UNIT-I

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. This language was initially called “Oak,” but was renamed “Java” in 1995.

C++ Vs. Java

Si. No.	C++	Java
1.	C++ is platform-dependent.	Java is platform-independent.
2.	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications.
3.	C++ is an extension/superset of C programming language.	Java was designed and created as a true object-oriented programming language
4.	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
5.	C++ supports operator overloading.	Java doesn't support operator overloading.
6.	C++ supports pointers.	Java does not support pointers in programs
7.	C++ uses compiler only. C++ source code is compiled into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Code written in Java is first compiled to bytecode by a program called *javac* . Then another program called *java* starts the Java runtime environment and it may compile and/or interpret the bytecode by using the Java Interpreter/JIT Compiler.
8.	C++ supports structures and unions.	Java doesn't support structures and unions.
9.	C++ doesn't have built-in support for threads. It relies on	Java has built-in thread support.

	third-party libraries for thread support.	
10.	C++ supports header files	Java does not support header files. Java uses the import keyword to include different packages, classes and methods.
11.	C++ uses delete operator to reclaim objects that are no longer required.	Java supports Garbage collector to reclaim objects that no longer hold any reference
12.	C++ supports global variables	Java does not support global variables

The Bytecode

- 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).*
- The original JVM was designed as an *interpreter for bytecode*.
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because 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.
- Although the details of the JVM differ from platform to platform, all understand the same Java bytecode.
- Execution of a Java program by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.
- Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.
- Although Java was designed as an interpreted language, Sun supplied HotSpot that provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected

portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis.

- A JIT compiler compiles code as it is needed, during execution. Only the code sequences that benefit from compilation are compiled. The remaining code is simply interpreted.
- 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 JVM is still in charge of the execution environment as shown in figure 1.

