CMR Institute of Technology

Dept. ECE/CSE/Civil/EEE

Programming in Java VTU solution (18CS532)

1(a) Explain three OOPs principles

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. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting

this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, 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.* (If you are familiar with C/C++, it may help to know that what a Java programmer calls a *method*, a C/C++ programmer calls a *function.*) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The *public* interface of a class represents everything that external users of the class need to know, or may know. The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

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. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog*, which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, 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. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

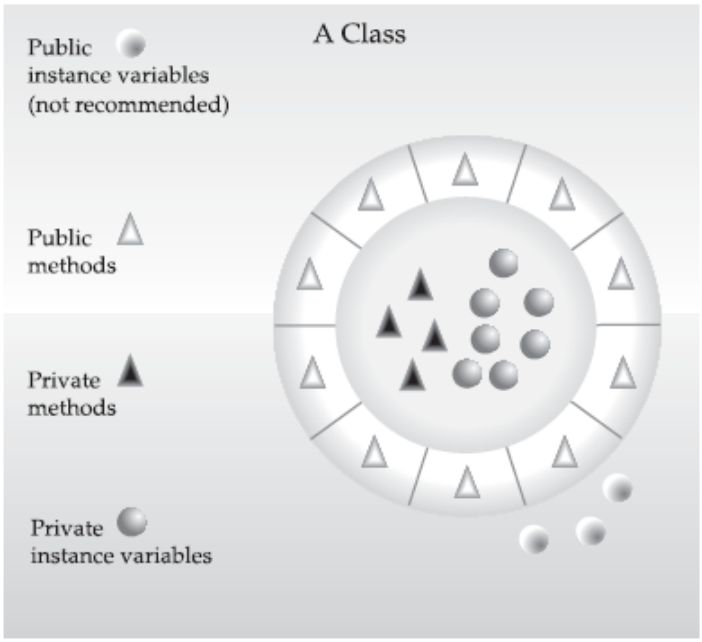
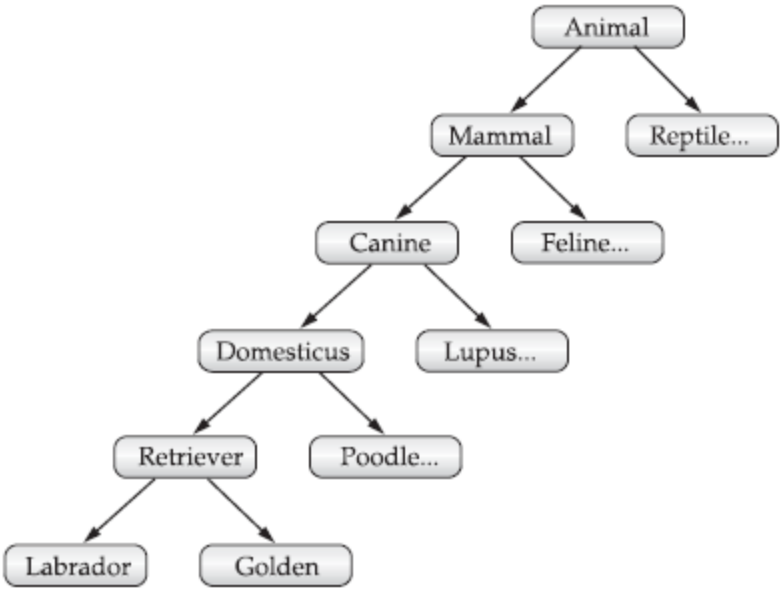


FIGURE 2-1
Encapsulation: public methods can be used to protect private data

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the *class* definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass.*

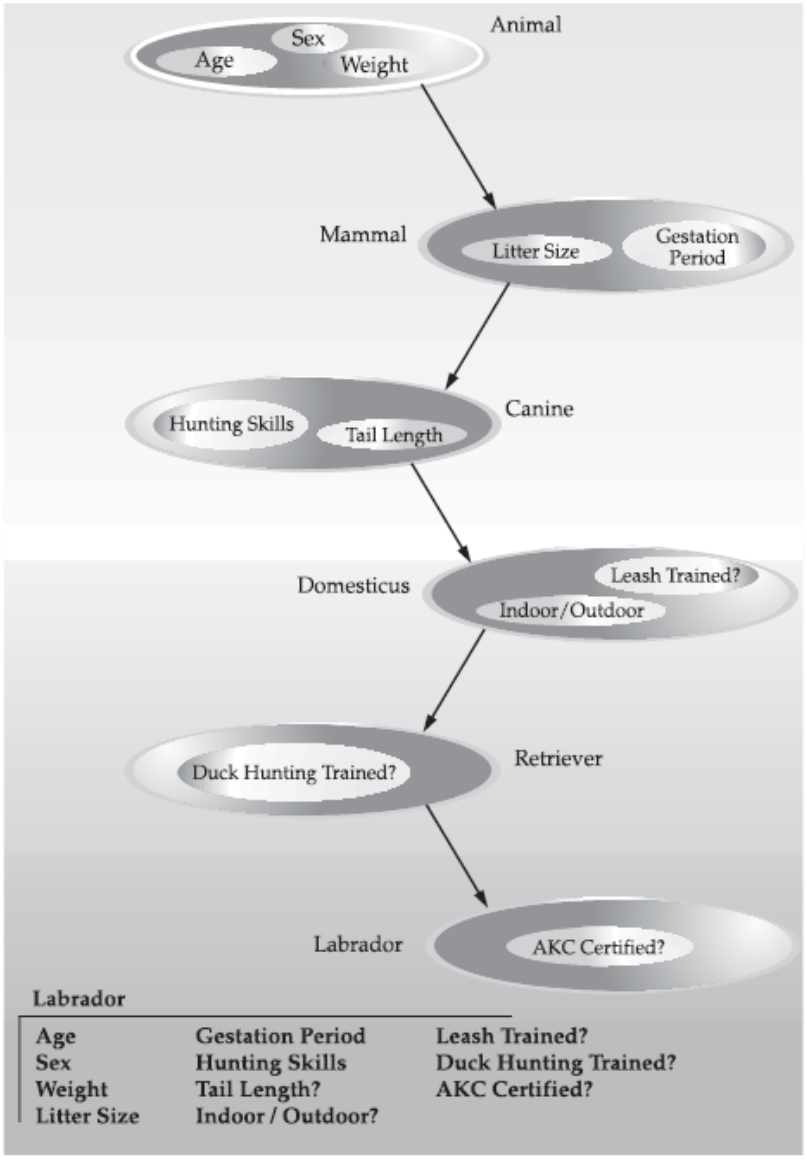
Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy.*



Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

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



1(b)Explain the process of compiling and running java application with example

Java, being a platform-independent programming language, doesn't work on the one-step compilation. Instead, it involves a two-step execution, first through an OS-independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system.

The two principal stages are explained below:

Principle 1: Compilation

First, the source '.java' file is passed through the compiler, which then encodes the source code into a machine-independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate '.class' file. While converting the source code into the bytecode, the compiler follows the following steps:

Step 1: Parse: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.

Step 2: Enter: Enters symbols for the definitions into the symbol table.

Step 3: Process annotations: If Requested, processes annotations found in the specified compilation units.

Step 4: Attribute: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.

Step 5: Flow: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.

Step 6: Desugar: Rewrites the AST and translates away some syntactic sugar.

Step 7: Generate: Generates '.Class' files.

https://www.youtube.com/watch?v=0f-Sx81bIWQ

Principle 2: Execution

The class files generated by the compiler are independent of the machine or the OS, which allows them to be run on any system. To run, the main class file (the class that contains the method main) is passed to the JVM and then goes through three main stages before the final machine code is executed. These stages are:

These states do include:

ClassLoader

Bytecode Verifier

Just-In-Time Compiler

Let us discuss all 3 stages.

Stage 1: Class Loader

The main class is loaded into the memory bypassing its '.class' file to the JVM, through invoking the latter. All the other classes referenced in the program are loaded through the class loader.

A class loader, itself an object, creates a flat namespace of class bodies that are referenced by a string name. The method definition is provided below illustration as follows:

Illustration:

```
// loadClass function prototype

Class r = loadClass(String className, boolean resolveIt);

// className: name of the class to be loaded
// resolveIt: flag to decide whether any referenced class should be loaded or not.
```

There are two types of class loaders

primordial

non-primordial

The primordial class loader is embedded into all the JVMs and is the default class loader. A non-primordial class loader is a user-defined class loader, which can be coded in order to customize the class-loading process. Non-primordial class loader, if defined, is preferred over the default one, to load classes.

Stage 2: Bytecode Verifier

After the bytecode of a class is loaded by the class loader, it has to be inspected by the bytecode verifier, whose job is to check that the instructions don’t perform damaging actions. The following are some of the checks carried out:

Variables are initialized before they are used.

Method calls match the types of object references.

Rules for accessing private data and methods are not violated.

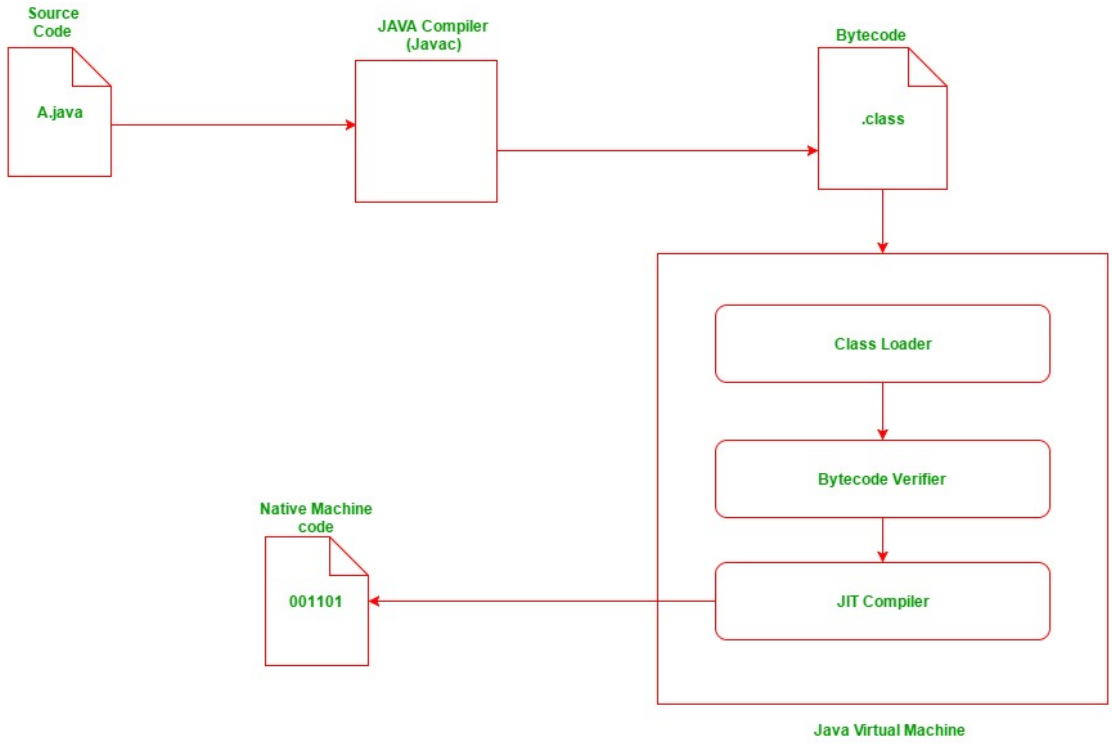
Local variable accesses fall within the runtime stack.

