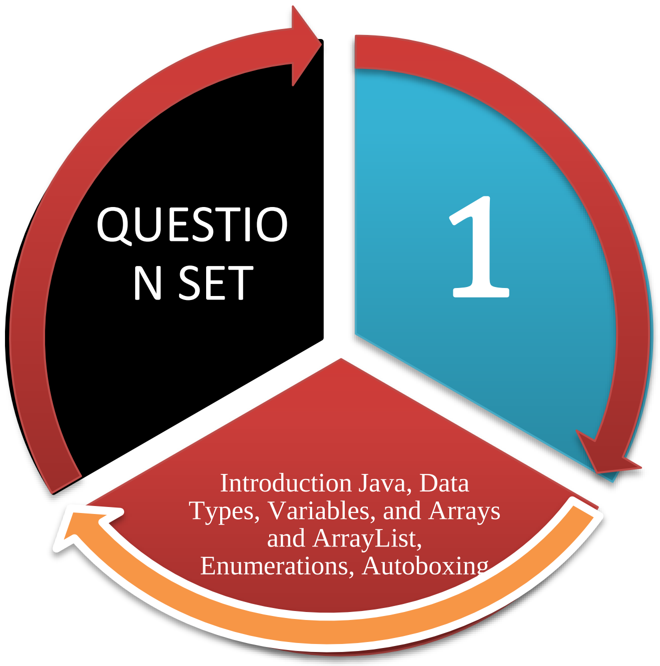
:


QUESTIO
N SET
1
Introduction Java, Data Types, Variables, and Arrays and ArrayList, Enumerations, Autoboxing

Question: Explain the structure of java program. CSE-2015,2013
Question: Show and discuss the general form of a java class?
Answer:
Structure of a Java program

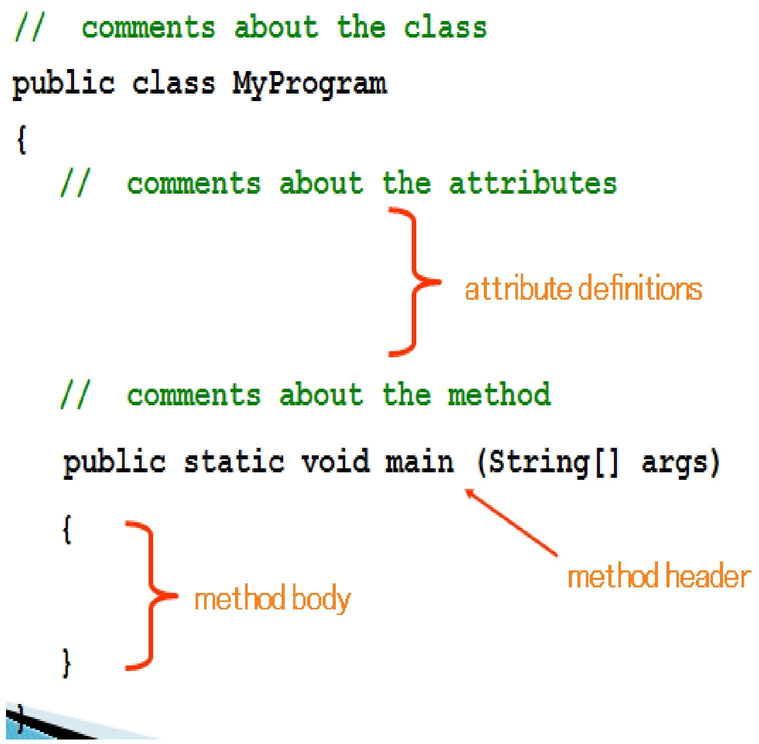
A Java program is a collection of classes. Each class is normally written in a separate file and the name of the file is the name of the class contained in the file, with the extension `.java`.

Public static void main(String[] args)

Here then, is a Java equivalent of the canonical Hello world program.

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

Here `println` is a static method in the class `System.out` which takes a string as argument and prints it out with a trailing newline character.
As noted earlier, we have to save this class in a file with the same name and the extension `.java`; that is, `helloworld.java`.



```
//  comments about the class
public class MyProgram
{
   //  comments about the attributes



   //  comments about the method
   public static void main (String[] args)
   {

   }
}
```

Question: Why main method static in java? Explain.

This is necessary because main () is called by the JVM before any objects are made. Since it is static it can be directly invoked via the class. Similarly, we use static sometime for user defined methods so that we need not to make objects. Void indicates that
the main() method being declared does not return a value.

```
class MyApplication {
  public MyApplication(){
    // Some init code here
  }
  public void main(String[] args){
    // real application code here
  }
}
```

Question: In System.out.println() what is System, out and println? CSE-2013
Ans:

It is provide soft output on console. System is a class in java.lang package. out is the static data member in System class and reference variable of PrintStream class. Println() is a normal (overloaded)
Method of PrintStream class

Question: What is the role of java virtual machine? CSE-2013

JVM:
The output of a Java compiler is not executable code. Rather, it is byte code. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).

Question: Why is java called strangely typed language? What are rules for automatic type conversion of a variable in java? CSE-2013
Answer:
Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

When one type of data is assigned to another type of variable, an automatic type conversion will take
place if the following two conditions are met:
• The two types are compatible.
• The destination type is larger than the source type.

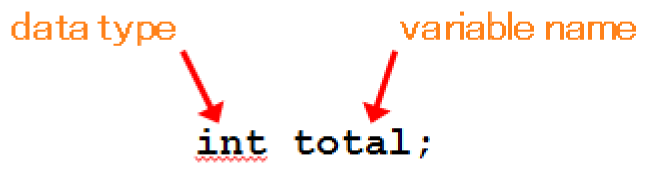
, there are no automatic conversions from the numeric types to char or
boolean. Also, char and boolean are not compatible with each other.
As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char

How does java garbage collection work? CSE-2013
Ans:
In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically.

