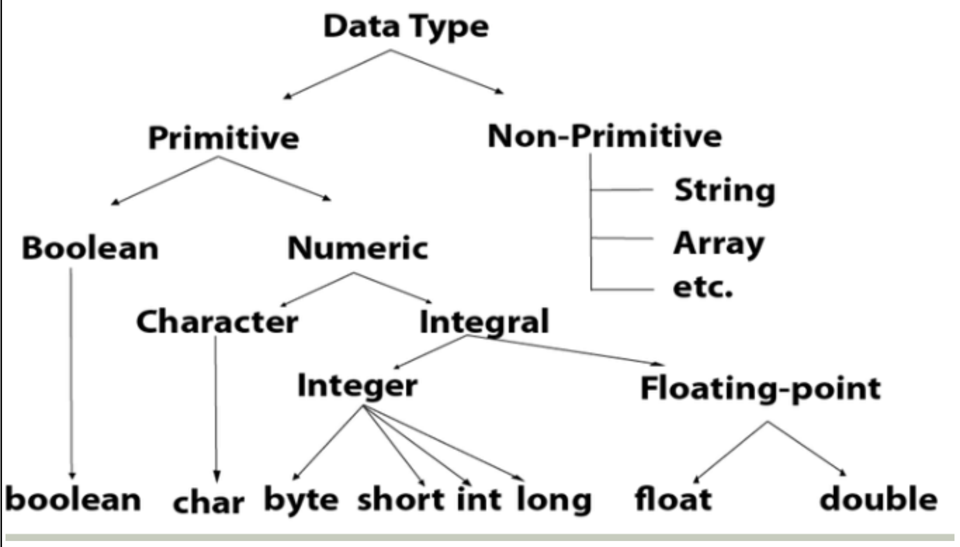
USN										

Internal Assessment Test 1 – Dec. 2021

Sub:	PROGRAMMING IN JAVA					Sub Code:	18CS653	Branch:	ECE
Date:	13-05-2022	Duration:	90 Minutes	Max Marks:	50	Sem / Sec:	6/A, B,C,D		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Explain breifly primtive type of data types in Java.	10	CO1	L2 L2



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
```

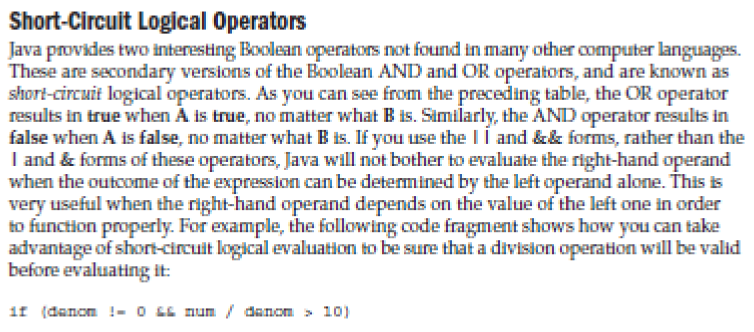
Char Data Type

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'
```

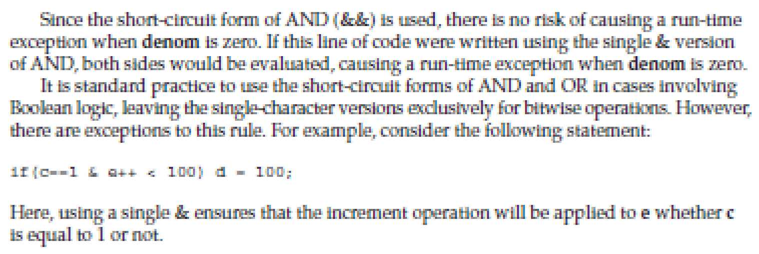
Explain use of Short circuit properties for logical operators (&& and ||).



Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators. As you can see from the preceding table, the OR operator results in **true** when **A** is **true**, no matter what **B** is. Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```



Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

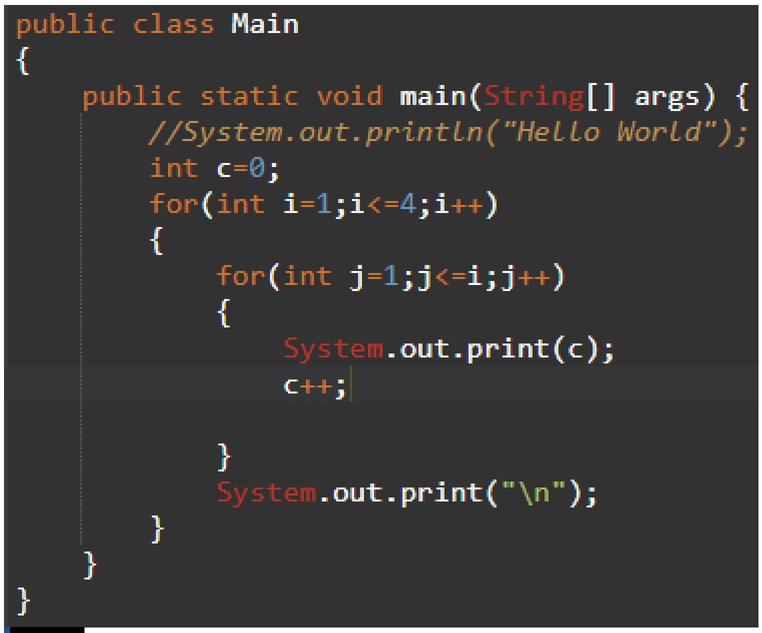
It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & e++ < 100) d = 100;
```

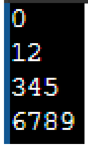
Here, using a single & ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

2	Write a Java program to print following pattern.	10	CO1	L3

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

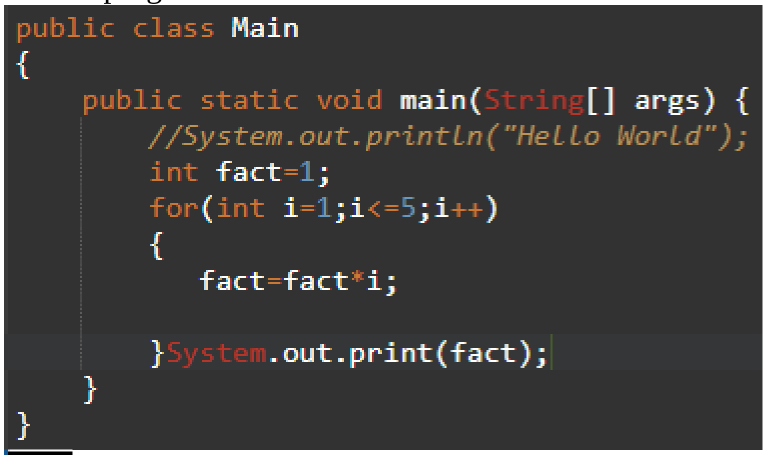


```
public class Main
{
    public static void main(String[] args) {
        //System.out.println("Hello World");
        int c=0;
        for(int i=1;i<=4;i++)
        {
            for(int j=1;j<=i;j++)
            {
                System.out.print(c);
                c++;

            }
            System.out.print("\n");
        }
    }
}
```

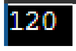


```
0
12
345
6789
```

	Write a program to find factorial of a number 5.			L3



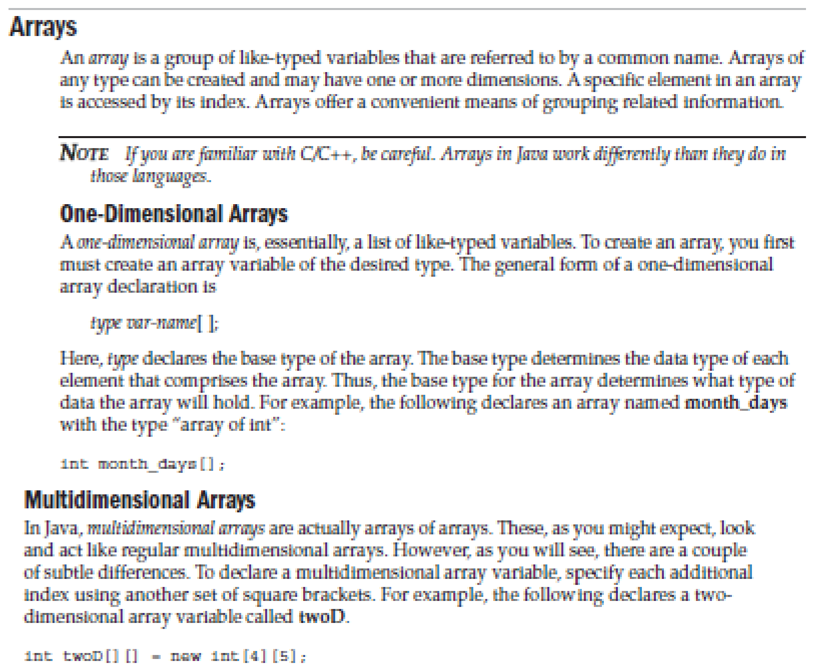
```
public class Main
{
    public static void main(String[] args) {
        //System.out.println("Hello World");
        int fact=1;
        for(int i=1;i<=5;i++)
        {
           fact=fact*i;