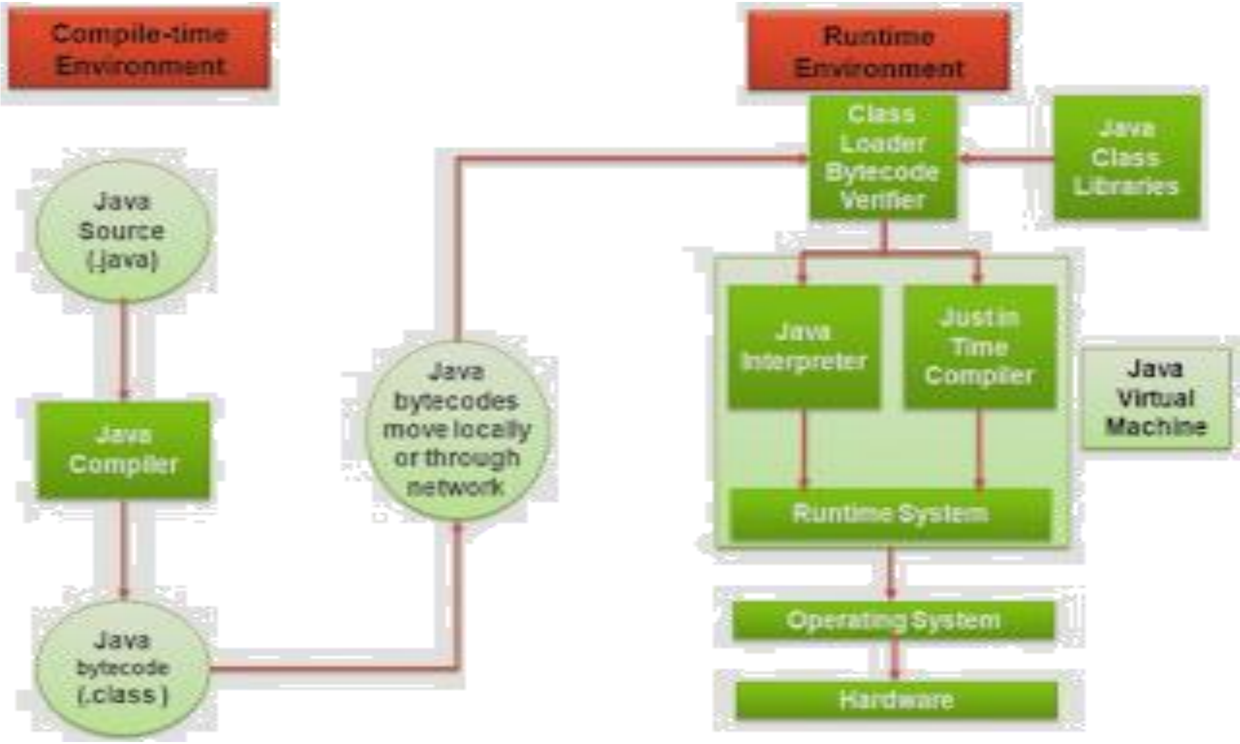


Figure1: Execution Environment of Java Program

The Java Buzzwords

The key considerations that played an important role in molding the language were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable

- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy to learn and use effectively. Java inherits the C/C++ syntax

Security

Java is said to be more secure programming language because **it does not have pointers concept**. Java provides a feature "applet" which can be embedded into a web application. The applet in java does not allow access to other parts of the computer, which keeps away from harmful programs like viruses and unauthorized access.

Portability

Portability is one of the core features of java which enables the java programs to run on any computer or operating system. For example, an applet developed using java runs on a wide variety of CPUs, operating systems, and browsers connected to the Internet.

Object-Oriented

Java is true Object Oriented language. Almost everything in Java is Object. Java has an extensive set of classes that are available to programmers in packages. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

Java is a robust language that provides many safeguards to ensure reliable code.

- It has strict compile time and run-time checking for data types.
- Java is designed as a garbage-collecting language relieving the programmers from memory management problems.
- The absence of pointers in Java ensures that the programs cannot gain access to memory locations without proper authorization.
- Java also incorporates the concept of object-oriented exception handling which captures serious errors and eliminates risk of system crash.

Multithreaded

Java supports multi-threading programming, which allows us to write programs that do multiple operations simultaneously.

Architecture-Neutral

Java has invented to archive "write once; run anywhere, any time, forever". The java provides JVM (Java Virtual Machine) to archive architectural-neutral or platform-independent. The JVM allows the java program created using one operating system can be executed on any other operating system.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI).* This feature enables a program to invoke methods across a network

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

Syllabus begins from here for unit I

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented.

Two Programming Paradigms

All computer programs consist of two elements: code and data. A program can be conceptually organized around its **code or around its data**. That is, some programs are written around "**what is happening**" and others are written around "**who is being affected**". These are the two paradigms that govern how a program is constructed.

- ***process-oriented model(procedure-oriented model)***

- This approach characterizes a program as a series of linear steps (that is, code).
- The process-oriented model can be thought of as ***code acting on data***.
- Procedural languages such as C employ this model
- Top-down approach.
- Difficult to manage as programs grow larger and more complex.

- ***object-oriented programming***
- Developed to manage increasing complexity.
- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized **as *data controlling access to code***.
- OOP offers several organizational benefits.
- Bottom-up approach.

Abstraction

An essential element of object-oriented programming is *abstraction.* Abstraction refers to the act of representing essential features without including background details.

For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. For example, from the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are **encapsulation, inheritance**, and **polymorphism**.

