1. Introduction & Features of Oops

Questions

1. What is Java? Explain its Features.

Java

- Java is a programming language and a computing platform for application development.
- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995
- Java is guaranteed to be Write Once, Run Anywhere.
- It is one of the most used programming languages.
- It is used for: Mobile applications, Desktop applications, Web applications, Web servers and application servers, Games, Database connection, and much, much more!

First Java Program

```
public class MyFirstJavaProgram {
   public static void main(String []args) {
      System.out.println("Hello World");
   }
}
```

Follows these steps to save the file, compile, and run the program -

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

Features of Java

1. **Object Oriented** – Java is an Object Oriented Programming Language, In Java, everything is an Object.
   Four main concepts of Object Oriented programming are:
   - Abstraction
   - Encapsulation
   - Inheritance
   - Polymorphism

2. **Platform Independent** − Java is a Platform Independent Programming language because it complied on Virtual Machine (JVM) on whichever platform it is being run on.

3. **Simple** − Java is designed to be easy to learn. Java Programming Language is easily understandable

4. **Secure** – Java is a Secure Programming Language, it enables to develop virus-free system. and also uses different authentication techniques

5. **Architecture-neutral** − Java is an architectural-neutral programming language; Java compiler generates an architecture-neutral object file format.

6. **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. C

7. **Robust** − Java is a Robust Programming Language; it allows compile time error checking and runtime checking.

8. **Multithreaded** – Java is a Multithreaded Programming Language. In Java it is possible to write programs that can perform many tasks simultaneously.

9. **Interpreted** –Java is both Compiled and Interpreted Programming Language. A class file in Java is interpreted by the JVM.

10. **High Performance** − with the use of Just-In-Time compilers, Java enables high performance.

11. **Distributed** − Java is designed for the distributed environment of the internet.

12. **Dynamic** − Java is considered to be more dynamic than C or C++ and other programming language.

2. An Overview of Java

Questions

1. Explain Object Oriented Programming, Abstraction and three oops principles.
2. Explain Security and Portability in regards of Oops

Object Oriented Programming

- Object-Oriented Programming is a paradigm that provides many concepts such **as** inheritance**,** data binding**,** polymorphism**, etc.**
- Many of the most widely used programming languages such as C++, Java, Python etc. support object-oriented programming.
- The main aim of object-oriented programming is to implement real-world entities for example object, classes, abstraction, inheritance, polymorphism, etc.

Features of OOP

There are following features of OOP Language.

1. Abstraction
2. Encapsulation.
3. Inheritance
4. Polymorphism

Abstraction

- Abstraction is selecting data from a larger pool to show only the relevant details to the object.
- It helps to reduce programming complexity and effort.
- In Java, abstraction is accomplished using Abstract classes and interfaces.
- It is one of the most important concepts of OOPs.
- Suppose you want to create a banking application and you are asked to collect all the information about your customer. There are chances that you will come up with a large amount of information about the customer But, not all the information is used to create a banking application. So, you need to select only the useful information for your banking application from that pool. Data like name, address, tax information, etc. make sense for a banking application. Since we have fetched/removed/selected the customer information from a larger pool, the process is referred as Abstraction.

The Three Oops Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.

They are:

1) Encapsulation
2) Inheritance
3) Polymorphism.

1) Encapsulation

- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

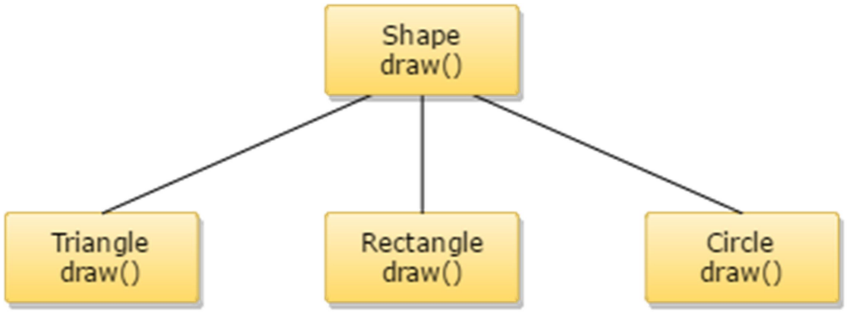
2) Inheritance

- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **inheritance**.

- The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.
- In Java, when an "Is-A" relationship exists between two classes we use Inheritance
- The parent class is termed super class and the inherited class is the sub class
- The keyword "extend" is used by the sub class to inherit the features of super class
- Inheritance is important since it leads to reusability of code

3) Polymorphism.

- Polymorphism is a OOPs concept where one name can have many forms.



Polymorphism

- For example, you have a smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called Polymorphism.
- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Security

- As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.
- For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system.
- In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
- Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.
- The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

Portability

- Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
- For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.
- It is not practical to have different versions of the applet for different computers.
- The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed.

3. Data Types

Questions

1. What is Data Types? Explain the different data types in Java.
2. What do you mean by operators? Explain the different operators in java.
3. What do you mean by Java's selection statements? Explain its types with example.
4. What do you mean by Iteration Statement/Loops in Java? Explain its types with example.
5. What do you mean by Jump Statement in Java? Explain its types with example.

Data Types

Data types are the means for the tasks related to identifying and assessing the type of data. Java is rich in data types which allows the programmer to select the appropriate type needed to build variables of an application.

- Every variable in Java has a data type which tells the compiler what type of variable it as and what type of data it is going to store.
- Data type specifies the size and type of values.
- Information is stored in computer memory with different data types.
- Whenever a variable is declared, it becomes necessary to define a data type that what will be the type of data that variable can hold.
- Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:

1. Primitive Data Types
2. Non Primitive Data Types:

1. Primitive Data Types

Java defines eight *primitive* types of data: byte, short, int, long, char, float, double, and Boolean. The primitive types are also commonly referred to as *simple* types, and both terms will be used in this book.

These can be put in four groups:

a) Integers:
b) Floating-Point Numbers:
c) Characters:
d) Boolean:

a) Integers

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

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 and Maximum value is 127
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 and Maximum value is 32,767
- Short data type can also be used to save memory as byte data type.
- A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

b) **Floating-Point Type**

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

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

c) **Characters**:

This group includes char, which represents symbols in a character set, like letters and numbers.

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

d) **Boolean**:

This group includes Boolean, which is a special type for representing true/false values.

- boolean data type represents one bit of information
- There are only two possible values: true and false
- Default value is false
- Example: boolean one = true

2. **Primitive Data Types**

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

Java Operators

Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, + is an operator that performs addition.
Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

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

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. Assume integer variable A holds 10 and variable B holds 20, then-

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

Example

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 25;
    int d = 25;

    System.out.println("a + b = " + (a + b) );
    System.out.println("a - b = " + (a - b) );
    System.out.println("a * b = " + (a * b) );
    System.out.println("b / a = " + (b / a) );
    System.out.println("b % a = " + (b % a) );
    System.out.println("c % a = " + (c % a) );
    System.out.println("a++   = " +  (a++) );
    System.out.println("b--   = " +  (a--) );

    // Check the difference in d++ and ++d
    System.out.println("d++   = " +  (d++) );
    System.out.println("++d   = " +  (++d) );
  }
}
```

Output

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and performs bit-by-bit operation.

Assume if a = 60 and b = 13; now in binary format they will be as follows −
a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

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

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Example

The following program is a simple example that demonstrates the bitwise operators.

```
public class Test {

  public static void main(String args[]) {
    int a = 60;       /* 60 = 0011 1100 */
    int b = 13;       /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;        /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );

    c = a | b;        /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );

    c = a ^ b;        /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );

    c = ~a;           /*-61 = 1100 0011 */
    System.out.println("~a = " + c );

    c = a << 2;       /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c );

    c = a >> 2;       /* 15 = 1111 */
```

```
    System.out.println("a >> 2  = " + c );

    c = a >>> 2;      /* 15 = 0000 1111 */
    System.out.println("a >>> 2 = " + c );
  }
}
```

Output

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2  = 15
a >>> 2 = 15
```

Relational Operators

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

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

Example

The following program is a simple example that demonstrates the relational operators.

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;

    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
```

```
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
  }
}
```

Output

This will produce the following result −

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

Boolean Logical Operators

The following table lists the logical operators –
Assume Boolean variables A holds true and variable B holds false, then –

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

Example

The following simple example program demonstrates the logical operators.

```
public class Test {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;

    System.out.println("a && b = " + (a&&b));
    System.out.println("a || b = " + (a||b) );
    System.out.println("!(a && b) = " + !(a && b));
  }
}
```

Output

This will produce the following result −

```
a && b = false
```

```
a || b = true
!(a && b) = true
```

Assignment Operators

Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assigns the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assigns the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assigns the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assigns the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example

The following program is a simple example that demonstrates the assignment operators.