The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most **part, you should not have to think about it while writing your programs**

Question: What is variable? What are the rules of variable naming?

Variables:
A variable is a name for a location in memory.
Or
A variable must be declared by specifying the variable's name and the type of information that it will hold



- Multiple variables can be created in one declaration:
- int count, temp, result;

Question: What is literal? Shortly explain integer, floating-point, Boolean, character and string literals used in java program?

Question: Write down the name of different primitive data types in java? Exam-2015
Literal: Java Literals are syntactic representations of Boolean, character, numeric, or string data. **Literals** provide a means of expressing specific values in your program.
Data types:
- Primitive Data Types
- Variables, Initialization, and Assignment
- Constants
- Characters
- Strings

There are eight primitive data types in Java
- Four of them represent integers:
  - byte, short, int, long
- Two of them represent floating point numbers:

- float, double

- One of them represents characters:
  - Char, String
- And one of them represents boolean values:
  - boolean

Type	Storage	Min Value	Max Value
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
float	32 bits	$+/- 3.4 \times 10^{38}$ with 7 significant digits	
Doubl	64 bits	$+/- 1.7 \times 10^{308}$ with 15 significant digits	

Question: Differentiate among instance variable, class variable, and local variable.

Local Variable	Instance Variable	Class Variable
Declared within the method	In a class, but outside a Method. Typically private.	In a class, but outside a method. Must be declared sta tic. Typically also final
Local variables hold values used in computations in a method.	Instance variables hold values that must be referenced by more than one method	Class variables are mostly used for constant s, variables that never change from their initial value.
Created when method or Constructor is entered. Destroyed on exit.	Created when instance of class is created With new.	Created when the program Starts.

Question: What is Instance Variable Hiding problem? How this problem can be solved?
Answer:

When a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If they had been, then width, for example, would have referred to the formal parameter, hiding the instance variable width. While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

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

```
this.width = width;
this.height = height;
this.depth = depth;
}
```

A word of caution: The use of this in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the
same names for clarity, and use this to overcome the instance variable hiding. It is a matter of taste which approach you adopt

(a) With example explain why you would declare a variable or method as static.
What are the restrictions of declaring a variable or method as static?
Ans:
To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to
any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.
Methods declared as static have several restrictions:
• They can only directly call other static methods.
• They can only directly access static data.
• They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)
If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
```

```
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

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

(b) How can you pass command line argument to a java console application?
Explain with an example.
Ans:
A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they
are stored as strings in a String array passed to the args parameter of main( ). The first command-line argument is stored at args[0], the second at args[1], and so on. For example, the following program displays all of the command-line arguments that it is called with:

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

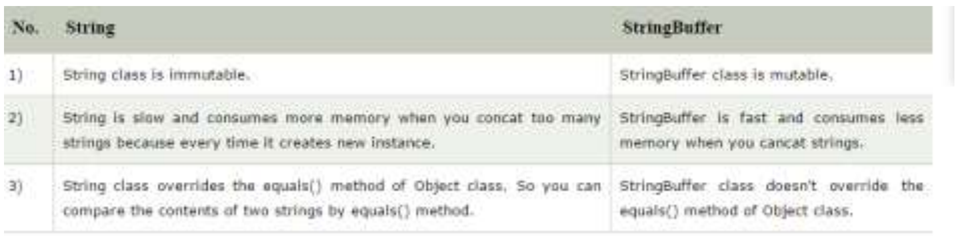
Try executing this program, as shown here:

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

When you do, you will see the following output:

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

© What is the difference between String and StringBuffer class?
Ans:



No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are
given below:

1. public class ConcatTest{
2. public static String concatWith

(e) What are the differences between the constants 7, '7' and ''7''?

Ans:

The constant 7 is the integer 7 (the number you get when you add 3 and 4). The constant '7' is a character constant consisting of the character `7' (the key between `6' and `8' on the keyboard, which a also has a `&' on it on mine). The constant "7" is a string constant consisting of one character, the character `7'.

Question: Discuss the scope and lifetime of a variable? CSE-2015

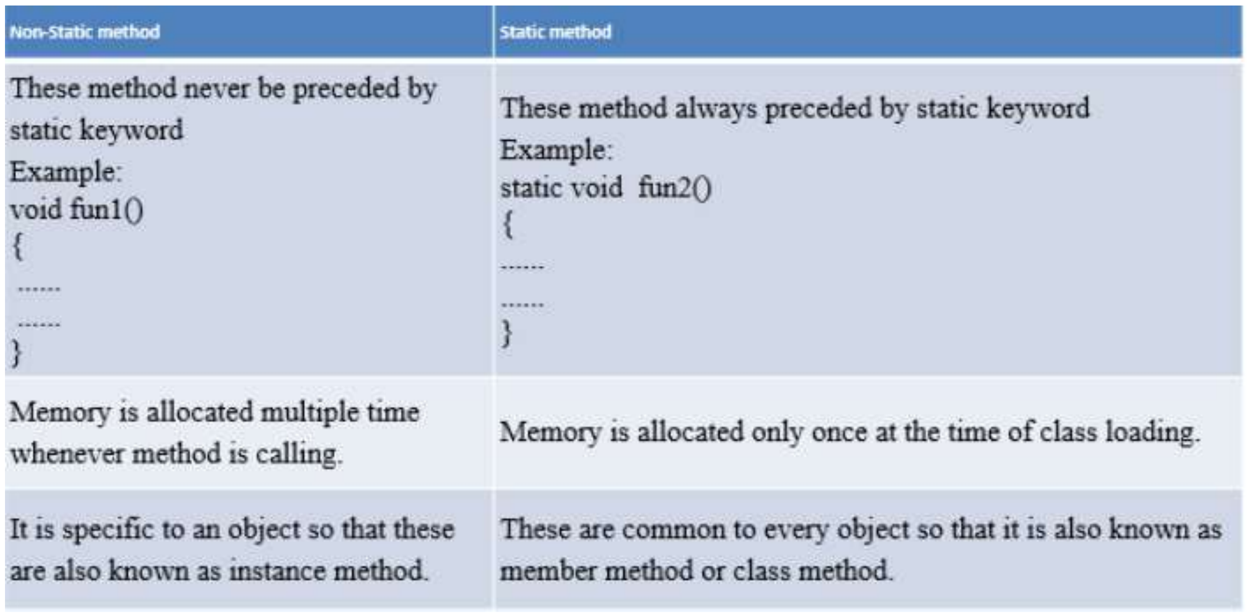
Scope of Variable:

- **Local scope:** "visible" within function or statement block from point of declaration until the end of the block.
- Class **scope:** "seen" by class members.
- Namespace **scope:** visible within namespace block.
- File **scope:** visible within current text file.
- Global **scope:** visible everywhere unless "hidden".

Lifetime of a Variable:

- **Static:** A static variable is stored in the data segment of the "object file" of a program. Its lifetime is the entire duration of the program's execution.
- **Automatic:** An automatic variable has a lifetime that begins when program execution enters the function or statement block or compound and ends when execution leaves the block. Automatic variables are stored in a "function call stack".
- **Dynamic:** The lifetime of a dynamic object begins when memory is allocated for the object (e.g., by a call to `malloc()` or using `new`) and ends when memory is deallocated (e.g., by a call to `free()` or using `delete`). Dynamic objects are stored in "the heap".

Question: What are the differences between static and non-static methods?
Answer:



Non-Static method	Static method

These method never be preceded by static keyword
Example:
void fun1()
{
......
......
}
	These method always preceded by static keyword
	Example:
	static void fun2()
	{
	......
	......
	}

Memory is allocated multiple time whenever method is calling.	Memory is allocated only once at the time of class loading.
It is specific to an object so that these are also known as instance method.	These are common to every object so that it is also known as member method or class method.

Question: What is final variable and final method? Write down the Reasons to use these. CSE-2013

Ans:

1) **Java final variable**

If you make any variable as final, you cannot change the value of final variable(It will be constant).
Example of final variable
There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike{
final int speedlimit=80;//final variable
void run(){
speedlimit=400;
}
public static void main(String args[]){
Bike  obj=new  Bike();
obj.run();
}
}//end of class
```

Question: With Example Explain why would declare a variable or method as static. What is the restriction of declaring a variable or method as static? Exam-2015

Benefits of static variables:

- constants can be defined without taking additional memory (one for each class)
- constants can be accessed without an instantiation of the class

Benefits of static methods:

- instance-independent behavior can be defined without fear of accidental interaction with an instance of the class

Example of Static variable:

Let's say you want to create a Vehicle class which has a variable color.
If you do a static variable it belongs straight to the vehicle class and you can call it:

```
Public class Vehicle {
Public static color = "red";
}
```

```
System.out.println(Vehicle.color)
```

This will return 'red'
If you do not do this as static you need to instantiate the class before calling the variable.

```
Public class Vehicle {
Public color = "red";
}
```

Vehicle car = new Vehicle;
System.out.println (car. Color);

Restriction of static member function:

There are several restrictions placed on **static** member functions. They may only directly refer to other **static** members of the class. (Of course, global functions and data may be accessed by **static** member Functions.)

- A **static** member function does not have this pointer.
- There cannot be a **static** and a non-**static** version of the same function.
- A **static** member function may not be virtual. Finally, they cannot be declared as **const** or **volatile.**

Question: Write down the difference between the following terms – CSE-2105, CSE-2015
Array and Array List 2. Method overriding and Method overloading

Array	**Array list**
Array is static in size that is fixed length data structure; One cannot change the length after creating the Array object.	ArrayList is dynamic in size . Each ArrayList object has instance variable *capacity* which indicates the size of the ArrayList. As elements are added to an ArrayList its capacity grows automatically.
ArrayList can contains primitive data types (like int , float , double)	ArrayList can not contains primitive data types (like int , float , double) it can only contains Object while Array can contain both primitive data types as well as objects.
Length of the ArrayList is provided by the user	Length of the ArrayList is provided by the size() method
Array can be multi-dimensional	ArrayList is always single dimensional.

Method Overriding	**Method Overloading**
Method overriding means having two methods with the same arguments, but different implementations.	Method **overloading** deals with the notion of having two or more methods in the same class with the same name but different arguments.

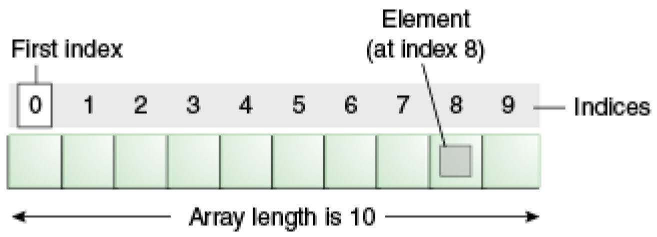
One of them would exist in the parent class, while another will be in the derived, or child class. The `@Override` annotation, while not required, can be helpful to enforce proper overriding of a method at compile time.
 	
`class Parent {`
`    void foo(double d) {`
`        // do something`
`    }`
`}`
 	
`class Child extends Parent {`
 	
`    @Override`
`    void foo(double d){`
	`void foo(int a)`
	`void foo(int a, float b)`

`// this method is overridden. } }`	
The real object type in the run-time, not the reference variable's type, determines which overridden method is used at *runtime*	reference type determines which overloaded method will be used at *compile time*.
Polymorphism applies to overriding	Polymorphism not to overloading.

Question: What is an array? Write a Java program that can print the value 0 to10 by using single dimension array. Exam-2014

Arrays:

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of same types of data,



Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

Syntax

dataType[] arrayRefVar;

Example

The following code snippets are examples of this syntax —

double[] myList; // preferred way.

You can create an array by using the new operator with the following syntax —
Syntax

arrayRefVar = new dataType[arraySize];

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList —

double[] myList = new double[10];

- nt a[]=new int[5];//declaration and instantiation

- 🕮 a[0]=10;//initialization
- 🕮 a[1]=20;
- 🕮 a[2]=70;
- 🕮 a[3]=40;
- 🕮 a[4]=50;

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.

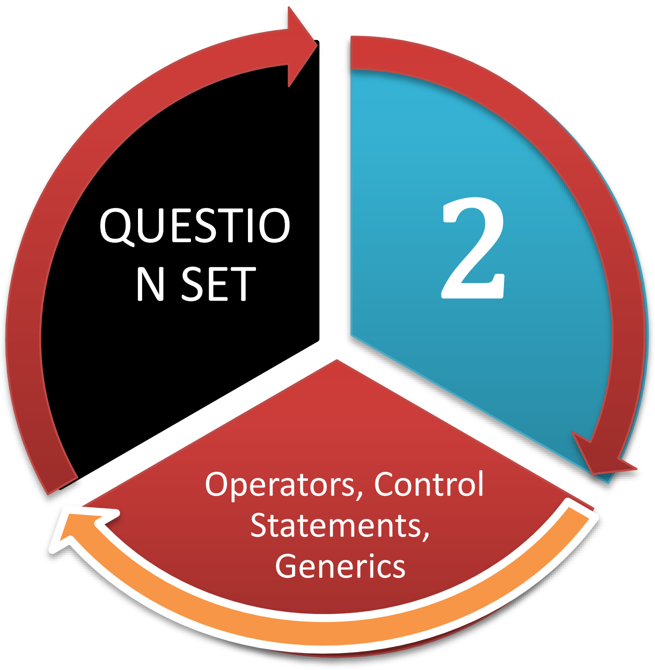
Processing Arrays

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays —

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }

    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);

    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
  }
}
```