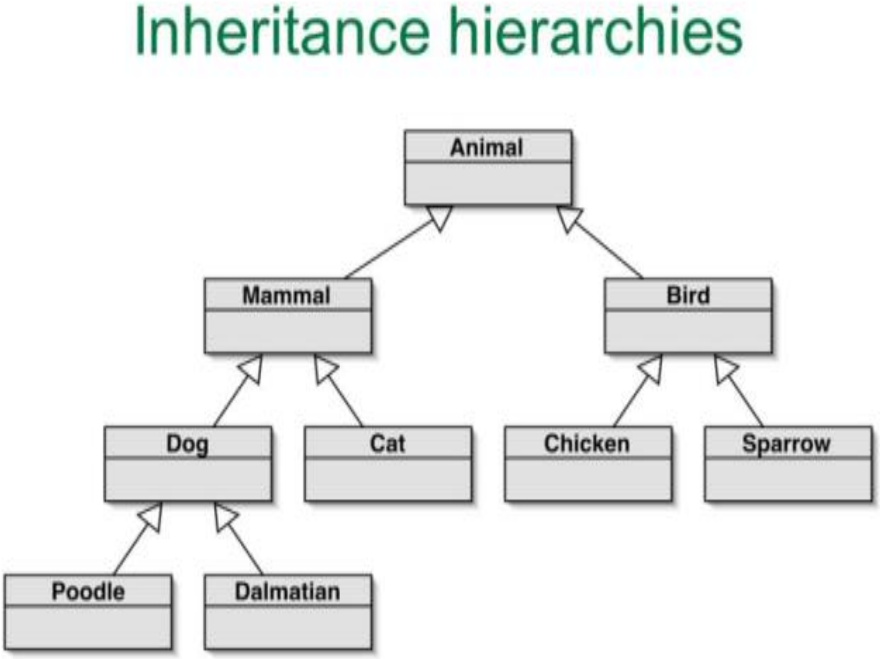
Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. Encapsulation is like a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

In Java, the basis of encapsulation is the class. A ***class*** defines the attributes/properties and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the properties and behavior defined by the class. **Objects** are sometimes referred to as *instances of a class*. Thus, a class is a logical construct; an object has physical reality. When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables*. The code that operates on that data is referred to as *member methods* or just *methods*.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of **hierarchical classification** and . Most knowledge is made manageable by hierarchical classifications. For example, a Dalmatian is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal*. By use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent.
In inheritance hierarchy shown below Animal class is the *superclass* for other classes. Other classes are subclasses of the superclass animal.



Polymorphism

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase **“one interface, multiple methods.”** This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler’s job to select the *specific action* (that is, method) as it applies to each situation.

Example: A dog’s sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

A First Simple Program

```
/*This is a simple Java program. Call this file "Example.java".*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Entering the Program

In Java, a source file is officially called a *compilation unit.* It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension. e.g. Example.java. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. The Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, you must use the Java application launcher, called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following **output** is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute **java** as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

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

This line begins the **main( )** method. This is the line at which the program will begin executing. The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.

The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made.

The keyword **void** simply tells the compiler that **main( )** does not return a value.

In **main( )**, there is only one parameter, **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

One other point: **main( )** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main( )** method to get things started.

System.out.println("This is a simple Java program.");

This line outputs the string “This is a simple Java program.” followed by a new line on the screen.
Java **System.out.println()** is used to print an argument that is passed to it. The statement can be broken into 3 parts which can be understood separately as:

- **System:** It is a final class defined in the java.lang package.
- **out:** This is an instance of PrintStream type, which is a public and static member field of the System class.
- **println()**: As all instances of PrintStream class have a public method println(), hence we can invoke the same on out as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. We can assume that System.out represents the Standard Output Stream.

Class Fundamentals

A Class defines a new datatype with its own properties and behavior. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

A class is declared by use of the **class** keyword. A simplified general form of a **class** definition is shown here:

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

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

A Simple Class Example

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

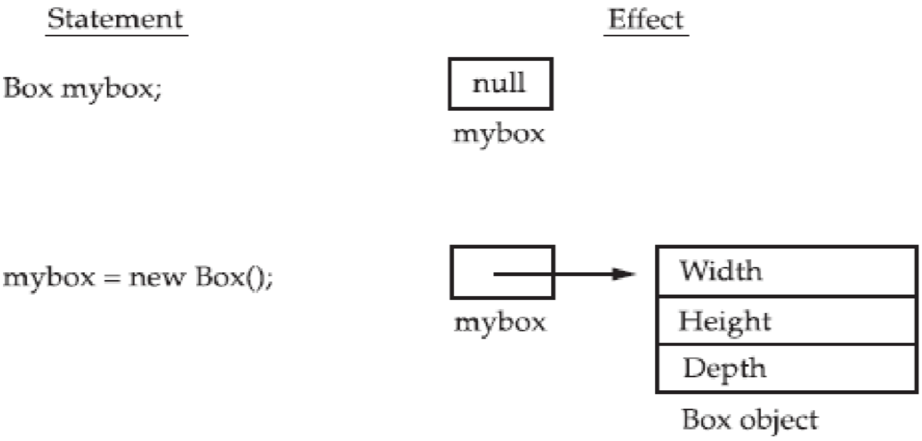
output:

```
Volume is 3000.0
```

A Closer Look at new

The **new** operator dynamically allocates memory for an object. It has this general form:

***class-var* = new *classname*( );**



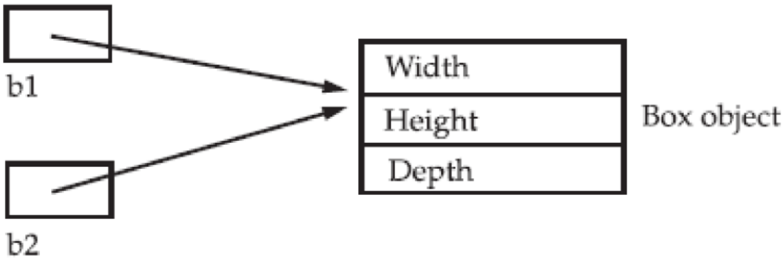
Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class.

Assigning Object Reference Variables

Object reference variables act differently when an assignment takes place. For example,

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

Here **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.

Introducing Methods

This is the general form of a method:

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

//Adding a Method to the Box Class

```
class Box1 {
double width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
class BoxDemo3 {
public static void main(String args[]) {
Box1 mybox1 = new Box1();
Box1 mybox2 = new Box1();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

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

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

Returning a Value

```
// Now, volume() returns the volume of a box.
class Box2 {
double width;
double height;
double depth;
// compute and return volume
double volume() {
```

```
return width * height * depth;
}
}
class BoxDemo4 {
public static void main(String args[]) {
Box2 mybox1 = new Box2();
Box2 mybox2 = new Box2();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Adding a Method That Takes Parameters

```
// This program uses a parameterized method.
class Box3{
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box3 mybox1 = new Box3();
Box3 mybox2 = new Box3();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Constructors

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.

```
/* Here, Box uses a constructor to initialize the dimensions of a box.*/
class Box4 {
double width;
double height;
double depth;
// This is the constructor for Box.
Box4() {
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
Box4 mybox1 = new Box4();
Box4 mybox2 = new Box4();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

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

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

Parameterized Constructors

```
/* 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
```

The this Keyword(uses)

1) **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked.

```
// A redundant use of this.
Box(double w, double h, double d) {
```

```
this.width = w;
this.height = h;
this.depth = d;
}
```

Inside **Box( )**, **this** will always refer to the invoking object.

2)Instance Variable Hiding

you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

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

3) To return invoking object from a method

```
   Example:    //method in Box class returning Box class object
                private Box   setdim(double w, double h, double d)
                {
                   width=w; height=h;depth=d;
                   return this;}
```

Garbage Collection

In C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs sporadically (at irregular intervals) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The 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

A Stack Class

```
// This class defines an integer stack that can hold 10 values.
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
```

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java supports polymorphism.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + "" + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

A Closer Look at Argument Passing

```
// Primitive types are passed by value.
class Test {
void fun(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.fun(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
```

// 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;
}
}
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 20
```

Example 1

```
package UNIT1;
//swap 2 objects
class number
```

```
{ private double n;
  number(double a)
  {
            n=a;
  }
  void swap(number n1, number n2)
  {

            double  temp=n1.n;
            n1.n=n2.n;
            n2.n=temp;
  }
  void display()
  {System.out.println(n);
  }
}
public class swapDemo {
public static void main(String[] args)
{
          number x =new number(10);
          number y =new number(20);
          System.out.println("before swap");
          x.display();     y.display();
          x.swap(x,y);
          System.out.println("After swap");
          x.display();     y.display();
}
}
```

Output

```
before swap
10.0
20.0
After swap
20.0
10.0
```

Returning Objects

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

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

The output generated by this program is shown here:

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

Understanding static

When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.