```
public class Test {

  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;

    c = a + b;
    System.out.println("c = a + b = " + c );
    c += a ;
    System.out.println("c += a  = " + c );
    c -= a ;
    System.out.println("c -= a = " + c );
    c *= a ;
    System.out.println("c *= a = " + c );
    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );
```

```
    a = 10;
    c = 15;
    c %= a ;
    System.out.println("c %= a  = " + c );
    c <<= 2 ;
    System.out.println("c <<= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c &= a ;
    System.out.println("c &= a  = " + c );
    c ^= a ;
    System.out.println("c ^= a   = " + c );
    c |= a ;
    System.out.println("c |= a   = " + c );
  }
}
```

Output

This will produce the following result −

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

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

The operator is written as –

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

Example

```
public class Test {

public static void main(String args[]) {
int a, b;
a = 10;
b = (a == 1) ? 20: 30;
```

```
System.out.println( "Value of b is : " +  b );

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

Output

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

JAVA's selection statements

In Java, these are used to control the flow of the program. These are the types of selection statements in Java.

a) If statement
b) If-else statement
c) Switch statement

a) If Statement

In Java, "if" is a conditional statement. It will provides execution of one of two statements (or blocks), depending on the condition.

Syntax

```
if(condition)
{
   statements;
   ...
   ...
}
```

If the condition is true then the statements inside the if block will be executed. The execution will be continued to the next statements after the execution of the statements in the if block. If the condition is false then the statements inside the if block will not be executed and the execution will start from the statements that are next to the if block.

Example

```
package test;
import java.util.Scanner;
public class Test
{
   public static void main(String args[])
   {
      int b,c;
      Scanner scnr = new Scanner(System.in);
      System.out.println("b is : ");
      b=scnr.nextInt();
      System.out.println("c is : ");
```

```
        c=scnr.nextInt();
        if (b > c)
        {
           System.out.println("b is greater than c");
        }
        System.out.println("example for the comparison of two numbers");
    }
}
```

Output

```
b is :
6
c is :
4
b is greater than c
example for the comparison of two numbers
```

b) **If-else Statement**

If the condition is true then the statements inside the if block will be executed and if the condition is false then the statements inside the else block will be executed.

Syntax

```
if (condition)
{
   first statement;
}
else
{
   second statement;
}
```

If the condition is true then the first statement will be executed and if the condition is false then the second statement will be executed.

Example

```
package demo;
import java.util.Scanner;
public class Demo
{
   public static void main(String args[])
   {
      int a,b;
      Scanner scnr = new Scanner(System.in);
      System.out.println("a is : ");
      a=scnr.nextInt();
      System.out.println("b is : ");
      b=scnr.nextInt();
      if (a > b)
      {
         System.out.println("a is largest");
      }
      else
      {
```

```
            System.out.println("b is largest");
        }
    }
}
```

Output

```
a is :
5
b is :
6
b is largest
```

c) **Switch Statement**

A switch statement can be easier than if-else statements. In the switch we have multiple cases. The matching case will be executed. In the switch statement we can only use int, char, byte and short data types.

Syntax

```
switch (expression)
{
   case 1:
   {
      statement;
   }
   break;
   case 2:
   {
      statement;
   }
   break;
   .
   .
   .
   case N:
   {
      statement;
   }
   break;
   default:
   {
      statement;
   }
   break;
}
```

Example

```
package demo;
import java.util.Scanner;
public class Demo
{
   public static void main(String[] args)
   {
      int x,y,r;
```

```
        double z;
        Scanner scnr = new Scanner(System.in);
        System.out.print("Enter x : ");
        x=scnr.nextInt();
        System.out.print("Enter y : ");
        y=scnr.nextInt();
        System.out.println("1 : Addition");
        System.out.println("2 : Subtraction");
        System.out.println("3 : Multiplication");
        System.out.println("4 : Division");
        System.out.print("Requirement : ");
        r=scnr.nextInt();
        switch(r)
        {
           case 1:
           {
              z=x+y;
              System.out.println(z);
           }
           break;
           case 2:
           {
              z=x-y;
              System.out.println(z);
           }
           break;
           case 3:
           {
              z=x*y;
              System.out.println(z);
           }
           break;
           case 4:
           {
              z=x/y;
              System.out.println(z);
           }
           break;
           default:
            {
               System.out.println("Requirement is invalid");
            }
         }
      }
   }
```

Output

```
Enter x : 10
Enter y : 20
1 : Addition
2 : Subtraction
3 : Multiplication
4 : Division
Requirement : 3
200.0
```

Iteration Statement

- Java's iteration statements are **for**, **while**, and **do-while**.
- These statements create what we commonly call *loops.* As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.
- Iteration statements or loop statements allow us to execute block of statements as long as condition is true.

Type of Iteration Statements
In Java there are 3 types of iteration statements.

1. while loop
2. do-while loop
3. for loop

1. while loop

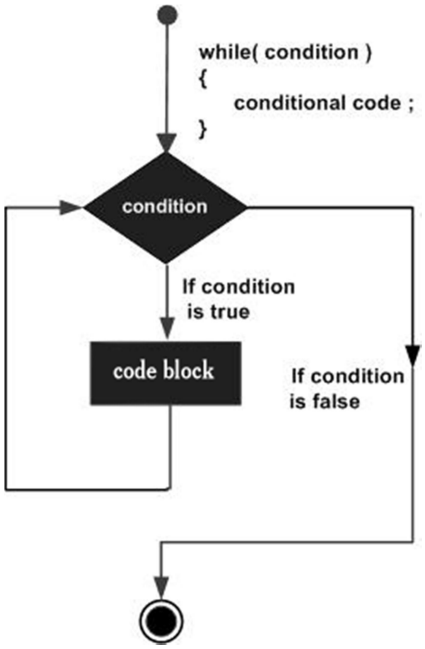
A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax
The syntax of a while loop is –

```
while(Boolean_expression) {
  // Statements
}
```

- Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value.
- When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.
- When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    while( x < 20 ) {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }
  }
}
```

This will produce the following result −

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

do…while loop

A **do…while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

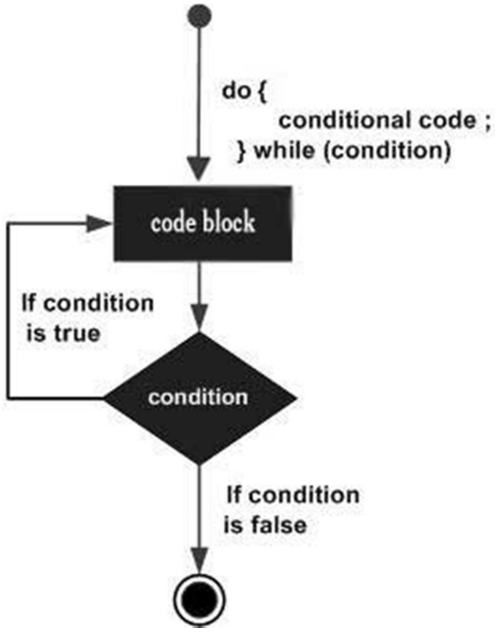
Following is the syntax of a do...while loop −

```
do {
  // Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Flow Diagram



Example

```
public class Test {

  public static void main(String args[]) {
    int x = 10;

    do {
      System.out.print("value of x : " + x );
      x++;
      System.out.print("\n");
    }while( x < 20 );
  }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

for loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.
- A **for** loop is useful when you know how many times a task is to be repeated.

Syntax

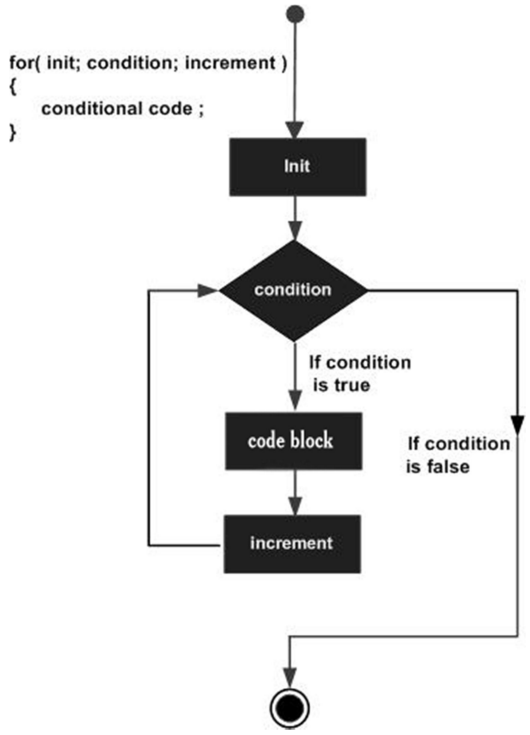
The syntax of a for loop is −

```
for(initialization; Boolean_expression; update) {
  // Statements
}
```

Here is the flow of control in a **for** loop −

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Flow Diagram



Example

Following is an example code of the for loop in Java.