        }System.out.print(fact);
    }
}
```

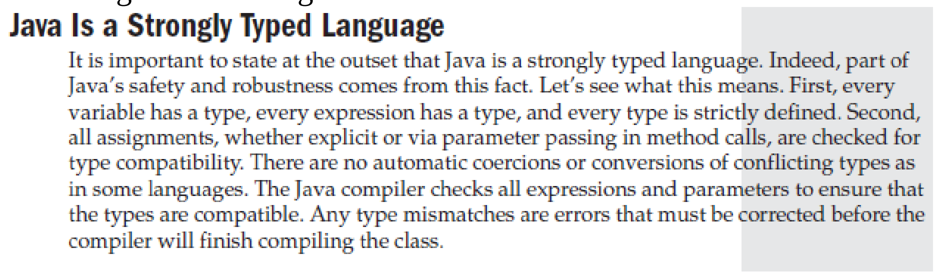


```
120
```

3	How Java arrays are initialized and declared ,explain with an example Write a Java program to display twisted prime numbers from 1 to 1000 using for loop. // Java program to check if a given number	10	CO1	L2 L3

```
// is Twisted Prime or not
import java.io.*;
import java.math.*;

class GFG
{
    static int reverse(int n)
    {
        int rev = 0, r;
        while (n > 0)
        {
            r = n % 10;
            rev = rev * 10 + r;
            n /= 10;
        }
        return rev;
    }
    static boolean isPrime(int n)
    {
        // Corner cases
        if (n <= 1)
            return false;
        if (n <= 3)
            return true;

        // This is checked so that we can skip
        // middle five numbers in below loop
        if (n % 2 == 0 || n % 3 == 0)
            return false;

        for (int i = 5; i * i <= n; i = i + 6)
            if (n % i == 0 || n % (i + 2) == 0)
                return false;

        return true;
    }

    // function to check Twisted Prime Number
    static boolean checkTwistedPrime(int n)
    {
        if (isPrime(n) == false)
            return false;

        return isPrime(reverse(n));
    }

    // Driver Code
    public static void main(String args[])
    throws IOException
    {
        // Printing Twisted Prime Numbers upto 200
        System.out.println("First few Twisted Prime" +
        " Numbers are :- n");
```

	for (int i = 2; i <= 1000; i++)
	if (checkTwistedPrime(i))
	System.out.print(i + " ");
	}
	}
	
	Arrays
	An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.
	NOTE *If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.*
	One-Dimensional Arrays
	A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is
	type var-name[ ];
	Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":
	`int month_days[];`
	Multidimensional Arrays
	In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.
	`int twoD[][] = new int[4][5];`			

4	Explain why java is a strongly typed language. Differentiate between narrowing and widening.  **Java Is a Strongly Typed Language** It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class. A **Type casting** in Java is used to convert objects or variables of one type into another. When we are converting or assigning one data type to another they might not compatible. If it is suitable then it will do smoothly otherwise chances of data loss. **Type Casting Types in Java** Java Type Casting is classified into two types. • Widening Casting (**Implicit**) – Automatic Type Conversion • Narrowing Casting (**Explicit**) – Need Explicit Conversion **Widening Casting (smaller to larger type)** **Widening Type Conversion** can happen if both types are compatible and the target type is larger than source type. Widening Casting takes place when **two types are compatible** and the **target type is larger than the source type** **Example1**	10	CO1	L2