You can declare both methods and variables to be **static**.

The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

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

Methods declared as **static** have several restrictions:
- They can only directly call other **static** methods.
- They must only directly access **static** data.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization
block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

***classname.method*( )**

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

Arrays

Arrays are implemented as objects. Because of this, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.
class Length {
public static void main(String args[]) {
int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
}
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

==

Example1

```
/*Design a student class with 2 instance variables Usn and name, a constructor and a method
to print student object. Write driver class to test an   array of student class of size N*/
package UNIT1;
import java.util.Scanner;
class student{
    String USN;
    String name;
        student(String u, String n)
    {
        USN= u;
        name = n;
    }
        void PrintStudent() {
        System.out.println("USN:"+ USN);
        System.out.println("Name:"+ name);
        }
}
class studentArray {
        public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("enter n value");
        int n=sc.nextInt();
    student[] s=new student[n];
    sc.nextLine();
    for(int i=0; i<s.length;i++){
        {
                System.out.println("enter usn");
                String usn=sc.nextLine();
                System.out.println("enter name");
                String name=sc.nextLine();
                s[i]=new student(usn,name);
        }
   }
   System.out.println("Student Details");
   for(int i=0; i<s.length;i++)
        {
                s[i].PrintStudent();
        }
   }
 }
```

Output

```
enter n value
2
enter usn
111
enter name
xxx
enter usn
222
enter name
rrr
Student Details
USN:111
Name:xxx
USN:222
Name:rrr
```

=========================

Exercise 2

```
package UNIT1;

//Java Program to Demonstrate Arrays Class--Binary search Via binarySearch() method

//Importing Arrays utility class from java.util package
import java.util.Arrays;
import java.util.Scanner;

//Main class
public class binSearch {
        // Main driver method
        public static void main(String[] args)
        {
                // Get the Array
                Scanner sc = new Scanner(System.in);
                System.out.println("enter n value");
                int n=sc.nextInt();
                int intArr[] = new int[n];
                System.out.println("enter array elements");
                for(int i=0; i<intArr.length;i++)
```

```
            intArr[i] =sc.nextInt();
        System.out.println("enter key value");
        int intKey=sc.nextInt();
        Arrays.sort(intArr);
        // Print the key and corresponding index
        int res=Arrays.binarySearch(intArr, intKey);
        if(res>=0)
                System.out.println(intKey + " found at index = "+ res);
        else
            System.out.println(intKey + " not found");
    }
}
```

Output

```
enter n value
4
enter array elements
30
20
80
60
enter key value
100
100 not found
==============
enter n value
5
enter array elements
60
20
40
10
50
enter key value
60
60 found at index = 4
==========================================================
```

Example 3

```
package UNIT1;

import java.util.Scanner;

/* Design a person class with instance variables name and age  and a constructor to initialize,
a compare method to compare 2 person objects and  return person object with greater age.
Write driver class to  find eldest person among N person objects*/
class Person{
    private String name;
    private int age;
    Person(String n, int a)
    { name=n;
    age=a;
    }
    Person compare(Person x)
    {
        if(age>x.age)
            return this;
        else
            return x;
    }
    void display() {
        System.out.println("Name:"+ name);
        System.out.println("Age:"+ age);
    }
}
public class comparePerson {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("enter n value");
        int n=sc.nextInt();
     Person[] p=new Person[n];
     for(int i=0; i<p.length;i++){
        {   sc.nextLine();
            System.out.println("enter name");
            String name=sc.nextLine();
            System.out.println("enter age");
            int age=sc.nextInt();
            p[i]=new Person(name,age);
        }
     }
     Person res=p[0];
     for(int i=1; i<p.length;i++)
       res=p[i].compare(res);
     System.out.println("Details of eldest person");
     res.display();
 }
}
```

Output

```
enter n value
3
enter name
xxx
enter age
45
enter name
yyy
enter age
23
enter name
zzz
enter age
12
```

```
Details of eldest person
Name:xxx
Age:45
```