Operator

An operator is a symbol that operates on one or more arguments to produce a result.

Operands

Operands are the values on which the operators act upon.

An operand can be:

- 🕮 A numeric variable - integer, floating point or character
- 🕮 Any primitive type variable - numeric and Boolean
- 🕮 Reference variable to an object
- 🕮 A literal - numeric value, Boolean value, or string.
- 🕮 An array element, "a[2]“
- 🕮 char primitive, which in numeric operations is treated as an unsigned two byte integer

Types of Operators

- 🕮 Assignment Operators
- 🕮 Increment Decrement Operators
- 🕮 Arithmetic Operators
- 🕮 Bitwise Operators
- 🕮 Relational Operators
- 🕮 Logical Operators
- 🕮 Ternary Operators

The for each Loops

For each loop enables us to traverse the complete array sequentially without using an index variable.

Example

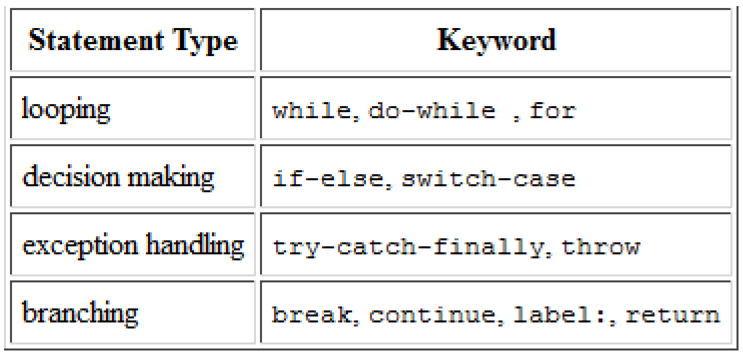
The following code displays all the elements in the array myList –

```
public class TestArray {

   public static void main(String[] args) {
     double[] myList = {1.9, 2.9, 3.4, 3.5};
```

```
        // Print all the array elements
        for (double element: myList) {
          System.out.println(element);
        }
      }
    }
```

Control flow:

The statements inside our source files are generally executed from top to bottom, in the order that they appear.



Statement Type	Keyword
looping	`while`, `do-while` , `for`
decision making	`if-else`, `switch-case`
exception handling	`try-catch-finally`, `throw`
branching	`break`, `continue`, `label:`, `return`

The If Statement

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program.

The simple if statement has the following syntax:

```
if (<conditional expression>)
   <statement action>
```

Below is an example that demonstrates conditional execution based on if statement condition.

```
public class IfStatementDemo {

        public static void main(String[] args) {
                int a = 10, b = 20;
                if (a > b)
                        System.out.println("a > b");
                if (a < b)
                        System.out.println("b < a");
        }
}
Output
b > a
```

The for Statement

The *for* statement provides a compact way to iterate over a range of values.

The general form of the `for` statement can be expressed like this:

```
for (initialization; termination; increment) {
   statement
}
```

The switch Statement

Use the switch statement to conditionally perform statements based on an integer expression.