The run-time stack does not overflow.

If any of the above checks fail, the verifier doesn’t allow the class to be loaded.

Stage 3: Just-In-Time Compiler

This is the final stage encountered by the java program, and its job is to convert the loaded bytecode into machine code. When using a JIT compiler, the hardware can execute the native code, as opposed to having the JVM interpret the same sequence of bytecode repeatedly and incurring the penalty of a relatively lengthy translation process. This can lead to performance gains in the execution speed unless methods are executed less frequently.



The process can be well-illustrated by the following diagram as given above as follows from which we landed up to the conclusion.

Conclusion: Due to the two-step execution process described above, a java program is independent of the target operating system. However, because of the same, the execution time is way more than a similar program written in a compiled platform-dependent program.

1(c) Discuss various primitive data types in Java

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

boolean data type

byte data type

char data type

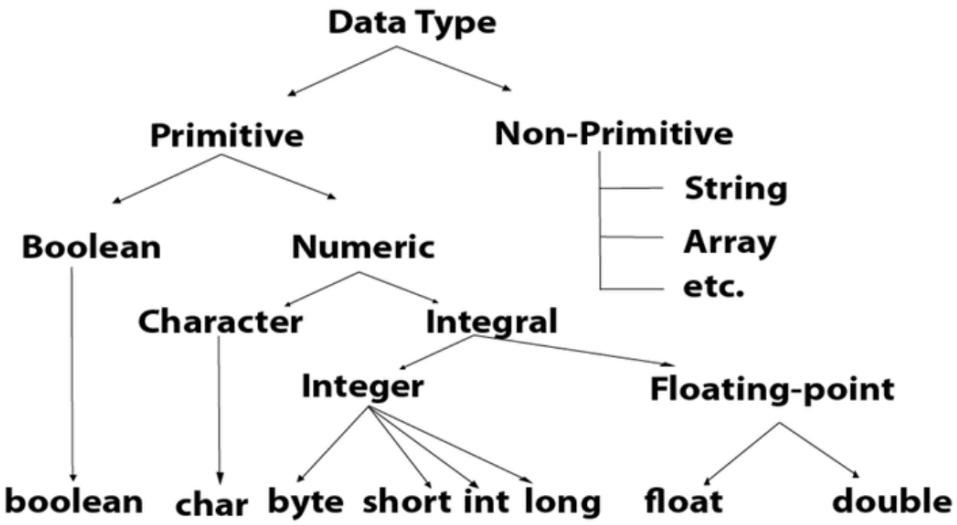
short data type

int data type

long data type

float data type

double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

char letterA = 'A'

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

2(a) How are arrays declared and initialized in Java. Explain with suitable example

Java Initialize array

Java initialize array is basically a term used for initializing an array in Java. We know that an array is a collection of similar types of data. The array is a very important data structure used for solving programming problems.

The word element is used for the values stored in different positions of the array. In order to use the Array data structure in our code, we first declare it, and after that, we initialize it.

Declaration of an Array

The syntax of declaring an array in Java is given below.

```
datatype [] arrayName;
```

Here, the datatype is the type of element that will be stored in the array, square bracket[] is for the size of the array, and arrayName is the name of the array.

Initializing an Array

Only the declaration of the array is not sufficient. In order to store values in the array, it is required to initialize it after declaration. The syntax of initializing an array is given below.

```
datatype [] arrayName = new datatype [ size ]
```

In Java, there is more than one way of initializing an array which is as follows:

1. Without assigning values

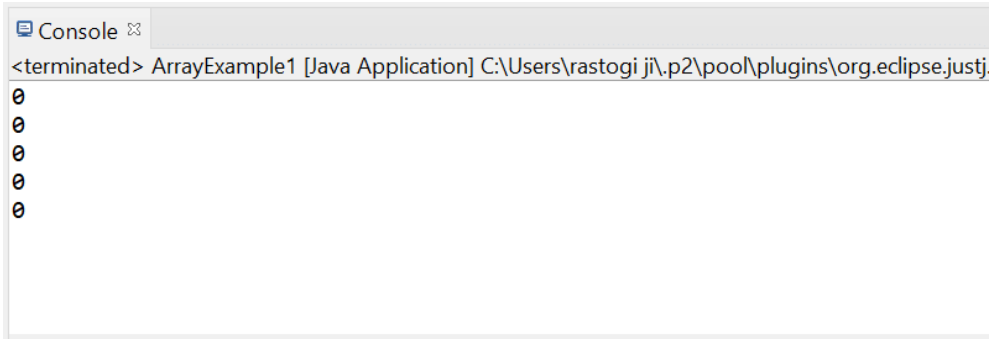
In this way, we pass the size to the square braces[], and the default value of each element present in the array is 0. Let's take an example and understand how we initialize an array without assigning values.