```
public class Test {

  public static void main(String args[]) {

    for(int x = 10; x < 20; x = x + 1) {
      System.out.print("value of x : " + x );
      System.out.print("\n");
    }
  }
}
```

Output

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Jump Statement

- In Java jump statements are mainly used to transfer control to another part of our program depending on the conditions.

- These statements are very useful from the programmer's view because these statements allow alteration of the flow of execution of the program.
- These statements can be used to jump directly to other statements, skip a specific statement and so on.
- In Java we have the following three jump statements:
  1. break
  2. continue
  3. return

1. Break

The break construct is used to break out of the middle of loops: for, do, or while loop. When a break statement is encountered, execution of the current loops immediately stops and resumes at the first statement following the current loop. That is, we can force immediate termination of a loop, bypassing any remaining code in the body of the loop.

Example

```
/* Print 1 to 10 using Break Statement Java Example*/
        class BreakStatement
        {
            public static void main(String args[] )
                {
                    int i;
                    i=1;
                    while(true)
                    {
                        if(i >10) break;
                        System.out.print(i+" ");
                        i++;
                    }
                }
            }
```

Continue

This command skips the whole body of the loop and executes the loop with the next iteration. On finding continue command, control leaves the rest of the statements in the loop and goes back to the top of the loop to execute it with the next iteration (value).

Example

```
/* Print Number from 1 to 10 Except 5 */
        class NumberExcept
        {
            public static void main(String args[] )
            {
                int i;
                for(i=1;i<=10;i++)
                {
                    if(i==5) continue;
                    System.out.print(i +" ");
```

```
                    }
                }
            }
```

Above program will display the value of variable i from 1 to 4. When the value of variable i becomes 5, continue statement will skip the body of the loop following continue statement i.e. it skips System.out.println(i) statement and again executes the loop with the next iteration (value) i.e .. 6.

The return statement

- The return statement is the last jump statement.
- The return statement is used to end the execution of a specific method and then return a value.
- When we use a return statement in our program then it sends the program control to the method caller.
- The data type of the returned value should always be equal to the data type of the method's declared return value.

Syntax:

```
if(condition)
{
  return;
}
```

Example:

```
package myclass1;
class Myclass1
{
  public static void main(String[] args)
  {
    yahoo(true);
    System.out.println("hi");
  }
  public static void yahoo(boolean a)
  {
    System.out.println("1");
    if (a)
    {
      return;
    }
    System.out.println("2");
    System.out.println("3");
  }
}
```

Output:

```
run:
1
hi
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Classes & Methods

Syllabus

Class fundamentals, declaring objects, overloading methods & constructs, access control, Nested and inner classes, exploring the string class.

Questions

1. Define Class and Object in Java with example.
2. Explain Method Overloading in java.
3. Explain different access control methods in java.
4. Explain inner and nested classes in java.
5. Explain the string class in java.

Class

- A class is an entity that determines how an object will behave and what the object will contain.
- In other words, it is a blueprint or a set of instruction to build a specific type of object.
- A class is a user defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.

Syntax of Class

```
class <class_name>{
   field;
   method;
 }
```

Object

- An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. Object determines the behavior of the class.
- When you send a message to an object, you are asking the object to invoke or execute one of its methods.
- From a programming point of view, an object can be a data structure, a variable or a function.
- It has a memory location allocated. The object is designed as class hierarchies.

Syntax of Object

```
ClassName ReferenceVariable = new ClassName();
```

Declaring an Object

In Java, the new keyword is used to create/declare new objects.

Example of Class and Object

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

Output

If we compile and run the above program, then it will produce the following result −

```
Passed Name is : tommy
```

Method Overloading

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**. If we have to perform only one operation, having same name of the methods increases the readability of the program. Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int, int, int) for three parameters then it may be difficult for you to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a, int b){return a+b;}
static int add(int a, int b, int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

- In this example, we have created two methods that differs in data type.
- The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

Introducing Access Control

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class. There are two types of modifiers in java: access modifiers and non-access modifiers.

There are 4 types of java access modifiers:

1. **Private:** The private access modifier is accessible only within class.

   Example

   The following class uses private access control −

```
public class Logger {
  private String format;

  public String getFormat() {
    return this.format;
  }

  public void setFormat(String format) {
    this.format = format;
  }
}
```

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

   Example

   Variables and methods can be declared without any modifiers, as in the following examples −

```
String version = "1.5.1";

boolean processOrder() {
  return true;
}
```

3. **Protected:** Accessible within package and outside the package but through inheritance only. It can be applied on the data member, method and constructor and can't be applied on the class.

   Example

   The following parent class uses protected access control, to allow its child class override *openSpeaker()* method –

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}

class StreamingAudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
  } }
```

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

   Example

   The following function uses public access control −

```
public static void main(String[] arguments) {
  // ...
}
```

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

Introducing Nested and Inner Classes

- **Java inner class** or nested class is a class which is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

- Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- **Code Optimization**: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

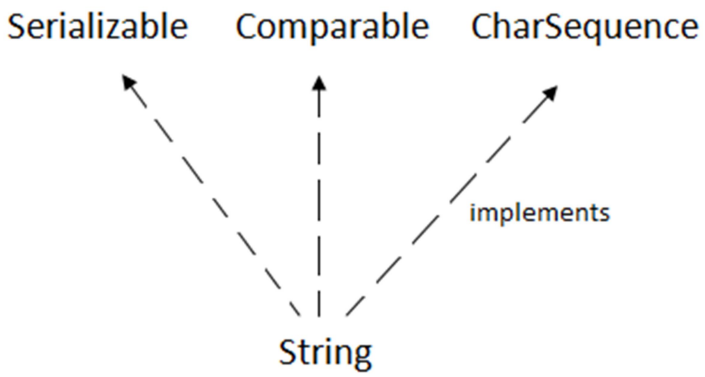
String Class

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
is same as:
String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



How to create a string object?
There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal
Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

.

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

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

5. Inheritance

Inheritance basics, member access and inheritance. Overriding: Method overriding, super keyword, polymorphism and virtual function.

Questions

1. Explain inheritance with example and its types.
2. Explain Method overloading.
3. Explain Super Keyword.
4. Explain Polymorphism.
5. Explain Virtual Function.

Inheritance in Java

The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **inheritance**.
The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

- In Java, when an "Is-A" relationship exists between two classes we use Inheritance
- The parent class is termed super class and the inherited class is the sub class
- The keyword "extend" is used by the sub class to inherit the features of super class
- Inheritance is important since it leads to reusability of code

Child Class:
The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:
The class whose properties and functionalities are used (inherited) by another class is known as parent class, super class or Base class.

Syntax:

```
class subClass extends superClass
{
  //methods and fields
}
```

Advantage of Inheritance

- It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
- It promotes polymorphism by allowing method overriding.

Disadvantages of Inheritance

- Main disadvantage of using inheritance is that the two classes (parent and child class) gets **tightly coupled**.
- This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

Inheritance Example

In this example, we have a base class Teacher and a sub class PhysicsTeacher. Since class PhysicsTeacher extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class. Here we have collegeName, designation and work() method which are common to all the teachers so we have declared them in the base class, this way the child classes like MathTeacher, MusicTeacher and PhysicsTeacher do not need to write this code and can be used directly from base class.

```
class Teacher {
  String designation = "Teacher";
  String collegeName = "Beginnersbook";
  void does(){
    System.out.println("Teaching");
  }
}

public class PhysicsTeacher extends Teacher{
  String mainSubject = "Physics";
  public static void main(String args[]){
    PhysicsTeacher obj = new PhysicsTeacher();
    System.out.println(obj.collegeName);
    System.out.println(obj.designation);
    System.out.println(obj.mainSubject);
    obj.does();
  }
}
```

Output:

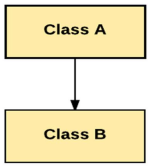
```
Beginnersbook
Teacher
Physics
Teaching
```

Types of Inheritance

There are various types of inheritance in Java:

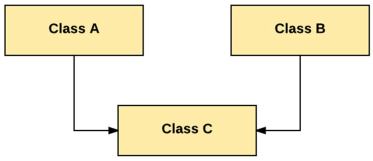
1. Single Inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. **Single Inheritance:**
   In Single Inheritance one class extends another class (one class only).



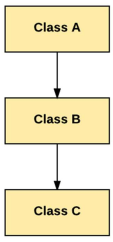
In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

2. **Multiple Inheritance:**
   In Multiple Inheritance, one class extending more than one class. Java does not support multiple inheritance.



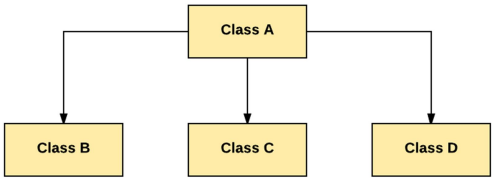
   As per above diagram, Class C extends Class A and Class B both.

3. **Multilevel Inheritance:**
   In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.



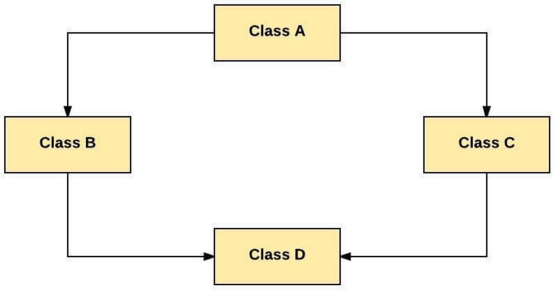
   As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

4. **Hierarchical Inheritance:**
   In Hierarchical Inheritance, one class is inherited by many sub classes.



   As per above example, Class B, C, and D inherit the same class A.

5. **Hybrid Inheritance:**
   Hybrid inheritance is a combination of Single and Multiple inheritances.



   As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

Method Overriding

- If child class has the same method as declared in the parent class, it is known as method overriding
- 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.
- Declaring a method in sub class which is already present in parent class is known as method overriding
- In this case the method in parent class is called overridden method and the method in child class is called overriding method.
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for method overriding:

- In java, a method can only be written in Subclass, not in same class.
- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level.
  For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. Constructors cannot be overridden.

Method Overriding Example