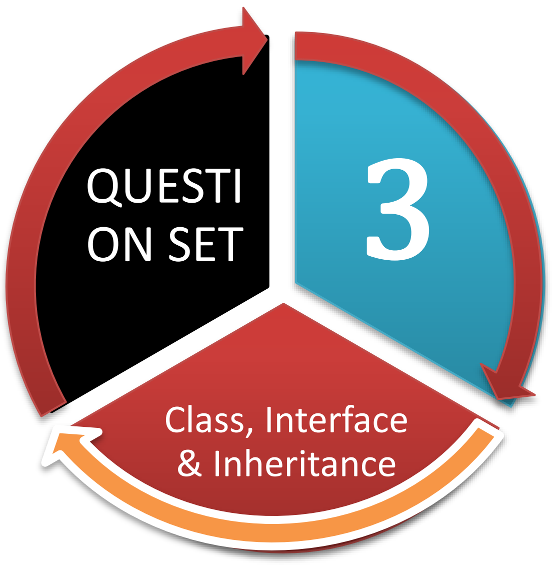
Following is a sample program, `SwitchDemo`, that declares an integer named `month` whose value supposedly represents the month in a date. The program displays the name of the month, based on the value of `month`, using the `switch` statement:

```
public class SwitchDemo {
   public static void main(String[] args) {
      int month = 8;
      switch (month) {
         case 1:  System.out.println("January"); break;
         case 2:  System.out.println("February"); break;
         case 3:  System.out.println("March"); break;
         case 4:  System.out.println("April"); break;
         case 5:  System.out.println("May"); break;
         case 6:  System.out.println("June"); break;
         case 7:  System.out.println("July"); break;
         case 8:  System.out.println("August"); break;
         case 9:  System.out.println("September"); break;
         case 10: System.out.println("October"); break;
         case 11: System.out.println("November"); break;
         case 12: System.out.println("December"); break;
      }
   }
}
```

The while and do-while Statements

We use a *while* statement to continually execute a block of statements while a condition remains true. The general syntax of the `while` statement is:

```
while (expression) {
   statement
}
```



Question: What are the difference between CPP and Java programming language? Discuss with example? CSE 2013

Difference between C++ and Java

C++	Java
Compatible with C source code, except for a few corner cases.	No backward compatibility with any previous language.
Write once, compile anywhere (WOCA).	Write once, run anywhere / everywhere (WORA / WORE).
Allows procedural programming, functional programming, object-oriented programming	Strongly encourages an object-oriented programming paradigm.
Allows direct calls to native system libraries.	Call through the Java Native Interface and recentlyJava Native Access.
Exposes low-level system facilities.	Runs in a virtual machine.
Only provides object types and type names.	Is reflective, allowing metaprogramming and dynamic code generation at runtime.
No standard inline documentation mechanism. Third-party software (e.g. Doxygen) exists.	Javadoc standard documentation.
Supports the goto statement.	Supports labels with loops and statement blocks.

Question: What is byte code? CSE-2014
Answer:

Bytecode is computer object code that is processed by a program, usually referred to as a virtual machine, rather than by the "real" computer machine, the hardware processor.

Question: What is object oriented language? Briefly discuss about the term Polymorphism? Exam-2013, CSE-2013
Answer:
Object Oriented Programming:

A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions or methods) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions or method

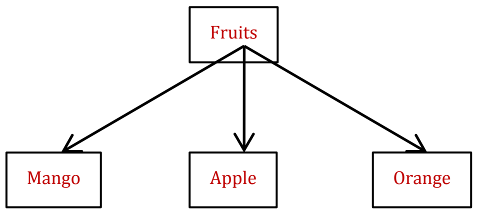
- An object-oriented programming language provide support for the following object oriented concepts
  - Class
  - Object
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Question: What is class? Give an example of class. Write down the general form of a class definition. CSE-2013
Question: what is class? How its accomplish data hiding? CSE 2014
Class:

A class can be defined as a template/blueprint that describes the behavior/state that the Object of its type support

- A class is simply a representation of a type of *object*
- For example, mango, apple, orange are the members of the class fruits
- If fruit has been defined as a class, then the statement

  Fruits mango;



- It will create an object mango belonging to the class fruit

- A class is created by using the keyword **class**

The general form of a class definition is

```
class classname {
// declare instance variables
type var1;
type var2;
 ...
type varN;
// declare methods
type method1(parameters)
{// body of method  }
type method2(parameters)
{// body of method }
// ...
type methodN(parameters)
{// body of method }
}
```

Example Program

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

public class Sample {
public static void main(String[] args) {
    int a = 30;
    int b = 45;
    Addi(a, b);
    Sub(a,b);
    }
public static void Addi(int a, int b) {
  int c = a+b;
  System.out.println("Addition = " + c);
 }

 public static void Sub(int a, int b) {
  int d = a+b;
  System.out.println("Sub = " + d);
  }
}
```

How class accomplish Data Hiding : class are combined data and function, and use private for data hiding.

Question: Define abstract class, concrete class and interface. Explain the use of abstract class.

1. Interface

An interface is just the declaration of methods of an Object, it's not the implementation

The point is interface cannot have any **concrete methods**. Concrete methods are those methods which have some code inside them; in one word - implemented. What your interface can have is static members and method signatures. The example below shall help you understand how to write an interface.

```
public interface Brain{
  public static final int number = 1;
  public void talk( String name );
  public abstract void doProgramming();
}
```

2. Abstract class

Abstract classes are a bit different from interfaces. These are also used to create blueprints for concrete classes but abstract classes may have implemented methods.

But to qualify as an abstract class, it must have at least one abstract method. Abstract classes can implement one or more interfaces and can extend one abstract class at most. There is a logical reason to this design which we will talk about later in this post. Here is an example of Abstract class creation.

```
public abstract class Car{
  public static final int wheels = 4;
  String turn( String direction ){
    System.out.println( "Turning" + direction );
  }
  public abstract void startWithSound( String sound );
  public abstract void shutdown( );
}
```

The declaration rules are as follows:

- A class can be an abstract class without having any methods inside it. But if it has any methods inside it, it must have at least one abstract method. This rule does not apply to static methods.
- As abstract classes can have both abstract and non abstract methods, hence the abstract modifier is necessary here (unlike in interface where only abstract methods are allowed).
- Static members are allowed.
- Abstract classes can extend other at most one abstract or concrete class and implement several interfaces.
- Any class that does not implement all the abstract methods of it's super class has to be an abstract class itself.

3. Concrete class

Concrete classes are the usual stuff that every java programmer has come across for sure. It is like the final implementation of a blueprint in case you are extending it some abstract super class.

A concrete class is complete in it and can extend and can be extended by any class.