ArrayExample1.java

```
public class ArrayExample1 {
```

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

        //initializing array without passing values
        int[] array = new int[5];

        //print each element of the array
        for (int i = 0; i < 5; i++)
        {
            System.out.println(array[i]);
        }
    }

}
```

Output:



Java Initialize array

2. After the declaration of the array

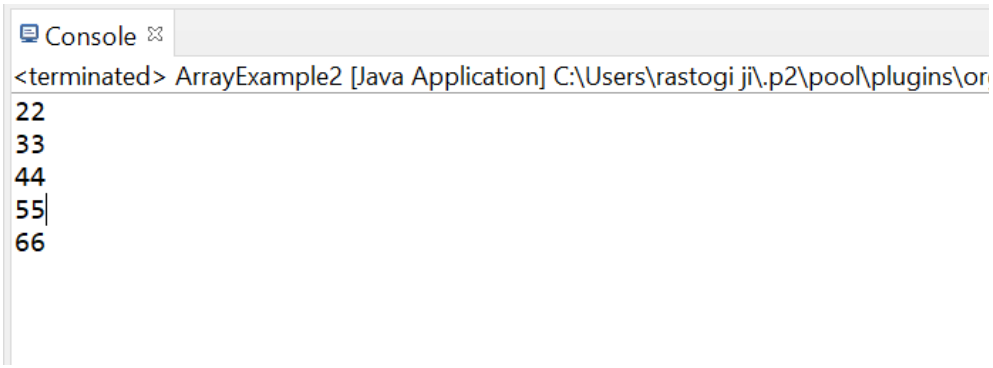
In this way, we initialize the array after the declaration of it. We use the new keyword assigning an array to a declared variable. Let's take an example and understand how we initialize an array after declaration.

ArrayExample2.java

```
public class ArrayExample2 {
```

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

        //declaration of an array
        int [] numbers;

        //initializing array after declaration
        numbers = new int[]{22,33,44,55,66};

        //print each element of the array
        for (int i = 0; i < 5; i++)
        {
            System.out.println(numbers[i]);
        }
    }

}
```

Output:



Java Initialize array

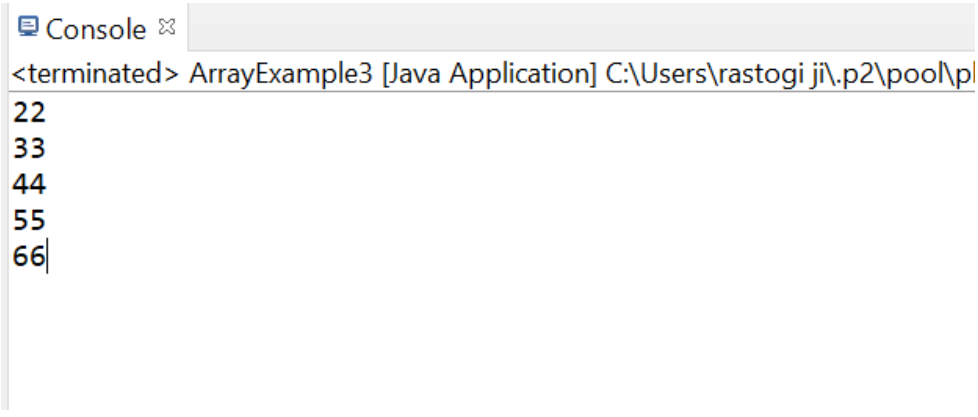
3. Initialize and assign values together

In this way, we declare and initialize the array together. We don't do both the declaration and initialization separately. Let's take an example and understand how we do both the thing together:

ArrayExample3.java

```
public class ArrayExample3 {
    //main() method start
    public static void main( String args[] ) {

        //declaration of an array
        int [] numbers = {22,33,44,55,66};

        //print each element of the array
        for (int i = 0; i < 5; i++)
        {
            System.out.println(numbers[i]);
        }
    }
}
```

Output:



2(b) Explain and scope and lifetime of a variable with example

Scope of Variables in Java

In programming, scope of variable defines how a specific variable is accessible within the program or across classes. In this section, we will discuss the scope of variables in Java.

Scope of a Variable

In programming, a variable can be declared and defined inside a class, method, or block. It defines the scope of the variable i.e. the visibility or accessibility of a variable. Variable declared inside a block or method are not visible to outside. If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.

We can declare variables anywhere in the program but it has limited scope.

A variable can be a parameter of a method or constructor.

A variable can be defined and declared inside the body of a method and constructor.

It can also be defined inside blocks and loops.

Variable declared inside main() function cannot be accessed outside the main() function

Variable Type	Scope	Lifetime
Instance variable	Troughout the class except in static methods	Until the object is available in the memory
Class variable	Troughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Demo.java

```
public class Demo
{
//instance variable
String name = "Andrew";
//class and static variable
static double height= 5.9;
```

```
public static void main(String args[])
{
//local variable
int marks = 72;
}
}
```

In Java, there are three types of variables based on their scope:

Member Variables (Class Level Scope)

Local Variables (Method Level Scope)

Member Variables (Class Level Scope)

These are the variables that are declared inside the class but outside any function have class-level scope. We can access these variables anywhere inside the class. Note that the access specifier of a member variable does not affect the scope within the class. Java allows us to access member variables outside the class with the following rules:

Access Modifier	Package	Subclass	Word
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	No	No	No
default	Yes	No	No

Syntax:

```
public class DemoClass
{
//variables declared inside the class have class level scope
int age;
private String name;
void displayName()
{
```

```
//statements
}
int dispalyAge()
{
//statements
}
char c;
}
```

Let's see an example.

VariableScopeExample1.java

```
public class VariableScopeExample1
{
public static void main(String args[])
{
int x=10;
{
//y has limited scope to this block only
int y=20;
System.out.println("Sum of x+y = " + (x+y));
}
//here y is unknown
y=100;
//x is still known
x=50;
}
}
```

```
/VariableScopeExample1.java:12: error: cannot find symbol
y=100;
^
  symbol:   variable y
  location: class VariableScopeExample1
1 error
```

2(c) Explain automatic type promotion in expressions with rules and a demo program.