```
class Human{
  //Overridden method
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  //Overriding method
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }
}
```

Output:

```
Boy is eating
```

Super Keyword

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class.
It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
```

```
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
barking...
```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.
Let's see a simple example:

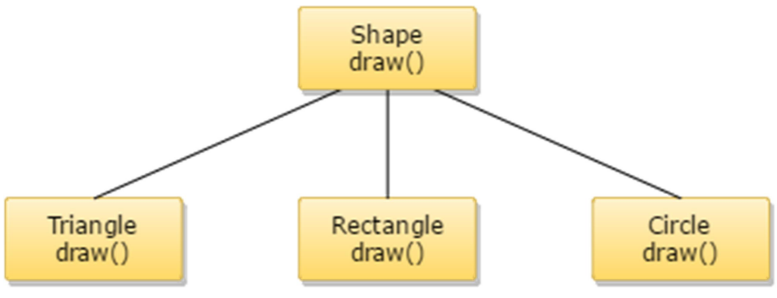
```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Polymorphism

- Polymorphism is a OOPs concept where one name can have many forms.



Polymorphism

- For example, you have a smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called Polymorphism.
- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Any Java object that can pass more than one IS-A test is considered to be polymorphic.
- In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
- It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
- The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
- A reference variable can refer to any object of its declared type or any subtype of its declared type.
- A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

Example

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Function

In object-oriented programming, a virtual function or virtual method is a function or method whose behaviour can be overridden within an inheriting class by a function with the same signature to provide the polymorphic behaviour.

Therefore according to definition, every non-static method in JAVA is by default *virtual method* except final and private methods. The methods which cannot be inherited for polymorphic behaviour is not a virtual method.

A virtual function a member function which is declared within base class and is re-defined (Over-ridden) by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

- They must be declared in public section of class.
- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

6. Packages and Interfaces

Defining, Creating and accessing a package, Understanding CLASSPATH, Importing packages, difference between classes and interface, defining an interface, implementing interface, applying interface, variable in interface and extending interface, Exploring java io.

Questions

1. Define Package and also explain creating and accessing a package.
2. Difference between class and interface.
3. Define interface and explain implementing an interface.
4. Exploring java i.o Stream and its type.

Defining Packages

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.
- Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Syntax

```
package nameOfPackage;
```

Types of packages in Java
There are following two types of packages in java-

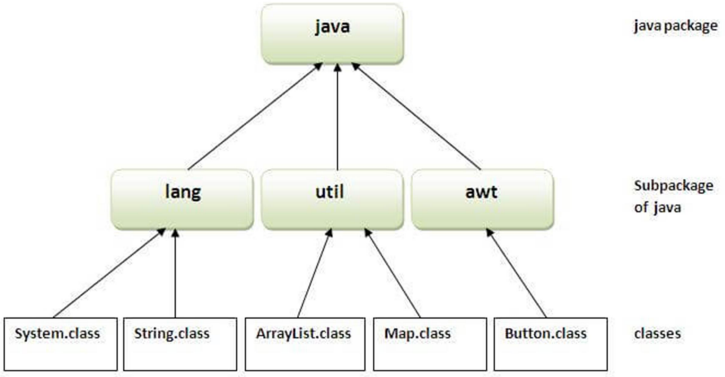
1) User defined package
The package we create is called user-defined package.

2) Built-in package
These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

- **java.lang:** Contains language support classes (e.g. classed which defines primitive data types, math operations). This package is automatically imported.
- **java.io:** Contains classed for supporting input / output operations.
- **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support; for Date / Time operations.
- **java.applet:** Contains classes for creating Applets.
- **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button, menus etc).
- **java.net:** Contain classes for supporting networking operations.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



Simple example of java package

The package keyword is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

Creating and Accessing a Package

While creating a package, you should choose a name for the package and include a package statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called animals. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* −

```
/* File name : Animal.java */
package animals;

interface Animal {
   public void eat();
   public void travel();
}
```

Now, let us implement the above interface in the same package *animals* −

```
package animals;
/* File name : MammalInt.java */

public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }

  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();

  }

}
```

Now compile the java files as shown below −

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name animals will be created in the current directory and these class files will be placed in it as shown below.

Output

You can execute the class file within the package and get the result as shown below.

```
Mammal eats
Mammal travels
```

Difference between Classes and Interface

- Class and Interface both are used to create new reference types.
- A class is a collection of fields and methods that operate on fields
- An interface has fully abstract methods i.e. methods with nobody.
- An interface is syntactically similar to the class but there is a major difference between class and interface that is a class can be instantiated, but an interface can never be instantiated.

Followings are some more difference between a class and interface in the comparison chart shown below.

BASIS FOR COMPARISON	CLASS	INTERFACE
Basic	A class is instantiated to create objects.	An interface can never be instantiated as the methods are unable to perform any action on invoking.
Keyword	“class” keyword should be used	“interface” keyword should be used
Access Specifier	The members of a class can be private,	The members of an interface are always

	public or protected.	public.
Methods	The methods of a class are defined to perform a specific action. Methods can be final and static	The methods in an interface are purely abstract. Methods should not be final and static
Implement/Extend	A class can implement any number of interfaces and can extend only one class.	An interface can extend multiple interfaces but cannot implement any interface.
Constructor	A class can have constructors to initialize the variables.	An interface can never have a constructor as there is hardly any variable to initialize.
Main Method	Can have main() method	Cannot have main() method as main() is a concrete method
Inheritance Support	Supports only multilevel and hierarchical inheritances but not multiple inheritance	Supports all types of inheritance – multilevel, hierarchical and multiple
Variable	Variables can be private	Variables should be public only

Defining an Interface

- 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.
- Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types.
- Method bodies exist only for default methods and static methods.
- Writing an interface is similar to writing a class. But a class describes the attributes and behaviours of an object. And an interface contains behaviours that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding byte-code file must be in a directory structure that matches the package name.

Rules for using Interface

- Methods inside Interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public static final variables (constants).
- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

```
interface Pet{
  public void test();
}
class Dog implements Pet{
   public void test(){
     System.out.println("Interface Method Implemented");
  }
   public static void main(String args[]){
     Pet p = new Dog();
     p.test();
  }
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Implementing Interface

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
```

```
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Exploring Java io

- The Java I/O package, a.k.a. java.io, provides a set of input streams and a set of output streams used to read and write data to files or other input and output sources.
- There are three categories of classes in java.io: input streams, output streams and everything else.
- This package provides for system input and output through data streams, serialization and the file system.
- Java I/O (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast.
- The java.io package contains all the classes required for input and output operations.
- We can perform file handling in Java by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.
1) System.out: standard output stream
2) System.in: standard input stream
3) System.err: standard error stream

Let's see the code to print output and an error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get input from console.

```
int i=System.in.read();          //returns ASCII code of 1st character
System.out.println((char)i);   //will print the character
```