```
public class Rocket{
  public static final int astronauts = 4;
  String turn( String direction ){
    System.out.println( "Turning" + direction );
  }
  public abstract void startWithSound( String sound ){
    System.out.println( "Engines on " + sound + "!!");
  }
  public abstract void shutdown( ){
    System.out.println( "Ignitions off !!" );
  }
}
```

Question: Give the syntax of interface. Differentiate between overriding and overloading. CSE-2014

```
interface Vehical {
            // declaration
            void changeGear(int newValue);
             void speedUp(int increment);
             void applyBrakes(int decrement);
}
class Car implements Vehical {
         int speed = 0;
         int gear = 1;
}
```

Question: In what ways do you initialize of instance field? When do you declare a method or class final? CSE 2014

As you have seen, you can often provide an initial value for a field in its declaration:

```
Public class BedAndBreakfast {

  // initialize to 10
  public static int capacity = 10;

  // initialize to false
  Private Boolean full = false;
}
```

Final method or class:

To protect overriding class and method used **Final Classes** and **Methods**.

Question: What is object? How are they creating from class? CSE 2014,2013

Object:

An Object is a software entity that models something in the real world. It has two main

Properties/ characteristics of Object:

- **State:** the object encapsulates information about itself - attributes or fields.
- **Behavior:** the object can do some things on behalf of other objects – methods

- **Identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each Object uniquely.

How create object:

```
    Public class Vehicle {
   Public static void main(string args[])
       {
         Vehicle minivan = new Vehicle ();
        }
```

Question: Write the different between class and object? CSE 2014
Answer:

Class	Object
A description of the *common properties of a* set of objects	A representation of the *properties of a single* instance
A concept	A phenomenon
A class is a part of a program	An object is part of data and a program execution
Example 1: Person	Example 1: Bill Clinton, Bono, Viggo Jensen

Question:
Question: Explain the following term of OOP. CSE-2014

Feature of JAVA/OOP:

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Abstraction:

Abstraction refers to the act of representing essential features without including the background details or explanations.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

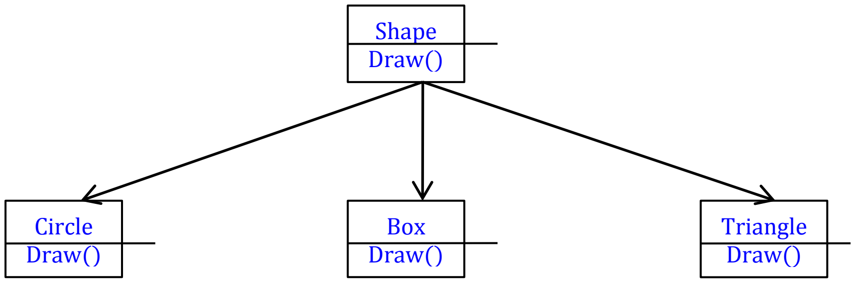
- The insulation from direct access by the program is called data hiding

Polymorphism:

Polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.

The behavior depends upon the type of data

- Ex: The operation of addition (Integer and string)



- Polymorphism play an important role in allowing objects having different internal structures to share the same external interface
- This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ

Question: Distinguish between encapsulation and polymorphism? CSE-2013

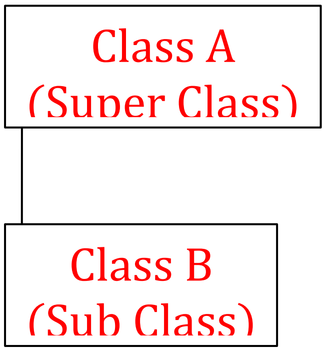
Encapsulation	Polymorphism
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse	Polymorphism is the attribute that allows one interface to control access to a general class of actions.
Its use for Security purpose	**Use for reusability of method and data**

Question: What is meant by inheritance and what are its advantages? Explain the use of supper () with example. CSE-2014

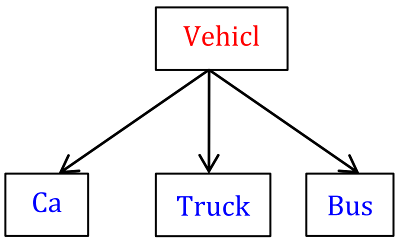
Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification.

```
class B extends A
{
.
.//additionsto,andmodifi
cationsof,
.//stuff inheritedfrom
```



Inheritance (Example):