```
public class ImplicitCastingExample {
   public static void main(String args[]) {
      byte i = 40;
      // No casting needed for below conversion
      short j = i;
      int k = j;
      long l = k;
      float m = l;
      double n = m;
      System.out.println("byte value : "+i);
      System.out.println("short value : "+j);
      System.out.println("int value : "+k);
      System.out.println("long value : "+l);
      System.out.println("float value : "+m);
      System.out.println("double value : "+n);
   }
}
```

Output

```
byte value : 40
short value : 40
int value : 40
long value : 40
float value : 40.0
double value : 40.0
```

Narrowing Casting (larger to smaller type)

When we are assigning a larger type to a smaller type, **Explicit Casting** is required.

Example1

```
public class ExplicitCastingExample {
   public static void main(String args[]) {
      double d = 30.0;
      // Explicit casting is needed for below
conversion
      float f = (float) d;
      long l = (long) f;
      int i = (int) l;
      short s = (short) i;
      byte b = (byte) s;
```

```
        System.out.println("double value : "+d);
        System.out.println("float value : "+f);
        System.out.println("long value : "+l);
        System.out.println("int value : "+i);
        System.out.println("short value : "+s);
        System.out.println("byte value : "+b);
    }
}
```

Output

```
double value : 30.0
float value : 30.0
long value : 30
int value : 30
short value : 30
byte value : 30
```

5	Explain three OOP principles. Explain steps to execute simple java program and role of JVM in execution.	10	CO1	L2

Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code.

The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction.

For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and

- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Java program execution follows 5 majors steps

- Edit - Here the programmer uses a simple editor or a notepad application to write the java program and in the end give it a ".java" extension.
- Compile - In this step, the programmer gives the javac command and the .java files are converted into bytecode which is the language understood by the Java virtual machine (and this is what makes Java platform independent language). Any compile time errors are raised at this step.