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

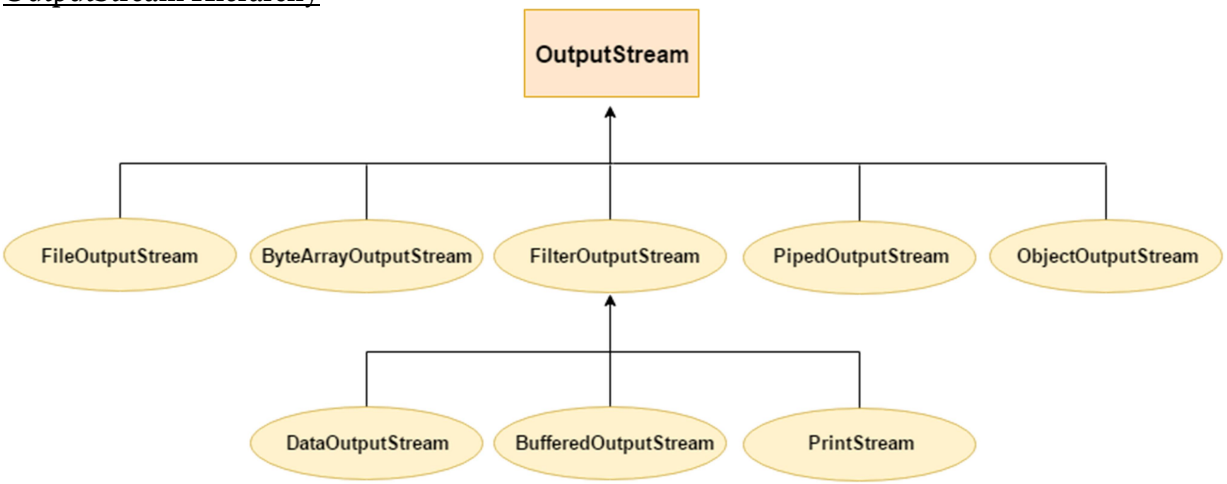
OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

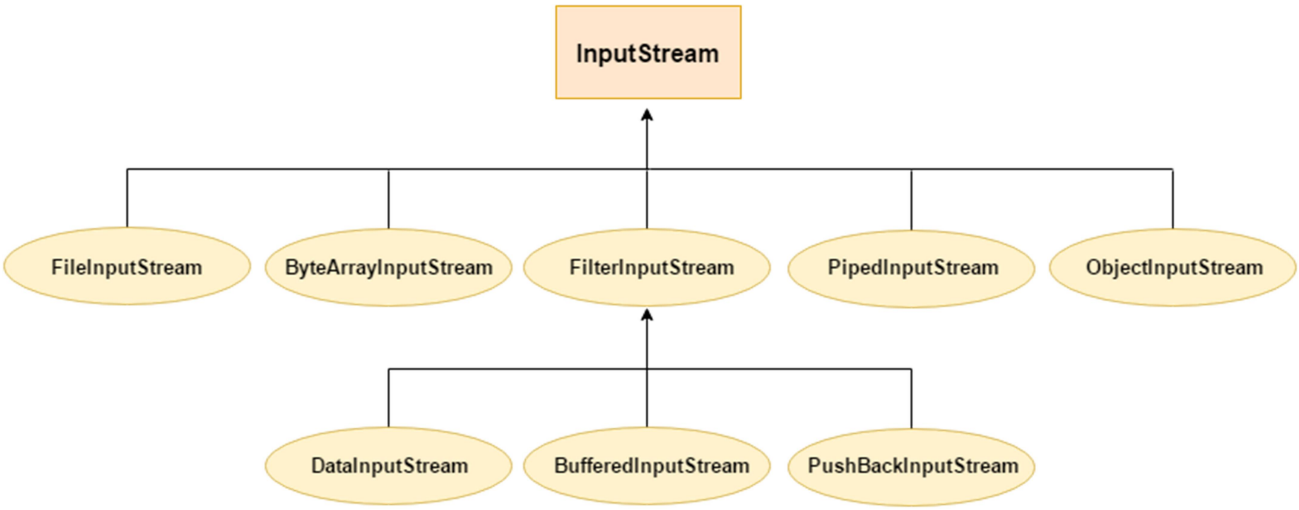
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy


InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
PipedInputStream
ObjectInputStream
DataInputStream
BufferedInputStream
PushBackInputStream

7. Exception Handling

Concept of exception handling, benefits of exception handling, termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating won exception sub classes. String handling, exploring java.util

Questions

1. What is Exception? Write a Program for exception handling in java and write its benefits also.
2. Explain Usage of Try, Catch, throw, throws and finally
3. Explain Built-in-exceptions
4. Explain String Handling.
5. Explain Java.util.

Concept of Exception Handling

- The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, which disrupts the normal flow of the program's instructions.

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.

Types of Exceptions

1. Checked Exceptions
   A checked exception is an exception that is checked (notified) by the compiler at compilation time; these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

   For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

   Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be
caught or declared to be thrown
    FileReader fr = new FileReader(file);
                  ^
1 error
```

2. Unchecked Exceptions

An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

```
public class Unchecked_Demo {

  public static void main(String args[]) {
    int num[] = {1, 2, 3, 4};
    System.out.println(num[5]);
  }
}
```

Output

If you compile and execute the above program, you will get the following exception.

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
   at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

Benefits of Exception Handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.
By handling we make sure that all the statements execute and the flow of program doesn't break.

Usage of Try, Catch, Throw, Throws and Finally

The try block

The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

The catch block

The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

Syntax

```
try
{
    //code that cause exception;
}
catch(Exception_type  e)
{
    //exception handling code
}
```

Example

A program to illustrate the uses of try and catch block:

```
class ExceptionDemo
{
    public static void main(String args[])
    {
        try
        {
            int arr[] = new int[5];
            arr[2] = 5;
            System.out.println("Access element two: " + arr[2]);
            arr[7] = 10;
            System.out.println("Access element seven: "+ arr[7]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
             System.out.println("Exception thrown:" + e);
        }
        System.out.println("End of the block");
    }
}
```

Output

Access element two: 5
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 7

The finally block

- The code present in finally block will always be executed even if try block generates some exception.
- Finally block must be followed by try or catch block.
- It is used to execute some important code.
- Finally block executes after try and catch block.

Syntax

```
try
{
    // code
}
catch(Exception_type1)
{
    // catch block1
}
Catch(Exception_type2)
{
    //catch block 2
}
finally
```

```
{
    //finally block
    //always execute
}
```

Example:
A program to implement finally block

```
class FinallyTest
{
    public static void main(String args[])
    {
        int arr[] = new int[5];
        try
        {
            arr[7] = 10;
            System.out.println("Seventh element value: " + arr[7]);
        }
        catch(ArrayIndexOutOfBoundsException ai)
        {
            System.out.println("Exception thrown : " + ai);
        }
        finally
        {
            arr[0] = 5;
            System.out.println("First element value: " +arr[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

Output
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 7
First element value: 5
The finally statement is executed

The throw keyword

- The throw keyword in Java is used to explicitly throw our own exception.
- It can be used for both checked and unchecked exception.

Syntax:
throw new Throwable_subclass;

Example
Programs to illustrate throw keyword in java:

```
public class ThrowTest
{
   static void test()
   {
       try
       {
           throw new ArithmeticException("Not valid ");
       }
       catch(ArithmeticException ae)
       {
           System.out.println("Inside test() ");
```

```
            throw ae;
        }
    }
    public static void main(String args[])
    {
        try
        {
            test();
        }
        catch( ArithmeticException ae)
        {
            System.out.println("Inside main(): "+ae);
        }
    }
}
```

Output:
Inside test()
Inside main(): java.lang.ArithmeticException: Not valid

The throws keyword
The throws keyword is generally used for handling checked exception.

- When you do not want to handle a program by try and catch block, it can be handled by throws.
- Without handling checked exceptions program can never be compiled.
- The throws keyword is always added after the method signature.

Example:
A program to illustrate uses of throws keyword:

```
public class ThrowsDemo
{
    static void throwMethod1() throws NullPointerException
    {
        System.out.println ("Inside throwMethod1");
        throw new NullPointerException ("Throws_Demo1");
    }
    static void throwMethod2() throws ArithmeticException
    {
        System.out.println("Inside throwsMethod2");
        throw new ArithmeticException("Throws_Demo2");
    }
    public static void main(String args[])
    {
        try
        {
            throwMethod1();
        }
        catch (NullPointerException exp)
        {
            System.out.println ("Exception is: " +exp);
        }
         try
         {
              throwMethod2();
         }
         catch(ArithmeticException ae)
```

```
                {
                    System.out.println("Exception is: "+ae);
                }
            }
        }
```

Output:
Inside throwMethod1
Exception is: java.lang.NullPointerException: Throws_Demo1
Inside throwsMethod2
Exception is: java.lang.ArithmeticException: Throws_Demo2

Built In Exceptions

- Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.
- Java defines several exception classes inside the standard package **java.lang**.
- The most general of these exceptions are subclasses of the standard type RuntimeException.
- Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.
- Java defines several other types of exceptions that relate to its various class libraries.

Following is the list of Java Unchecked RuntimeException.

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

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

S.N.	Exception & Description
1	**ClassNotFoundException** Class not found.
2	**CloneNotSupportedException** Attempt to clone an object that does not implement the Cloneable interface.
3	**IllegalAccessException** Access to a class is denied.
4	**InstantiationException** Attempt to create an object of an abstract class or interface.
5	**InterruptedException** One thread has been interrupted by another thread.
6	**NoSuchFieldException** A requested field does not exist.
7	**NoSuchMethodException** A requested method does not exist.