```
class Vehicle {
   int registrationNumber;
            Person owner; / / ( Assuming t h a t a Person class has been def ined
            ! )
            void transferOwnership(Person newOwner) {
            . . . ……………………………………………..}
            . . .
            }
                 class Car extends Vehicle {
                     int numberOfDoors;
                     . . .
                 }
                 class Truck extends Vehicle {
                     int numberOfAxels;
                     . . .
                 }
                 class Bus extends Vehicle {
                     boolean hasSidecar;
                     . . .
               }
```

Advantage of inheritance:

- **Reusability** - facility to use public methods of base class without rewriting the same.
- **Extensibility** - extending the base class logic as per business logic of the derived class.
- **Data hiding** - base class can decide to keep some data private so that it cannot be altered by the derived class
- **Overriding** -With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class

Disadvantages:-

- One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the increased time/effort it takes the program to jump through all the levels of overloaded classes. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes
- Main disadvantage of using inheritance is that the two classes (base and inherited class) get tightly coupled. This means one cannot be used independent of each other.
- Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)

Super ():

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

- Super can be used to refer immediate parent class instance variable.
- Super can be used to invoke immediate parent class method.
- Super () can be used to invoke immediate parent class constructor.

Question: Explain why do need to use 'final' with inheritance? Cse-2014
Answer:

In Java we use final keyword with variables to specify its values are not to be changed. But I see that you can change the value in the constructor / methods of the class. Again, if the variable is static then it is a compilation error.

Question: What is constructor? What is specialized? CSE 2014
Constructor:
Constructor is a Spatial class member function that name is same as class name.

Properties;

Constructor function automatically called when an object created.

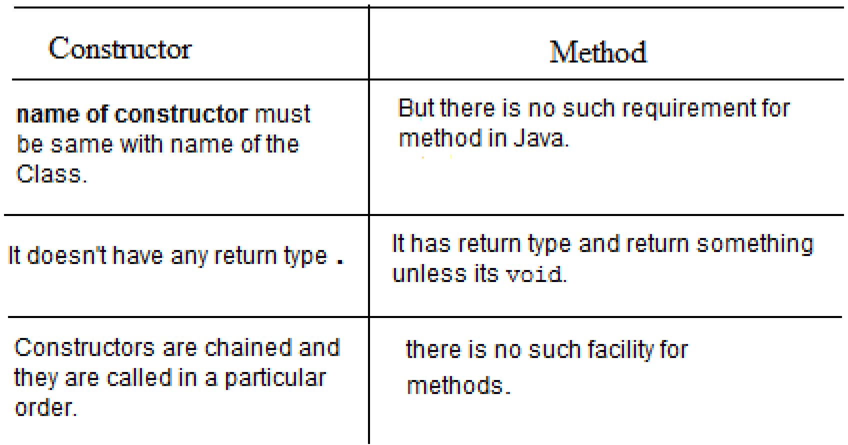
Question: how do invoked constructor? Explain with example? CSE 2014
Invoked constructor:

Constructor is invoked by created by object.

Example:

```
 Public Class A{

           A()
               { System.out.printf("Hello"); }
Public static void main(String args[])
                              {
                        A ob=new A();
                              }

             }
```

Question: Different between constructor and method?



Constructor	Method
name of constructor must be same with name of the Class.	But there is no such requirement for method in Java.
It doesn't have any return type .	It has return type and return something unless its `void`.
Constructors are chained and they are called in a particular order.	there is no such facility for methods.

Question: What is access specifier ? Explain about "object down casting" with example CSE-2013
Access Specifier:

Access Specifier is nothing but specify access of class member.

Question: What is of inheritance support in Java? Give an example that shows implementation of inheritance in Java. Exam-2014

Implementation of inheritance:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.
Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Question: Explain the various form of inheritance? CSE-2014

Types of inheritance in java:

On the basis of class, there can be three types of inheritance in java:

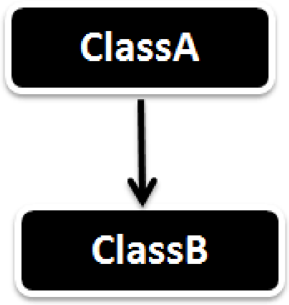
- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Single Inheritance in Java

Single Inheritance is the simple inheritance of all, When a class extends another class(Only one class) then we call it as Single inheritance.

The below diagram represents the single inheritance in java where **Class B** extends only one class **Class A.** Here **Class B** will be the **Sub class** and **Class A** will be one and only **Super class**.

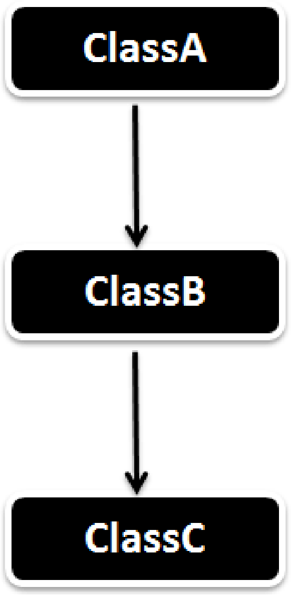
```
class A
{
  public void methodA()
  {
    System.out.println("Base              class
method");
  }
}
```



```
Class B extends A
{
  public void methodB()
  {
   System.out.println("Child class method");
  }
  public static void main(String args[])
  {
   B obj = new B();
   obj.methodA(); //calling super class method
   obj.methodB(); //calling local method
 }
}
```

Multilevel Inheritance
Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.

As you can see in below flow diagram C is subclass or child class of B and B is a child class of A. For more details and example refer – Multilevel inheritance in Java.

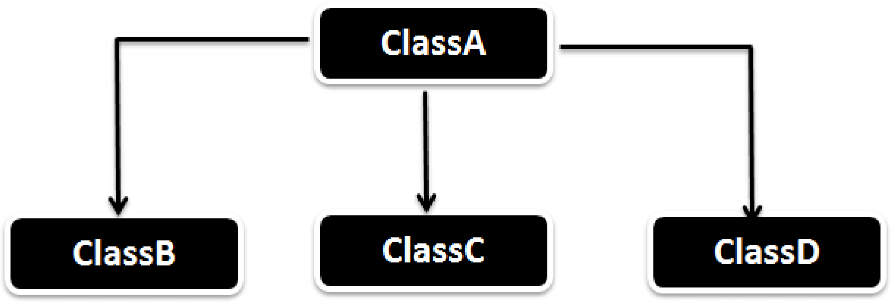