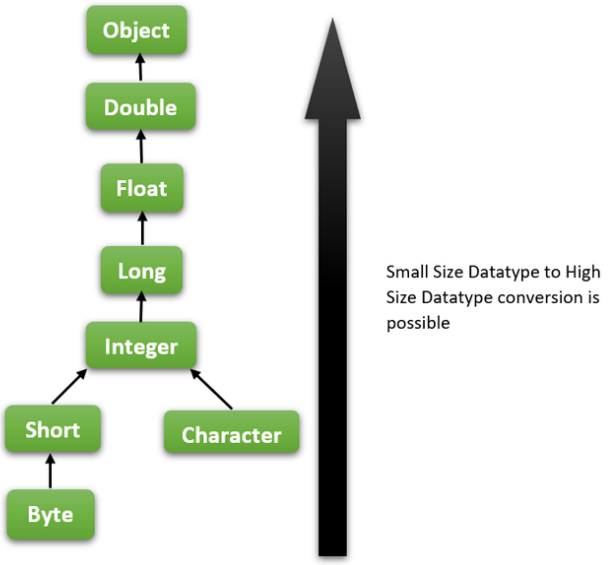
The name Type Promotion specifies that a small size datatype can be promoted to a large size datatype. i.e., an Integer data type can be promoted to long, float, double, etc. This Automatic Type Promotion is done when any method which accepts a higher size data type argument is called with the smaller data type.

Example:

```
public void method(double a){
   System.out.println("Method called");
}

public static void main(){
   method(2);
}
```

In the above method call, we passed an integer as an argument, but no method accepts an integer in the below code. The Java compiler won't throw an error because of the Automatic Type Promotion. The Integer is promoted to the available large size datatype, double.

Note:- This is important to remember is Automatic Type Promotion is only possible from small size datatype to higher size datatype but not from higher size to smaller size. i.e., integer to character is not possible.



Example 1: In this example, we are testing the automatic type promotion from small size datatype to high size datatype.

```
class GFG {

    // A method that accept double as parameter
    public static void method(double d)
    {
        System.out.println(
            "Automatic Type Promoted to Double-" + d);
    }

    public static void main(String[] args)
    {
        // method call with int as parameter
        method(2);
    }
}
```

Output

Automatic Type Promoted to Double-2.0

Explanation: Here we passed an Integer as a parameter to a method and there is a method in the same class that accepts double as parameter but not Integer. In this case, the Java compiler performs automatic type promotion from int to double and calls the method.

3(a) What are the different types of operators in Java? Explain them

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are:

Arithmetic Operators

Unary Operators

Assignment Operator

Relational Operators

Logical Operators

Ternary Operator

Bitwise Operators

Shift Operators

instance of operator

Let’s take a look at them in detail.

1. Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

* : Multiplication

/ : Division

% : Modulo

+ : Addition

– : Subtraction

2. Unary Operators: Unary operators need only one operand. They are used to increment, decrement or negate a value.

– : Unary minus, used for negating the values.

+ : Unary plus indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.

++ : Increment operator, used for incrementing the value by 1. There are two varieties of increment operators.

Post-Increment: Value is first used for computing the result and then incremented.

Pre-Increment: Value is incremented first, and then the result is computed.

— : Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operators.

Post-decrement: Value is first used for computing the result and then decremented.

Pre-Decrement: Value is decremented first, and then the result is computed.

! : Logical not operator, used for inverting a boolean value.

3. Assignment Operator: '=' Assignment operator is used to assigning a value to any variable. It has a right to left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

The general format of the assignment operator is:

variable = value;

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a Compound Statement. For example, instead of a = a+5, we can write a += 5.

+=, for adding left operand with right operand and then assigning it to the variable on the left.

-=, for subtracting right operand from left operand and then assigning it to the variable on the left.

*=, for multiplying left operand with right operand and then assigning it to the variable on the left.

/=, for dividing left operand by right operand and then assigning it to the variable on the left.

%=, for assigning modulo of left operand by right operand and then assigning it to the variable on the left.

4. Relational Operators: These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,

variable relation_operator value

Some of the relational operators are-

==, Equal to returns true if the left-hand side is equal to the right-hand side.

!=, Not Equal to returns true if the left-hand side is not equal to the right-hand side.

<, less than: returns true if the left-hand side is less than the right-hand side.

<=, less than or equal to returns true if the left-hand side is less than or equal to the right-hand side.

>, Greater than: returns true if the left-hand side is greater than the right-hand side.

>=, Greater than or equal to returns true if the left-hand side is greater than or equal to the right-hand side.

5. Logical Operators: These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

Conditional operators are:

&&, Logical AND: returns true when both conditions are true.

||, Logical OR: returns true if at least one condition is true.

!, Logical NOT: returns true when a condition is false and vice-versa

6. Ternary operator: Ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name ternary.

The general format is:

condition ? if true : if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

```
// Java program to illustrate
// max of three numbers using
// ternary operator.
public class operators {
    public static void main(String[] args)
    {
        int a = 20, b = 10, c = 30, result;

        // result holds max of three
        // numbers
        result
            = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
        System.out.println("Max of three numbers = "
                           + result);
    }
}
```

Output

Max of three numbers = 30

7. Bitwise Operators: These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

&, Bitwise AND operator: returns bit by bit AND of input values.

|, Bitwise OR operator: returns bit by bit OR of input values.

^, Bitwise XOR operator: returns bit-by-bit XOR of input values.

~, Bitwise Complement Operator: This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

8. Shift Operators: These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. General format-

number shift_op number_of_places_to_shift;

<<, Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.

>>, Signed Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of the initial number. Similar effect as dividing the number with some power of two.

