with two import statements.

1 _ The first imports the Abstract Window Toolkit (AWT) classes.

– Applets interact with the user through the AWT, not through the console-based I/O classes.

– The AWT contains support for a window-based, graphical interface.

2 _ The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet.

After Import Stmts.....

_Class :

• The class must be declared as public, because it will be accessed by code that is outside the program.

_ Paint()

• Inside every class a paint() method is defined by the AWT and this must be overridden by the applet.

• paint( ) is called each time that the applet must redisplay its output.

• paint() is also called when the applet begins execution.

• Whatever the cause, whenever the applet must redraw its output, paint( ) is called.

• 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.

_ drawString() :

• Inside paint() there 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.

Note

• Applet does not have a main( ) method.

• Unlike Java programs, applets do not begin execution at main( ).

• In fact, most applets don't even have a main( ) method.

• Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

Running an Applet Program

• Running SimpleApplet involves a different process.

• In fact, there are two ways in which you can run an applet:

■ Executing the applet within a Java-compatible Web browser.

■ Using an applet viewer, such as the standard SDK tool, appletviewer.

An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

HTML code

• To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.

• SampleApplet:

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

• The width and height statements specify the dimensions of the display area used by the applet.

A Simple Applet Program

```
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);
}
}
```

Applet Architecture

• An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs.

1 _ First, applets are event driven.

• An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet.

• Once this happens, the applet must take appropriate action and then quickly return control to the AWT.

• This is a crucial point.

2 _ Second, the user initiates interaction with an applet—not the other way around.

• As you know, in a nonwindowed program, when the program needs input, it will prompt the user and then call some input method, such as readLine( ).

• This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants.

• These interactions are sent to the applet as events to which the applet must respond.

An Applet Skeleton

• Applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.

• Four of these methods are

_ init( ),

_ start( ),

_ stop( ), &

_ destroy( )

are defined by Applet.

Another,

_ paint( )

is defined by the AWT Component class.

Applet Initialization and Termination

• It is important to understand the order in which the various methods shown in the skeleton are called.

• When an applet begins, the AWT calls the following methods, in this sequence:

1. init( )

2. start( )

3. paint( )

• When an applet is terminated, the following sequence of method calls takes place:

1. stop( )

2. destroy( )

_init( )

• The init( ) method is the first method to be called.

• This is where you should initialize variables.

• This method is called only once during the run time of your applet.

_start( )

• The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped.

• Whereas init( ) is called once—the first time an applet is loaded,start( ) is called each time an applet's HTML document is displayed onscreen.

• So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

_paint( )

• The paint( ) method is called each time your applet's output must be redrawn.

• This situation can occur for several reasons. The paint( ) method has one parameter of type

Graphics.

• This parameter will contain 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.

_stop( )

• The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example.

• When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible.

• You can restart them when start( ) is called if the user returns to the page.

_destroy( )

• The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory.

• At this point, you should free up any resources the applet may be using.

• The stop( ) method is always called before destroy( ).

Program 1 (Applet Skeleton)

```
import java.awt.*;
import java.applet.*;
/*<applet code="AppletSkel" width=300 height=100> </applet> */
public class AppletSkel extends Applet
{
// Called first.
public void init()
{
// initialization
}
/* Called second, after init(). Also called whenever the applet is restarted. */
public void start()
{
// start or resume execution
}
// Called when the applet is stopped.
public void stop()
{
// suspends execution
}
/* Called when applet is terminated. This is the last method executed. */
```

```
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

Simple Applet Display Methods

- To set the background color of an applet's window, use setBackground( ).
- To set the foreground color, use setForeground( ).
- These methods have the following general forms:

void setBackground(Color newColor)

void setForeground(Color newColor)

- newColor specifies the new color.
- The class Color defines the constants shown here that can be used to specify colors:

Color.black Color.magenta
Color.blue Color.orange
Color.cyan Color.pink
Color.darkGray Color.red
Color.gray Color.white
Color.green Color.yellow
Color.lightGray

Examples:

```
setBackground(Color.green);
setForeground(Color.red);
```

Repainting

- As a general rule, an applet writes to its window only when its update( ) or paint( ) method is called by the AWT.
- But, how can the applet itself cause its window to be updated when its information changes?
- It cannot create a loop inside paint( ) that repeatedly scrolls
- Solution is : whenever your applet needs to update the information displayed in its window, it simply calls repaint( ).
- The repaint( ) method is defined by the AWT.
- It causes the AWT run-time system to execute a call to your applet's update( ) method, which, in its default implementation, calls paint( ).

• Thus, for another part of your applet to output to its window, simply store the output and then call repaint( ).

• The repaint( ) method has four forms.

1_ void repaint( )

• This version causes the entire window to be repainted.

2_ void repaint(int left, int top, int width, int height)

• This version specifies a region that will be repainted

• Here, the coordinates of the upper-left corner of the region are specified by left and top, and the width and height of the region are passed in width and height.

• These dimensions are specified in pixels. You save time by specifying a region to repaint.

• Calling repaint( ) is essentially a request that your applet be repainted sometime soon.

• However, if your system is slow or busy, update( ) might not be called immediately.

• This can be a problem in many situations, including animation, in which a consistent update time is necessary.

• One solution to this problem is to use the following forms of repaint( ):

3_ void repaint(long maxDelay)

4_ void repaint(long maxDelay, int x, int y, int width, int height)

• Here, maxDelay specifies the maximum number of milliseconds that can elapse before update( ) is called.

Program 2

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/
public class SimpleBanner extends Applet implements Runnable {
String msg = " A Simple Moving Banner.";
Thread t = null;
int state;
boolean stopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
```

```
public void start() {
t = new Thread(this);
stopFlag = false;
t.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char ch;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}
}
```

Status Window

• In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.

• To do so, call showStatus( ) with the string that you want displayed.

• The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors.

Program 3

```
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);


showStatus("This is shown in the status window.");
}
}
```

Passing Parameters to Applets

- APPLET tag in HTML allows you to pass parameters to your applet.
- To retrieve a parameter, use the getParameter() method.

Program 4

```
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);
}
}
```