Examples of Built-in Exception:

1. Arithmetic Exception

```
class ArithmeticException_Demo {
public static void main(String args[])
   {
      try {
         int a = 30, b = 0;
         int c = a / b; // cannot divide by zero
         System.out.println("Result = " + c);
      }
      catch (ArithmeticException e) {
         System.out.println("Can't divide a number by 0");
      }
   }
}
```

Output:
Can't divide a number by 0

2. ArrayIndexOutOfBounds Exception

```
class ArrayIndexOutOfBound_Demo {
public static void main(String args[])
   {
      try {
         int a[] = new int[5];
         a[6] = 9; // accessing 7th element in an array of
         // size 5
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Array Index is Out Of Bounds");
      }
   }
}
```

Output:
Array Index is Out Of Bounds

String Handling

- String is probably the most commonly used class in java library.
- String class is encapsulated under java.lang package.
- In java, every string that you create is actually an object of type String.
- One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.
- Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

Creating a String object
String can be created in number of ways, here are a few ways of creating string object.

1) Using a String literal
String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
String str1 = "Hello";
```

2) Using another String object

```
String str2 = new String(str1);
```

3) Using new Keyword

```
String str3 = new String("Java");
```

4) Using + operator (Concatenation)

```
String str4 = str1 + str2;
or,
String str5 = "hello"+"Java";
```

Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as **string constant pool** inside the heap memory.

Exploring Java.util

It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Following are the Important Classes in Java.util package:

1. AbstractCollection:

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

2. AbstractList:
   This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

3. AbstractMap:
   This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.

4. AbstractMap.SimpleEntry:
   An Entry maintaining a key and a value.

5. AbstractMap.SimpleImmutableEntry:
   An Entry maintaining an immutable key and value.

6. AbstractQueue:
   This class provides skeletal implementations of some Queue operations.

7. AbstractSequentialList:
   This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).

8. AbstractSet:
   This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.

9. ArrayDeque:
   Resizable-array implementation of the Deque interface.

10. ArrayList:
    Resizable-array implementation of the List interface. Arrays: This class contains various methods for manipulating arrays (such as sorting and searching).

11. BitSet:
    This class implements a vector of bits that grows as needed.

12. Calendar:
    The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.

13. Collections:
    This class consists exclusively of static methods that operate on or return collections.

14. Currency:
    Represents a currency.

15. Date:
    The class Date represents a specific instant in time, with millisecond precision.

16. Dictionary:
    The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.

17. EnumMap:

A specialized Map implementation for use with enum type keys.

18. EnumSet:
    A specialized Set implementation for use with enum types.

19. EventListenerProxy:
    An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.

20. EventObject:
    The root class from which all event state objects shall be derived.

21. FormattableFlags:
    FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables.

22. Formatter:
    An interpreter for printf-style format strings.

23. GregorianCalendar:
    GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.

24. HashMap:
    Hash table based implementation of the Map interface.

25. HashSet:
    This class implements the Set interface, backed by a hash table (actually a HashMap instance).

8. Multithreading

Difference between multi-threading and multi-tasking, thread life cycle, creating threads, thread priorities, synchronizing threads inter thread communication, thread groups, daemon threads, enumerations, auto boxing annotations, generics.

Questions

1. Difference between multi-threading and multi-tasking.
2. Explain thread life cycle.
3. Explain thread priorities.

Difference between Multi-Threading And Multi-Tasking

Multithreading and Multitasking look similar but they are two different concepts.
A computer performs many tasks simultaneously. Multithreading and Multitasking both relate to computer performance. The key difference between multithreading and multitasking is that in multithreading, multiple threads are executing in a process concurrently and, in multitasking, multiple processes are running concurrently.

	Multithreading	Multitasking
Definition	Multithreading is to execute multiple threads in a process concurrently.	Multitasking is to run multiple processes on a computer concurrently.
Execution	In Multithreading, the CPU switches between multiple threads in the same process.	In Multitasking, the CPU switches between multiple processes to complete the execution.
Resource Sharing	In Multithreading, resources are shared among multiple threads in a process.	In Multitasking, resources are shared among multiple processes.
Complexity	Multithreading is light-weight and easy to create.	Multitasking is heavy-weight and harder to create.
Basic	Multithreading let CPU to execute multiple threads of a process simultaneously.	Multitasking let CPU to execute multiple tasks at the same time.
Switching	In multitasking CPU switches between programs frequently.	In multitasking CPU switches between programs frequently.
Memory and Resource	In multithreading system has to allocate memory to a process, multiple threads of that process shares the same memory and resources allocated to the process.	In multitasking system has to allocate separate memory and resources to each program that CPU is executing.

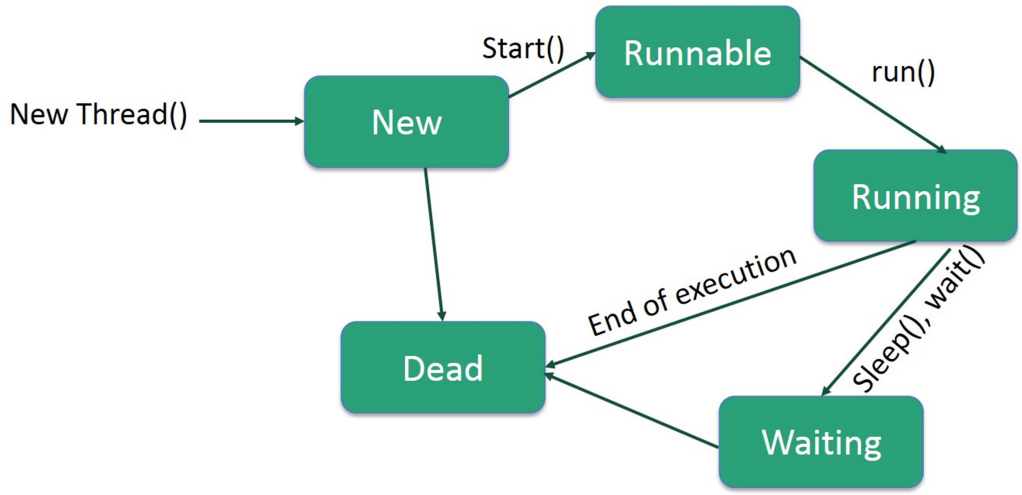
Thread Life Cycle

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

1. New
2. Runnable
3. Waiting
4. Timed Waiting
5. Terminated



Following are the stages of the life cycle −

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

Enumerations

- The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- Enumeration is a list of named constants, and these Java enumerations define a class type. By making enumerations into classes, the concept of enumeration is greatly expanded in Java.
- In the simplest form, Java enumerations appear similar to enumerations in other languages, except that the classes play a significant role in this concept.
- **Enum in java** is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH,

SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

- Java Enums can be thought of as classes that have fixed set of constants.
- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

The methods declared by Enumeration are summarized in the following table –

S.N.	Method & Description
1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

Example

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {

   public static void main(String args[]) {
      Enumeration days;
      Vector dayNames = new Vector();

      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();

      while (days.hasMoreElements()) {
         System.out.println(days.nextElement());
      }
   }
}
```

Output

```
Sunday
Monday
Tuesday
```

```
Wednesday
Thursday
Friday
Saturday
```

AutoBoxing

- Autoboxing and Unboxing features was added in Java5.
- Autoboxing is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
- Auto-Unboxing is a process by which the value of an object is automatically extracted from a type Wrapper class.
- Converting a primitive value into an object of the corresponding wrapper class is called autoboxing.
- For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:
    - Passed as a parameter to a method that expects an object of the corresponding wrapper class.
    - Assigned to a variable of the corresponding wrapper class.

Example

```
class BoxingExample1{
  public static void main(String args[]){
    int a=50;
        Integer a2=new Integer(a);  //Boxing
        Integer a3=5;                    //Boxing
        System.out.println(a2+" "+a3);
 }
}
```

Output

```
Output:50 5
```

Annotations

- Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.
- Annotations are used to provide supplement information about a program. There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.
- Annotations start with '**@**'.
- Annotations do not change action of a compiled program.

- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler. See below code for example.

Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Generics

- Java Generics were introduced in JDK 5.0 with the aim of reducing bugs and adding an extra layer of abstraction over types.
- Generics was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety.
- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Following are the rules to define Generic Methods –

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method −

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

Output

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

9. Event Handling

Events, Events sources, Event classes, Event Listeners, Delegation event model, handling mouse and key board events, Adapter classes. AWT class hierarchy, user interface components - labels, button, canvas, scrollbars, text components, check box, check box groups, choice, list panels – scroll pane, dialogs, menu bar, graphics, layout manager - layout manager types border, grid, flow card and grid bag.