>>>, Unsigned Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

9. instanceof operator: The instance of the operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. General format-

object instance of class/subclass/interface

```
// Java program to illustrate
// instance of operator
class operators {
    public static void main(String[] args)
    {

        Person obj1 = new Person();
        Person obj2 = new Boy();
```

```
        // As obj is of type person, it is not an
        // instance of Boy or interface
        System.out.println("obj1 instanceof Person: "
                           + (obj1 instanceof Person));
        System.out.println("obj1 instanceof Boy: "
                           + (obj1 instanceof Boy));
        System.out.println("obj1 instanceof MyInterface: "
                           + (obj1 instanceof MyInterface));

        // Since obj2 is of type boy,
        // whose parent class is person
        // and it implements the interface Myinterface
        // it is instance of all of these classes
        System.out.println("obj2 instanceof Person: "
                           + (obj2 instanceof Person));
        System.out.println("obj2 instanceof Boy: "
                           + (obj2 instanceof Boy));
        System.out.println("obj2 instanceof MyInterface: "
                           + (obj2 instanceof MyInterface));
    }
}

class Person {
}

class Boy extends Person implements MyInterface {
}

interface MyInterface {
```

}

Output

obj1 instanceof Person: true

obj1 instanceof Boy: false

obj1 instanceof MyInterface: false

obj2 instanceof Person: true

obj2 instanceof Boy: true

obj2 instanceof MyInterface: true

3(b) Discuss for each loop with an example

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Advantages

It makes the code more readable.

It eliminates the possibility of programming errors.

Syntax

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

for(data_type variable : array | collection){

```
//body of for-each loop
}
```

How it works?

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.

For-each loop Example: Traversing the array elements

```
//An example of Java for-each loop
class ForEachExample1{
  public static void main(String args[]){
   //declaring an array
   int arr[]={12,13,14,44};
   //traversing the array with for-each loop
   for(int i:arr){
     System.out.println(i);
   }
 }
}
```

Output:

```
12
12
14
44
```

Let us see another of Java for-each loop where we are going to total the elements.

```
class ForEachExample1{
  public static void main(String args[]){
   int arr[]={12,13,14,44};
```

```
   int total=0;
   for(int i:arr){
     total=total+i;
   }
  System.out.println("Total: "+total);
 }
}
```

Output:

```
Total: 83
```

For-each loop Example: Traversing the collection elements

```
import java.util.*;
class ForEachExample2{
  public static void main(String args[]){
   //Creating a list of elements
   ArrayList<String> list=new ArrayList<String>();
   list.add("vimal");
   list.add("sonoo");
   list.add("ratan");
   //traversing the list of elements using for-each loop
   for(String s:list){
     System.out.println(s);
   }

 }
}
```

Output:

```
vimal
sonoo
```

ratan

3(c) Differentiate between while loop and do-while loop in java

Here is the difference table:

while	do-while
Condition is checked first then statement(s) is executed.	Statement(s) is executed atleast once, thereafter condition is checked.
It might occur statement(s) is executed zero times, If condition is false.	At least once the statement(s) is executed.
No semicolon at the end of while. while(condition)	Semicolon at the end of while. while(condition);
If there is a single statement, brackets are not required.	Brackets are always required.
Variable in condition is initialized before the execution of loop.	variable may be initialized before or within the loop.
while loop is entry controlled loop.	do-while loop is exit controlled loop.
while(condition) { statement(s); }	do { statement(s); } while(condition);

4(a) Write a Java program to perform simple calculator operation.

```
import java.util.Scanner;
public class Calculator {
  public static void main(String[] args) {
```

```
    double num1;
    double num2;
    double ans;
    char op;
    Scanner reader = new Scanner(System.in);
    System.out.print("Enter two numbers: ");
    num1 = reader.nextDouble();
    num2 = reader.nextDouble();
    System.out.print("\nEnter an operator (+, -, *, /): ");
    op = reader.next().charAt(0);
    switch(op) {
      case '+': ans = num1 + num2;
        break;
      case '-': ans = num1 - num2;
        break;
      case '*': ans = num1 * num2;
        break;
      case '/': ans = num1 / num2;
        break;
    default: System.out.printf("Error! Enter correct operator");
      return;
    }
    System.out.print("\nThe result is given as follows:\n");
    System.out.printf(num1 + " " + op + " " + num2 + " = " + ans);
  }
}
```

4(b)Explain for-each loop with an example.

In Java, the **for-each** loop is used to iterate through elements of arrays and collections (like ArrayList). It is also known as the enhanced for loop.

- ## for-each Loop Sytnax

- The syntax of the Java **for-each** loop is:

```
for(dataType item : array) {
    ...
}
```

array - an array or a collection

item - each item of array/collection is assigned to this variable

dataType - the data type of the array/collection

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

    // create an array
    int[] numbers = {3, 9, 5, -5};

    // for each loop
    for (int number: numbers) {
        System.out.println(number);
    }
  }
}
```

4©Explain Java break and continue with while loop

Java Break

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch statement.

The break statement can also be used to jump out of a **loop**.

Java Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
    int i = 0;
    while (i < 10) {
    System.out.println(i);
    i++;
 if (i == 4) {
   break;
 }
}
int i = 0;
while (i < 10) {
 if (i == 4) {
         i++;
   continue;
 }
 System.out.println(i);
```

```
 i++;}
```

Module-3

5(a) Class defines a new data type. Once defined, this new type can be used to create

objects of that type.

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

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

and instance used interchangeably.

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(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.

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.

A Simple Class

Here is a class called Box that defines three instance variables: width, height, and depth.

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

As stated, a class defines a new type of data.

In this case, the new data type is called Box.

You will use this name to declare objects of type Box.

It is important to remember that a class declaration only creates a template; it does not create an actual object

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

5(b) What are constructors explain with an example.

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created.

Even when you add convenience functions like setDim( ), it would be simpler and more concise to have all of the setup done at the time the object is first created.

Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.

This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation.

It has the same name as the class in which it resides and is syntactically similar to a method.

Once defined, the constructor is automatically called 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.