==

Example 4

```java
package UNIT1;
//Java Program to sort an anonymous int array.
import java.util.Arrays;
public class testAnonymousArray{
//creating a method which receives an array as a parameter
static void sortAndprintArray(int arr[]){
Arrays.sort(arr);
for(int i=0;i<arr.length;i++)
		System.out.println(arr[i]);
}
public static void main(String args[]){
	System.out.println("Array elements");
	sortAndprintArray(new int[]{10,40,20,50,30});//passing anonymous array to method
}}
```

Output

```
Sorted Array
10
20
30
40
50
```

Exploring the String Class

Every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement System.out.println("This is a String, too"); the string “This is a String, too” is a **String** constant.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered.

Example:

```java
   class Testimmutablestring{
    public static void main(String args[]){
      String s="Nitte";
      s.concat(" Meenakshi");//concat() method appends the string at the end
      System.out.println(s);//will print Nitte because strings are immutable objects
    }
   }
```

While this may seem like a serious restriction, it is not, for two reasons:

1)If you need to change a string, you can always create a new one that contains the modifications.

Example:

```java
   class Testimmutablestring{
    public static void main(String args[]){
        String s="Nitte";
        s=s.concat(" Meenakshi");//concat() method appends the string at the end
        System.out.println(s);//will print Nitte because strings are immutable objects
      }
     }
```

2) Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java. Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

Example:

```
package UNIT1;
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Nitte");
System.out.println(sb);
sb.reverse();
System.out.println(sb);
           }
          }
```

Output

```
Nitte
ettiN
```

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings.

For example, this statement
String myString = "I" + " like " + "Java."; results in **myString** containing “I like Java.”

The following program demonstrates the preceding concepts:

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

The output produced by this program is shown here:

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

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:

boolean equals(String *object*)
int length( )
char charAt(int *index*)

Here is a program that demonstrates these methods:

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

This program generates the following output:

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

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

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

Here is the output from this program:

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

Methods of String class

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
10	boolean equals(Object another)	It checks the equality of string with the given object.
11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	`boolean equalsIgnoreCase(String anotherString)`	It compares another string. It doesn't check case.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
25	String toUpperCase()	It returns a string in uppercase.

27	String trim()	It removes beginning and ending spaces of this string.

Methods of StringBuffer class

Methods	**Action Performed**
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
capacity()	the total allocated capacity can be found by the capacity( ) method
delete()	Deletes a sequence of characters from the invoking object
deleteCharAt()	Deletes the character at the index specified by *loc*
ensureCapacity()	Ensures capacity is at least equals to the given minimum.
insert()	Inserts text at the specified index position
length()	Returns length of the string
reverse()	Reverse the characters within a StringBuffer object
replace()	Replace one set of characters with another set inside a StringBuffer object

Example programs

```
//create string
package UNITI;

public class createString{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

```
//immutable string
package UNITI;
class immutableString{
        public static void main(String args[]){
          String is="Sachin";
          is.concat(" Tendulkar");
          System.out.println(is);
          String s=is.concat(" Tendulkar");
          System.out.println(s);
        }
       }
```

Output:

```
Sachin
Sachin Tendulkar
```

```
package UNITI;
//Java code to illustrate different constructors and methods String class.
//import java.io.*;
//import java.util.*;
class stringFunctionTest
{
        public static void main (String[] args)
        {
                String s= "Nitte Meenakshi";// or String s= new String ("Nitte Meenakshi");

                // Returns the number of characters in the String.
                System.out.println("String length = " + s.length());

                // Returns the character at ith index.
                System.out.println("Character at 4th position = "
                                                + s.charAt(4));

                // Return the substring from the ith index character to end of string
                System.out.println("Substring " + s.substring(3));

                // Returns the substring from i to j-1 index.
                System.out.println("Substring = " + s.substring(2,5));

                // Concatenates string2 to the end of string1.
                String s1 = "nitte";
                String s2 = "college";
                System.out.println("Concatenated string = " +
                                                        s1.concat(s2));
                // Returns the index within the string
                // of the first occurrence of the specified string.
                String s4 = "Learn Share Learn";
                System.out.println("Index of Share " +
                                                s4.indexOf("Share"));