Questions

1. Explain Events and Event Source in Java.
2. Explain Event Class.
3. Explain Event Listner.
4. Explain Delegation Event Model.
5. Explain Adapter Classes.
6. Explain different component of AWT.

Events

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.
- Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Types of Event

The events can be broadly classified into two categories:

Foreground Events

- Those events which require the direct interaction of user.
- They are generated as consequences of a person interacting with the graphical components in Graphical User Interface.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

Background Events

- Those events that require the interaction of end user are known as background events.
- Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Events Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add *Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener.

For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all

registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Classes

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

Following is the list of commonly used event classes.

1. **AWTEvent**
   It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
   Following is the declaration for **java.awt.AWTEvent** class:

```
public class AWTEvent
  extends EventObject
```

2. **ActionEvent**
   The ActionEvent is generated when button is clicked or the item of a list is double clicked.
   Following is the declaration for **java.awt.event.ActionEvent** class:

```
public class ActionEvent
extends AWTEvent
```

3. **InputEvent**
   The InputEvent class is root event class for all component-level input events.
   Following is the declaration for **java.awt.event.InputEvent** class:

```
public abstract class InputEvent
  extends ComponentEvent
```

4. **KeyEvent**
   On entering the character the Key event is generated.
   Following is the declaration for **java.awt.event.KeyEvent** class:

```
public class KeyEvent
  extends InputEvent
```

5. **MouseEvent**
   This event indicates a mouse action occurred in a component.

Following is the declaration for **java.awt.event.MouseEvent** class:

```
public class MouseEvent
  extends InputEvent
```

6. **TextEvent**
   The object of this class represents the text events.
   Following is the declaration for **java.awt.event.TextEvent** class:

```
public class TextEvent
  extends AWTEvent
```

7. **WindowEvent**
   The object of this class represents the change in state of a window.
   Following is the declaration for **java.awt.event.WindowEvent** class:

```
public class WindowEvent
  extends ComponentEvent
```

8. **AdjustmentEvent**
   The object of this class represents the adjustment event emitted by Adjustable objects.
   Following is the declaration for **java.awt.event.AdjustmentEvent** class:

```
public class AdjustmentEvent
  extends AWTEvent
```

9. **ComponentEvent**
   The object of this class represents the change in state of a window.
   Following is the declaration for **java.awt.event.ComponentEvent** class:

```
public class ComponentEvent
  extends AWTEvent
```

10. **ContainerEvent**
    The object of this class represents the change in state of a window.
    Following is the declaration for **java.awt.event.ContainerEvent** class:

```
public class ContainerEvent
  extends ComponentEvent
```

11. **MouseMotionEvent**
    The object of this class represents the change in state of a window.
    Following is the declaration for **java.awt.event.MouseMotionEvent** Class:

```
public class MouseMotionEvent
  extends InputEvent
```

12. **PaintEvent**
    The object of this class represents the change in state of a window.
    Following is the declaration for **java.awt.event.PaintEvent** class:

```
public class PaintEvent
  extends ComponentEvent
```

Event Listeners

- A *listener* is an object that is notified when an event occurs.
- It has two major requirements.
- First, it must have been registered with one or more sources to receive notifications about specific types of events.
- Second, it must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive

notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

AWT Event Listener Interfaces:

Following is the list of commonly used event listeners.

1. **Action Listener**
   This interface is used for receiving the action events.
   Following is the declaration for **java.awt.event.ActionListener** interface:

```
public interface ActionListener
  extends EventListener
```

2. **Component Listener**
   This interface is used for receiving the component events.
   Following is the declaration for **java.awt.event.ComponentListener**interface:

```
public interface ComponentListener
  extends EventListener
```

3. **ItemListener**
   This interface is used for receiving the item events.
   Following is the declaration for **java.awt.event.ItemListener** interface:

```
public interface ItemListener
  extends EventListener
```

4. **KeyListener**
   This interface is used for receiving the key events.
   Following is the declaration for **java.awt.event.KeyListener** interface:

```
public interface KeyListener
  extends EventListener
```

5. **MouseListener**
   This interface is used for receiving the mouse events.
   Following is the declaration for **java.awt.event.MouseListener** interface:

```
public interface MouseListener
  extends EventListener
```

6. **TextListener**
   This interface is used for receiving the text events.
   Following is the declaration for **java.awt.event.TextListener** interface:

```
public interface TextListener
  extends EventListener
```

7. **AdjustmentListener**
   This interface is used for receiving the adjustment events.
   Following is the declaration for **java.awt.event.AdjustmentListener**interface:

```
public interface AdjustmentListener
extends EventListener
```

8. **ContainerListener**
   This interface is used for receiving the container events.
   Following is the declaration for **java.awt.event.ContainerListener** interface:

```
public interface ContainerListener
extends EventListener
```

9. **MouseMotionListener**
   This interface is used for receiving the mouse motion events.
   Following is the declaration for **java.awt.event.MouseMotionListener**interface:

```
public interface MouseMotionListener
extends EventListener
```

10. **FocusListener**
    This interface is used for receiving the focus events.
    Following is the declaration for **java.awt.event.FocusListener** interface:

```
public interface FocusListener
extends EventListener
```

Delegation Event Model

- The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to "delegate" the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Using the Delegation Event Model

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

Adapter Classes.

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exist as convenience for creating listener objects.

Following is the list of commonly used adapters while listening GUI events in AWT.

1. **FocusAdapter**
   An abstract adapter class for receiving focus events.
   Following is the declaration for **java.awt.event.FocusAdapter** class:

```
public abstract class FocusAdapter
```

```
  extends Object
    implements FocusListener
```

2. **KeyAdapter**
   An abstract adapter class for receiving key events.
   Following is the declaration for **java.awt.event.KeyAdapter** class:

```
public abstract class KeyAdapter
  extends Object
    implements KeyListener
```

3. **MouseAdapter**
   An abstract adapter class for receiving mouse events.
   Following is the declaration for **java.awt.event.MouseAdapter** class:

```
public abstract class MouseAdapter
  extends Object
    implements MouseListener, MouseWheelListener, MouseMotionListener
```

4. **MouseMotionAdapter**
   An abstract adapter class for receiving mouse motion events.
   Following is the declaration for **java.awt.event.MouseMotionAdapter** class:

```
public abstract class MouseMotionAdapter
  extends Object
    implements MouseMotionListener
```

5. **WindowAdapter**
   An abstract adapter class for receiving window events.
   Following is the declaration for **java.awt.event.WindowAdapter** class:

```
public abstract class WindowAdapter
  extends Object
    implements WindowListener, WindowStateListener, WindowFocusListener
```

Abstract Window Toolkit (AWT)

- AWT contains large number of classes and methods that allows you to create and manage graphical user interface (GUI) applications, such as windows, buttons, scroll bars, etc.
- The AWT was designed to provide a common set of tools for GUI design that could work on a variety of platforms.
- The tools provided by the AWT are implemented using each platform's native GUI toolkit, hence preserving the look and feel of each platform. This is an advantage of using AWT.
- But the disadvantage of such an approach is that GUI designed on one platform may look different when displayed on another platform.
- AWT is the foundation upon which Swing is made i.e Swing is a set of GUI interfaces that extends the AWT. But nowadays AWT is merely used because most GUI Java programs are implemented using Swing because of its rich implementation of GUI controls and light-weighted nature.

Labels

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

AWT Label Class Declaration

```
public class Label extends Component implements Accessible
```

Button

The button class is used to create a labelled button that has platform independent implementation. The application result in some action when the button is pushed.

AWT Button Class declaration

public class Button extends Component implements Accessible

Check Box

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

AWT Checkbox Class Declaration

public class Checkbox extends Component implements ItemSelectable, Accessible

Check Box Groups

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

AWT CheckboxGroup Class Declaration

public class CheckboxGroup extends Object implements Serializable

Choice

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

AWT Choice Class Declaration

public class Choice extends Component implements ItemSelectable, Accessible

List Panels

The object of List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It inherits Component class.

AWT List class Declaration

public class List extends Component implements ItemSelectable, Accessible

Scroll Pane

The object of Scrollbar class is used to add horizontal and vertical scrollbar. Scrollbar is a GUI component allows us to see invisible number of rows and columns.

AWT Scrollbar class declaration

public class Scrollbar extends Component implements Adjustable, Accessible

Dialogs

The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize buttons.

Frame and Dialog both inherits Window class. Frame has maximize and minimize buttons but Dialog doesn't have.

Menu Bar

The object of MenuItem class adds a simple labelled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

Java CardLayout

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class

- CardLayout(): creates a card layout with zero horizontal and vertical gap.
- CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Grid Bag.

- The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.
- The components may not be of same size.
- Each GridBagLayout object maintains a dynamic, rectangular grid of cells.
- Each component occupies one or more cells known as its display area.
- Each component associates an instance of GridBagConstraints.
- With the help of constraints object we arrange component's display area on the grid.
- The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.