It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
```

```
System.out.println(&quot;Constructing Box&quot;);
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(&quot;Volume is &quot; + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println(&quot;Volume is &quot; + vol);
}
}
```

5© Explain the following a)this keyword b) Garage collection c)Finalize method

The this keyword

Sometimes a method will need to refer to the object that invoked it.

To allow this, Java defines the this keyword. this can be used inside any method to

refer to the current object

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Uses of this:

To overcome shadowing or instance variable hiding.

To call an overload constructor

Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be

wondering how such objects are destroyed and their memory released for later

reallocation.

Java takes a different approach; it handles deallocation for you automatically.

The technique that accomplishes this is called garbage collection.

It works like this: when no references to an object exist, that object is assumed to be

no longer needed, and the memory occupied by the object can be reclaimed.

There is no explicit need to destroy objects as in C++.

Garbage collection only occurs sporadically (if at all) 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.

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.

6 a) What is inheritance? Discuss different types of inheritance with an eample.

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.     //methods and fields
4. }
```

Types of Inheritance in Java

Explain over

b) Explain method overriding with an example.

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

6© Explain abstract class and method in java with an example.

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
```

```
 Bike obj = new Honda4();
 obj.run();
}
}
```

Q. 7 a)What is a Package ? How to create and import the package in Java. Explain with an Example. 10M

Packages

A package as the name suggests is a pack(group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two **types of packages in Java**: built-in packages and the packages we can create (also known as user defined package). In this guide we will learn what are packages, what are user-defined packages in java and how to use them.

In java we have several built-in packages, for example when we need user input, we import a package like this:

import java.util.Scanner

Here:

→ **java** is a top level package

→ **util** is a sub package

→ and **Scanner** is a class which is present in the sub package **util**.

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

import pack.A;


class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }

}
```

Output:Hello

To Compile:

javac -d .
A.java

javac B.java

To Run: java
B

Q. 7 b)What is an interface? Explain how to define and implement interface by taking suitable example. 10M

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

```
interface printable{

void print( );  //public abstract void print();

}

class A6 implements printable{

public void print(){System.out.println("Hello");}


public static void main(String args[]){

A6 obj = new A6();

obj.print();

    }

   }
```

Q. 8 a) What is an Exception? Explain the following 12M

i) try

ii)catch

iii) throw

iv)throws

v)finally

What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs, the program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

```
public class TryCatchExample2 {


    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }


}
```

Output:

```
java.lang.ArithmeticException: / by zero

rest of the code
```

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) throws ArithmeticException {
        if (age < 18) {
        throw new ArithmeticException("Access denied - You must be at least 18 years old.");
        }
        else {
        System.out.println("Access granted - You are old enough!");
        }
  }


  public static void main(String[] args) {
        checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Finally

```
public class Main {
  public static void main(String[] args) {
        try {
        int[] myNumbers = {1, 2, 3};
        System.out.println(myNumbers[10]);
        } catch (Exception e) {
        System.out.println("Something went wrong.");
        } finally {
        System.out.println("The 'try catch' is finished.");
        }
  }
}
```

Output:

Q. 8 b) How do you create your own exception class? Explain with a program 8M

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

```
// A Class that represents use-defined exception


class MyException extends Exception {
```

```
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main {
    // Driver Program
    public static void main(String args[])
    {
        try {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex) {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
```

}

9) a. What is an applet?Explain program in applet with an example.
b.Explain the following :
i)Enumeration
ii)Type Wrappers

Answer:

a.
Applet is another type of program in java. Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity. Applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas.
simple applet shown here:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface. As you might expect, the AWT is quite large and sophisticated, and a complete discussion of it consumes several chapters in Part II of this book. Fortunately, this simple applet makes very limited use of the AWT. (Applets can also use Swing to provide the graphical user interface, but this approach is described later in this book.) The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet.
The next line in the program declares the class SimpleApplet. This class must be declared as public, because it will be accessed by code that is outside the program.
Inside SimpleApplet, paint( ) is declared. This method is defined by the AWT and must be overridden by the applet. paint( ) is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the

applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. 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.
Inside paint( ) is a call to drawString( ), which is a member of the Graphics class.
This method outputs a string beginning at the specified X,Y location. It has the following general form:
void drawString(String message, int x, int y)
Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to drawString( ) in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.
Notice that the 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.

b.i)
Versions prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations. In its simplest form, an enumeration is a list of named constants. Although Java offered other features that provide somewhat similar functionality, such as final variables, many programmers still missed the conceptual purity of enumerations—especially because enumerations are supported by most other commonly used languages. Beginning with JDK 5, enumerations were added to the Java language, and they are now available to the Java programmer.
In their simplest form, Java enumerations appear similar to enumerations in other languages. However, this similarity is only skin deep. In languages such as C++, enumerations are simply lists of named integer constants. In Java, an enumeration defines a class type. By making enumerations into classes, the concept of the enumeration is greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Therefore, although enumerations were several years in the making, Java’s rich implementation made them well worth the wait.
An enumeration is created using the enum keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants. Each is implicitly declared as a public, static final member of Apple. Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case. Thus, in the language of Java, these constants are called self-typed, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an enum using new. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares ap as a variable of enumeration type Apple:
Apple ap;
Because ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns ap the value RedDel:
ap = Apple.RedDel;
All enumerations automatically contain two predefined methods: values( ) and valueOf( ). Their general forms are shown here:
public static enum-type[ ] values( )
public static enum-type valueOf(String str)
The values( ) method returns an array that contains a list of the enumeration constants. The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str. In both cases, enum-type is the type of the enumeration.