                // Returns the index within the string of the
                // first occurrence of the specified string,
                // starting at the specified index.
                System.out.println("Index of a = " +
                                                s4.indexOf('a',3));

                // Checking equality of Strings
                Boolean out = "Nitte".equals("nitte");
                System.out.println("Checking Equality " + out);
                out = "Nitte".equals("Nitte");
                System.out.println("Checking Equality " + out);

                out = "Nitte".equalsIgnoreCase("nitte");
                System.out.println("Checking Equality " + out);

                //If ASCII difference is zero then the two strings are similar
                int out1 = s1.compareTo(s2);
                System.out.println("the difference between ASCII value is="+out1);
                // Converting cases
                String word1 = "NiTTe";
                System.out.println("Changing to lower Case " +
                                            word1.toLowerCase());

                // Converting cases
                String word2 = "NiTtE";
                System.out.println("Changing to UPPER Case " +
                                            word2.toUpperCase());

                // Trimming the word
                String word4 = "    JAVA PROGRAMMING    ";
                System.out.println("Trim the word(REMOVE SPACE IN THE BEGINNING
AND END)  " + word4.trim());

                // Replacing characters
                String str1 = "MITTE";
                System.out.println("Original String " + str1);
                String str2 = str1.replace('M' ,'N') ;
                System.out.println("Replaced M with N -> " + str2);
        }
}
```

Output:

```
String length = 15
Character at 4th position = e
```

```
Substring te Meenakshi
Substring = tte
Concatenated string = nittecollege
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equality true
the difference between ASCII value is=11
Changing to lower Case nitte
Changing to UPPER Case NITTE
Trim the word(REMOVE SPACE IN THE BEGINNING AND END)  JAVA
PROGRAMMING
Original String MITTE
Replaced M with N -> NITTE
```

```
// StringBufferFunctions
package UNITI;
class StringBufferFunctions{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
//append()
sb.append("Java");//now original string is changed
System.out.println("Append result:"+sb);//prints Hello Java
//insert()
sb.insert(1,"Java");//now original string is changed
System.out.println("Insert result:"+sb);
sb.replace(1,9,"ello    ");
System.out.println("Replace result:"+sb);
}
}
```

Output:

```
Append result:Hello Java
Insert result:HJavaello Java
Replace result:Hello     Java
```

Exercise

1. Write a program to input 2 strings, concatenate and reverse the concatenated string

Using Command-Line Arguments

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
```

Varargs: Variable-Length Arguments

A method that takes a variable number of arguments is called a **variable-arity method**, or simply a **varargs method**.

First Method

// **Use an array to pass a variable number of arguments to a method. This is the old-style approach //to variable-length arguments**.

```
 class PassArray {
 static void vaTest(int v[]) {
 System.out.print("Number of args: " + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 public static void main(String args[])
 {
 // Notice how an array must be created to hold the arguments.
 int n1[] = { 10 };
 int n2[] = { 1, 2, 3 };
 int n3[] = { };
 vaTest(n1); // 1 arg
 vaTest(n2); // 3 args
 vaTest(n3); // no args
 }
}
```

The output from the program is shown here:

```
 Number of args: 1 Contents: 10
 Number of args: 3 Contents: 1 2 3
 Number of args: 0 Contents:
```

In the program, the method vaTest( ) is passed its arguments through the array v. This old-style approach to variable-length arguments does enable vaTest( ) to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling vaTest( ). Not only is it tedious to construct an array each time vaTest( ) is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