- Load - The program is then loaded into memory. This is done by the class loader which takes the .class files containing the bytecode and stores it in the memory. The .class file can be loaded from your hard disk or from the network as well.
- Verify - The bytecode verifier checks if the bytecode loaded are valid and do not breach java security restrictions.
- Execute - The JVM interprets the program one bytecode at a time and runs the program.

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

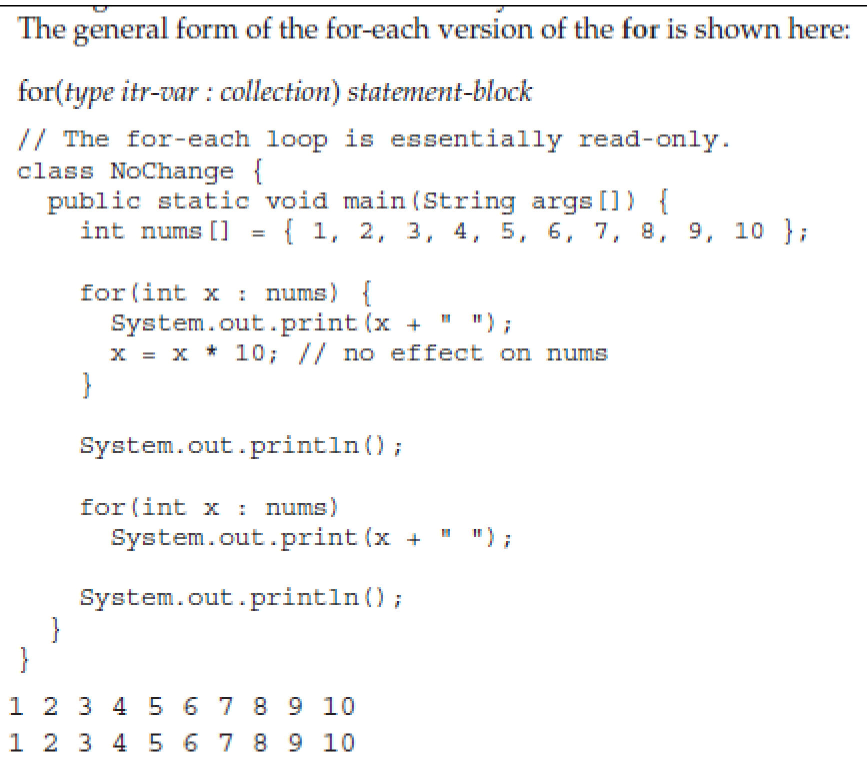
The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

6	Explain use of for..each loop with suitable example. Write program to find sum of first five numbers using for..each loop.	10	CO1	L2 L3



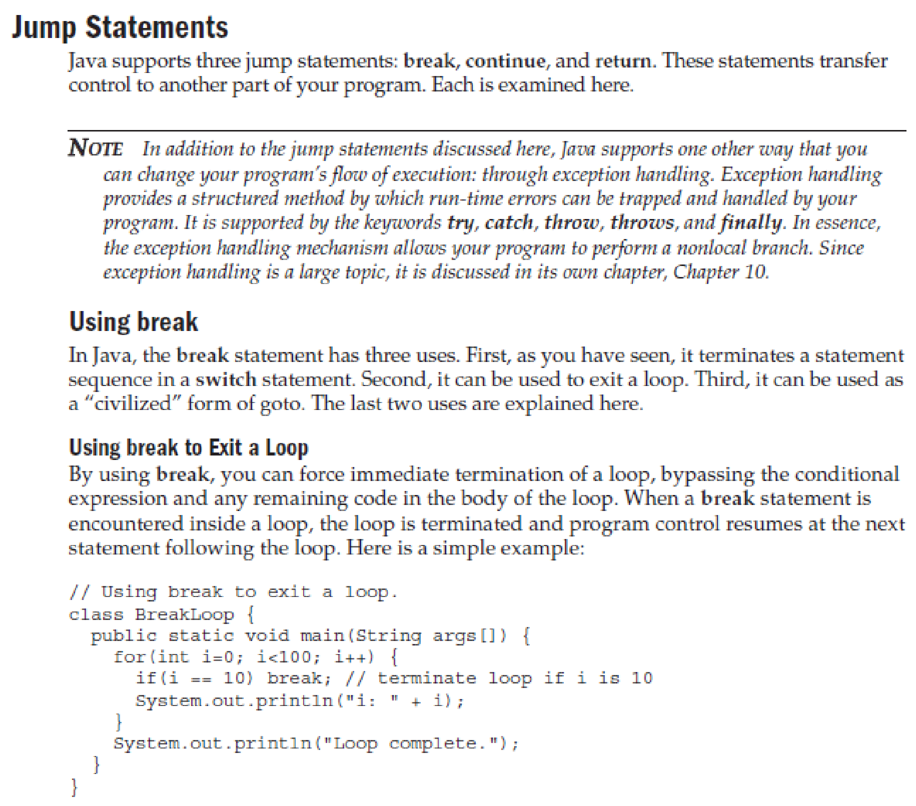
The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*) *statement-block*

```
// The for-each loop is essentially read-only.
class NoChange {
  public static void main(String args[]) {
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for(int x : nums) {
      System.out.print(x + " ");
      x = x * 10; // no effect on nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

    System.out.println();
  }
}
```

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

7	Explain break, continue and return statement with Syntax and example.	10	CO1	L2



Jump Statements

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

Note *In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords **try**, **catch**, **throw**, **throws**, and **finally**. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.*

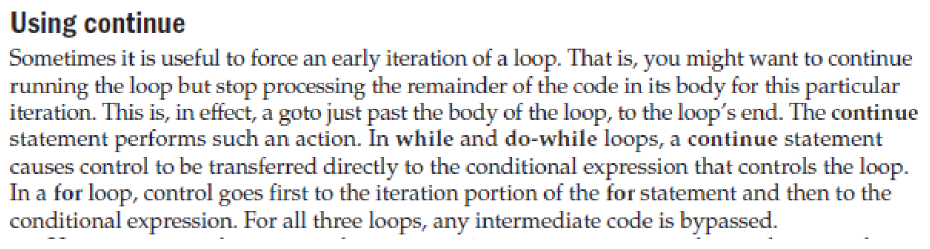
Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

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



Using continue

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

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

This code uses the % operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

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

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 6, a brief look at **return** is presented here.

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