10. Applets.

Concept of Applets, difference between applets and application, life cycle of an applet, types of Applets, creating applets, passing parameters to applets. Swing - Introduction, limitations of AWT, MVC architecture, components, containers, exploring swing- JApplet, JFrame and JComponent, Icons and Labels, text fields, button – the JButton class, Check boxes, Radio buttons, Combo boxes, Tabbed Panes, Scroll Panes, Trees and Tables.

Questions

1. What is Applet? What is the difference between local and remote applet.
2. What is the difference between applets and applications?
3. Explain Life Cycle of an Applet.
4. Explain Swing.
5. Explain MVC Architecture.

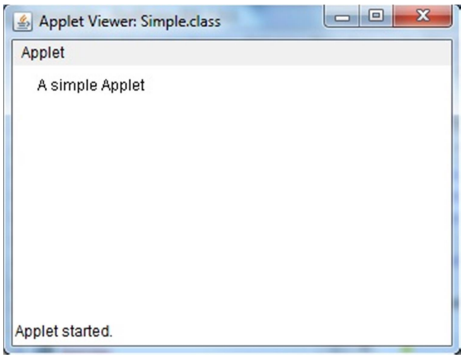
Concept of Applets

- An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.
- Applets are used to make the web site more dynamic and entertaining.
- Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as a part of a web document.
- After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.
- Any applet in Java is a class that extends the java.applet.Applet class.
- An Applet class does not have any main() method.
- It is viewed using JVM. The JVM can use either a plug-in of the Web browser or a separate runtime environment to run an applet application.
- JVM creates an instance of the applet class and invokes init() method to initialize an Applet.

A Simple Applet

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

Output

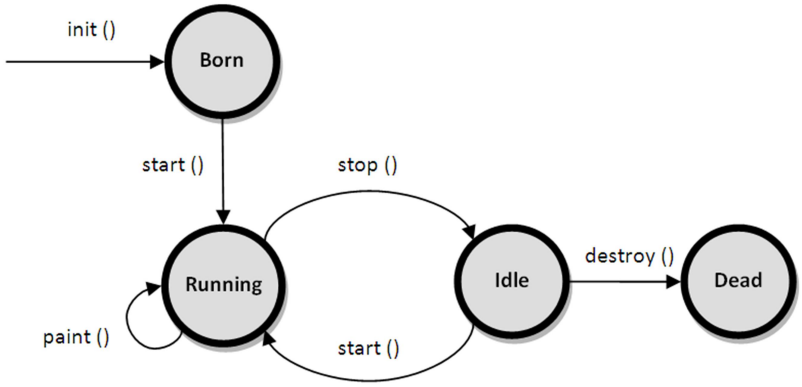


Difference between Applets And Application

Characteristic	Java Application	Java Applet
Definition	An application is a standalone Java program which can be run independently on client/server without the need of a web browser.	An applet is a form of Java program which is embedded with HTML page and loaded by a web server to be run on a web browser.
main() method	The execution of the program starts from the main() method.	There is no requirement of main() method for the execution of the program.
Access Restrictions	Application can access local disk files/ folders and network system.	Applet doesn't have access to the local network files and folders.
GUI	It doesn't require any Graphical User Interface (GUI).	It must run within GUI.
Environment for Execution	It requires Java Runtime Environment (JRE) for its successful execution.	It requires a web browser like Chrome, Firefox, etc for its successful execution.
requirement	Application program does not require any HTML code.	Applet requires HTML code.
Use	The application programs are used to develop application.	Applets are used for creating dynamic and interactive web application.
output	System.out.println() method is used to display output on screen.	drawstring () method is used to display output on screen.
Constructor	Application program contain Parameterized Constructor.	Applet does not contain any Parameterized Constructor.
destroy() method	Application program does not contain any destroy() method to destroy any application.	Applet contain destroy() method to destroy an application.
Restrictions	Can access any data or file available on the system.	Applets cannot access files residing on the local computer.
Read and write operation	Applications are capable of performing those operations to the files on the local computer.	The files cannot be read and write on the local computer through applet.
Execution	Can run alone but require JRE.	Cannot run independently require API's (Ex. Web API).

Life Cycle of an Applet

The life cycle of an applet is as shown in the figure below:



As shown in the above diagram, the life cycle of an applet starts with init() method and ends with destroy() method. Other life cycle methods are start(), stop() and paint(). The methods to execute only once in the applet life cycle are init() and destroy(). Other methods execute multiple times.

Following are the methods.

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

Below is the description of each applet life cycle method:

1. **init():**
   - The init() method is the first method to execute when the applet is executed.
   - Variable declaration and initialization operations are performed in this method.

2. **start():**
   - The start() method contains the actual code of the applet that should run.
   - The start() method executes immediately after the init() method.
   - It also executes whenever the applet is restored, maximized or moving from one tab to another tab in the browser.

3. **stop():**
   - The stop() method stops the execution of the applet.
   - The stop() method executes when the applet is minimized or when moving from one tab to another in the browser.

4. **destroy():**
   - The destroy()method executes when the applet window is closed or when the tab containing the webpage is closed.
   - stop() method executes just before when destroy() method is invoked.
   - The destroy() method removes the applet object from memory.

5. **paint():**
   - The paint() method is used to redraw the output on the applet display area.
   - The paint() method executes after the execution of start() method and whenever the applet or browser is resized.

The method execution sequence when an applet is executed is:

- init()
- start()
- paint()

The method execution sequence when an applet is closed is:

- stop()
- destroy()

Example program that demonstrates the life cycle of an applet is as follows:

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet
{
        public void init()
                {
                System.out.println("Applet initialized");
```

```
        }
    public void start()
        {
        System.out.println("Applet execution started");
        }
    public void stop()
        {
        System.out.println("Applet execution stopped");
        }
    public void paint(Graphics g)
        {
        System.out.println("Painting...");
        }
    public void destroy()
        {
        System.out.println("Applet destroyed");
        }
}
```

Swing - Introduction

- Swing is an advanced GUI toolkit. It has a rich set of widgets. From basic widgets like buttons, labels, scrollbars to advanced widgets like trees and tables. Swing itself is written in Java.
- Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT).
- Swing provides a look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform.
- It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.
- Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent. The term "lightweight" is used to describe such an element.
- Swing toolkit is:
    - platform independent
    - customizable
    - extensible
    - configurable
    - lightweight

- Swing is a part of JFC, Java Foundation Classes. It is a collection of packages for creating full featured desktop applications. JFC consists of AWT, Swing, and Accessibility, Java 2D, and Drag and Drop. Swing was released in 1997 with JDK 1.2. It is a mature toolkit.
- The Java platform has Java2D library, which enables developers to create advanced 2D graphics and imaging.

MVC Architecture

MVC is an architecture that separates business logic, presentation and data.
In MVC,

- M stands for Model
- V stands for View
- C stands for controller.

MVC is a systematic way to use the application where the flow starts from the view layer, where the request is raised and processed in controller layer and sent to model layer to insert data and get back the success or failure message.

Model Layer:

- This is the data layer which consists of the business logic of the system.
- It consists of all the data of the application
- It also represents the state of the application.
- It consists of classes which have the connection to the database.
- The controller connects with model and fetches the data and sends to the view layer.
- The model connects with the database as well and stores the data into a database which is connected to it.

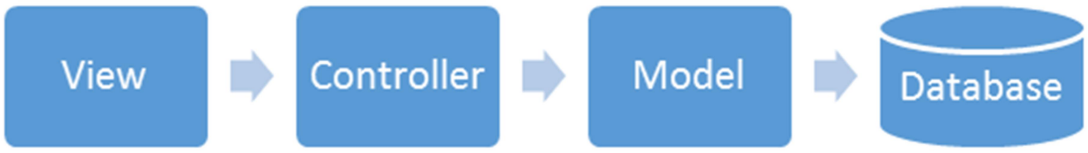
View Layer:

- This is a presentation layer.
- It consists of HTML, JSP, etc. into it.
- It normally presents the UI of the application.
- It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.
- This view layer shows the data on UI of the application.

Controller Layer:

- It acts as an interface between View and Model.
- It intercepts all the requests which are coming from the view layer.
- It receives the requests from the view layer and processes the requests and does the necessary validation for the request.
- This request is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.

The diagram is represented below:



The advantages of MVC are:

- Easy to maintain
- Easy to extend
- Easy to test
- Navigation control is centralized

Example of MVC architecture

In this example, we are going to show how to use MVC architecture

- We are taking the example of a form with two variables "email" and "password" which is our view layer.
- Once the user enters email, and password and clicks on submit then the action is passed in mvc_servlet where email and password are passed.
- This mvc_servlet is controller layer. Here in mvc_servlet the request is sent to the bean object which act as model layer.
- The email and password values are set into the bean and stored for further purpose.
- From the bean, the value is fetched and shown in the view layer.