Second Method

A variable-length argument is specified by three periods (…). For example, here is how vaTest( ) is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that vaTest( ) can be called with zero or more arguments. As a result, v is implicitly declared as an array of type int[ ]. Thus, inside vaTest( ), v is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {
 // vaTest() now uses a vararg.
 static void vaTest(int ... v) {
 System.out.print("Number of args: " + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 public static void main(String args[])
 {
 // Notice how vaTest() can be called with a variable number of arguments.
 vaTest(10); // 1 arg
 vaTest(1, 2, 3); // 3 args
 vaTest(); // no args
 }
}
```

The output from the program is the same as the original version.

--A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the **last parameter** declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the vaTest( ) method that takes a regular argument and a variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {
 // Here, msg is a normal parameter and v is a varargs parameter.
 static void vaTest(String msg, int ... v) {
 System.out.print(msg + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 public static void main(String args[])
 {
 vaTest("One vararg: ", 10);
 vaTest("Three varargs: ", 1, 2, 3);
 vaTest("No varargs: ");
 }
}
```

The output from this program is shown here:

```
 One vararg: 1 Contents: 10
 Three varargs: 3 Contents: 1 2 3
 No varargs: 0 Contents:
```

Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads vaTest( ) three times:

```
// Varargs and overloading.
class VarArgs3 {
 static void vaTest(int ... v) {
 System.out.print("vaTest(int ...): " + "Number of args: " + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 static void vaTest(boolean ... v) {
 System.out.print("vaTest(boolean ...) " + "Number of args: " + v.length + " Contents: ");
 for(boolean x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 static void vaTest(String msg, int ... v) {
 System.out.print("vaTest(String, int ...): " + msg + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 public static void main(String args[])
 {
 vaTest(1, 2, 3);
 vaTest("Testing: ", 10, 20);
 vaTest(true, false, false);
 }
}
```

The output produced by this program is shown here:

```
 vaTest(int ...): Number of args: 3 Contents: 1 2 3
 vaTest(String, int ...): Testing: 2 Contents: 10 20
 vaTest(boolean ...) Number of args: 3 Contents: true false false
```

Varargs and Ambiguity

Ambiguous situation is one in which compiler is unable to choose one among several overloaded methods.

When overloading a method that takes a variablelength argument, it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

// Varargs, overloading, and ambiguity. **This program contains an error and will not compile!**

```
class VarArgs4 {
 static void vaTest(int ... v) {
 System.out.print("vaTest(int ...): " + "Number of args: " + v.length + " Contents: ");
 for(int x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 static void vaTest(boolean ... v) {
 System.out.print("vaTest(boolean ...) " + "Number of args: " + v.length + " Contents: ");
 for(boolean x : v)
 System.out.print(x + " ");
 System.out.println();
 }
 public static void main(String args[])
 {
vaTest(1, 2, 3); // OK
 vaTest(true, false, false); // OK
 vaTest(); // Error: Ambiguous!
 }
}
```

In this program, the overloading of vaTest( ) is perfectly correct. However, this program will not compile because of the following call:

vaTest(); // Error: Ambiguous!

Because the vararg parameter can be empty, this call could be translated into a call to vaTest(int …) or vaTest(boolean …). Both are equally valid. Thus, the call is inherently ambiguous.

--Here is **another example of ambiguity**. The following overloaded versions of vaTest( )are **inherently ambiguous** even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of vaTest( ) differ, there is no way for the compiler to resolve the following call:

vaTest(1)

Does this translate into a call to vaTest(int …), with one varargs argument, or into a call to vaTest(int, int …) with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution

Example

```
package UNIT1;
```

```
//sorting strings passed as varargs to a method
import java.util.Arrays;
class names {
        static void vaSort(String ...n) {
        System.out.println("Number of Names: " + n.length);
        System.out.print( "Names: ");
        for(String x : n)
        System.out.print(x + "     ");
        System.out.println();
        Arrays.sort(n);
        System.out.print( "Sorted Names: ");
        for(String x : n)
        System.out.print(x + "     ");
        System.out.println();
        }
}
class sortVarargs{
        public static void main(String args[]){
                names.vaSort("Nitte","Meenakshi");
                System.out.println("----------------------");
                names.vaSort("Nitte","Engineering","College");
          }
        }
```

```
Number of Names: 2
Names: Nitte     Meenakshi
Sorted Names: Meenakshi     Nitte
----------------------
Number of Names: 3
Names: Nitte     Engineering     College
Sorted Names: College     Engineering     Nitte
```