b.ii)
As you know, Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit Object.
Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object. The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.
These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.
Character
Character is a wrapper around a char. The constructor for Character is
Character(char ch)
Here, ch specifies the character that will be wrapped by the Character object being created.
To obtain the char value contained in a Character object, call charValue( ), shown here:
char charValue( )
It returns the encapsulated character.
Boolean
Boolean is a wrapper around boolean values. It defines these constructors:
Boolean(boolean boolValue)
Boolean(String boolString)
In the first version, boolValue must be either true or false. In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be

true. Otherwise, it will be false.
To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:
boolean booleanValue( )
It returns the boolean equivalent of the invoking object.
The Numeric Type Wrappers
By far, the most commonly used type wrappers are those that represent numeric values. These are Byte, Short, Integer, Long, Float, and Double. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of an object in each of the different number formats. These methods are shown here:
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
For example, doubleValue( ) returns the value of an object as a double, floatValue( ) returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.
All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for Integer:
Integer(int num)
Integer(String str)
If str does not contain a valid numeric value, then a NumberFormatException is thrown.
All of the type wrappers override toString( ). It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to println( ), for example, without having to convert it into its primitive type.

10) a. What is String in Java?Explain string class constructors with an example
B. Explain the Following
i)String comparison Method
ii)Modifying a String

Answer:
a.
As is the case in most other programming languages, in Java
a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String.
Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.

The String, StringBuffer, and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically. All are declared final, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the CharSequence interface.

One last point: To say that the strings within objects of type String are unchangeable means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

The String Constructors

The String class supports several constructors. To create an empty String, you call the default constructor. For example,

```
String s = new String();
```

will create an instance of String with no characters in it.

Frequently, you will want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes s with the string “abc”.

You can specify a subrange of a character array as an initializer using the following Constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes s with the characters cde.

You can construct a String object that contains the same character sequence as another String object using this constructor:

```
String(String strObj)
```

Here, strObj is a String object. Consider this example:

```
// Construct one String from another.
class MakeString {
```

```
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, s1 and s2 contain the same string.

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

```
String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)
```

Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
public static void main(String args[]) {
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

This program generates the following output:

```
ABCDEF
CDE
```

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However,
most of the time, you will want to use the default encoding provided by the platform.

b.i)

String Comparison

The String class includes several methods that compare strings or substrings within strings.

Each is examined here.

equals( ) and equalsIgnoreCase( )

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

boolean equals(Object str)

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

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

boolean equalsIgnoreCase(String str)

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

Here is an example that demonstrates equals( ) and equalsIgnoreCase( ):

```
// Demonstrate equals() and equalsIgnoreCase().
        class equalsDemo {
        public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + " equals " + s2 + " -> " +
        s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " +
        s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " +
        s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
        s1.equalsIgnoreCase(s4));
        }
        }
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

regionMatches( )

The regionMatches( ) method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

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

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

startsWith( ) and endsWith( )

String defines two routines that are, more or less, specialized forms of regionMatches( ). The startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified string. They have the following general forms:

boolean startsWith(String str)
boolean endsWith(String str)

Here, str is the String being tested. If the string matches, true is returned. Otherwise, false is returned. For example,

"Foobar".endsWith("bar")

and

"Foobar".startsWith("Foo")

are both true.

A second form of startsWith( ), shown here, lets you specify a starting point:

boolean startsWith(String str, int startIndex)

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,

"Foobar".startsWith("bar", 3)

returns true.

equals( ) Versus ==

It is important to understand that the equals( ) method and the == operator perform two different operations. As just explained, the equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

The variable s1 refers to the String instance created by “Hello”. The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

Hello equals Hello -> true
Hello == Hello -> false

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo( ) serves this purpose. It has this general form:

int compareTo(String str)

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted, as shown here:

Value Meaning

Less than zero The invoking string is less than str.

Greater than zero The invoking string is greater than str.

Zero The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses compareTo( ) to determine sort ordering for a bubble sort:

```
// A bubble sort for Strings.
class SortString {
static String arr[] = {
"Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country"
};
public static void main(String args[]) {
for(int j = 0; j < arr.length; j++) {
for(int i = j + 1; i < arr.length; i++) {
if(arr[i].compareTo(arr[j]) < 0) {
String t = arr[j];
arr[i] = t;
}
}
System.out.println(arr[j]);
}
}
}
```

The output of this program is the list of words:

Now
aid
all
come
country
for
good
is

men
of
the
the
their
time
to
to

As you can see from the output of this example, compareTo( ) takes into account uppercase and lowercase letters. The word “Now” came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use compareToIgnoreCase( ), as shown here:

int compareToIgnoreCase(String str)

This method returns the same results as compareTo( ), except that case differences are ignored.

You might want to try substituting it into the previous program. After doing so, “Now” will no longer be first.

b.ii)

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

substring( )

You can extract a substring using substring( ). It has two forms. The first is

String substring(int startIndex)

Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

The second form of substring( ) allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses substring( ) to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
```

```
String result = "";
int i;
do { // replace all matching substrings
System.out.println(org);
i = org.indexOf(search);
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result;
}
} while(i != -1);
}
}
```

The output from this program is shown here:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

You can concatenate two strings using concat( ), shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into s2. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The replace( ) method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into s.

The second form of replace( ) replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

This form was added by J2SE 5.

trim( )

The trim( ) method returns a copy of the invoking string from which any leading and trailing

whitespace has been removed. It has this general form:
String trim( )
Here is an example:
String s = " Hello World ".trim();
This puts the string “Hello World” into s.
The trim( ) method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state’s capital. It uses trim( ) to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;
class UseTrim {
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 'stop' to quit.");
System.out.println("Enter State: ");
do {
str = br.readLine();
str = str.trim(); // remove whitespace
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
else if(str.equals("Missouri"))
System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
System.out.println("Capital is Olympia.");
// ...
} while(!str.equals("stop"));
}
}
```