```
Class X
{
            public void methodX()
            {
             System.out.println("Class X method");
            }
        }
        Class Y extends X
        {
        public void methodY()
        {
        System.out.println("class Y method");
        }
        }
        Class Z extends Y
        {
          public void methodZ()
          {
           System.out.println("class Z method");
          }
          public static void main(String args[])
          {
           Z obj = new Z();
           obj.methodX(); //calling grand parent class method
           obj.methodY(); //calling parent class method
           obj.methodZ(); //calling local method
         }
}
```

Hierarchical Inheritance

In Hierarchical inheritance one parent class will be inherited by many sub classes. As per the below example ClassA will be inherited by ClassB, ClassC and ClassD. ClassA will be acting as a parent class for ClassB, ClassC and ClassD.

In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



```
class A
{
  public void methodA()
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
    System.out.println("method of Class B");
  }
}
class C extends A
{
 public void methodC()
 {
   System.out.println("method of Class C");
 }
}
class D extends A
{
 public void methodD()
 {
   System.out.println("method of Class D");
 }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
   //All classes can access the method of class A
   obj1.methodA();
   obj2.methodA();
   obj3.methodA();
 }
}
Output:
method of Class A
method of Class A
method of Class A
```

Question: What is wrapper class? Why should we need a wrapper class?

Answer:

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*.

Wrapper class Example: Primitive to Wrapper

```
Public class WrapperExample1{
Public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
Output:
20 20 20
```

Need of wrapper class:

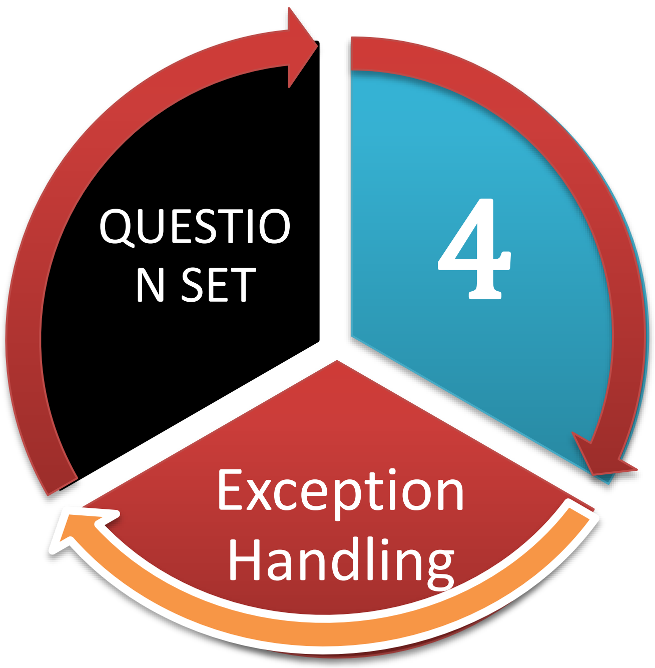
Wrapper classes are used to convert any data type into an object. The primitive data types are not objects; they do not belong to any **class**; they are defined in the language itself. Sometimes, it is required to convert data types into objects in **Java** language.

Question: Write the Advantage of Java?

Advantage of Java:

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- Java Is Secure
- Java Is Architecture-Neutral

- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic


QUESTIO
N SET
4
Exception
Handling

Question: What is an Exception? List some common types of exception that may occur in java. Exam-2015

Exception:

An exception is an indication of a problem that occurs during a program's execution. The name "exception" implies that the problem occurs infrequently if the "rule" is that a statement normally executes correctly, and then the "exception to the rule" is that a problem occurs.

Common types of exception that may occur in java:

ArithmeticException	You are trying to use your computer to solve a mathematical problem that you cannot solve yourself. Read up on your arithmetics and try again.
ArrayStoreException	You have used all your arrays and need to buy more from the array store.
RuntimeException	You cannot run fast enough, possibly due to obesity. Turn off your computer and go out and get som exercise.
SecurityException	You have been deemed a threat to nationaly security. Please sit still and wait for the authorities to come and get you.
AWTException	You are using AWT, which means your GUI will be ugly. This exception is only a warning and can be ignored.

IOException	IO stands for input/output and has to do with sending and recieving data. IO is a security problem and should not be used.

Question: What do you understand by exception handling? Explain with suitable codes?. Exam-2014

Exception Handling:

Exception handling enables programmers to create applications that can resolve (or handle) exceptions.

If an exception occurs within the try block, then it is thrown.
Example:

```
This is the general form of an exception-handling block:
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, ExceptionType is the type of exception that has occurred.

Example of ArithmeticException.
`InputMismatchException`

```
import java.util.Scanner;

  public class DivideByZeroNoExceptionHandling
 {
   // demonstrates throwing an exception when a divide-by-zero occurs
    public static int quotient( int numerator, int denominator )
     {
       return numerator / denominator; // possible division by zero
    } // end method quotient

  public static void main( String args[] )
   {
       Scanner scanner = new Scanner( System.in ); // scanner for input
       System.out.print( "Please enter an integer numerator: " );
       int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
       int denominator = scanner.nextInt();
       int result = quotient( numerator, denominator );
       System.out.printf(
          "\nResult: %d / %d = %d\n", numerator, denominator, result );
    } // end main
  } // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
       at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoException-
Handling.java:10)
       at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHan-
dling. java:22)
```

```
Please enter an integer numerator: 100
Please e

nter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHan-
dling. java:20)
```

Figure 13.2. Handling ArithmeticExceptions and InputMismatchExceptions.

Question: In detail explain the try-catch finally blocks of the exception handling. Exam-2015
`Answer:`
The try Block

The `try` block encases any statements that might cause an exception to occur.

For example,

if you are reading data from a file using the `FileReader` class its expected that you handle the `IOExceptions` associated with using a `FileReader` object (e.g, `FileNotFoundException`, `IOException`). To ensure this happens you can place the statements that deal with creating and using the `FileReader` object inside a `try` block:

```
 public static void main(String[] args){
FileReader fileInput = null;

try
{
//Open the input file
fileInput = new FileReader("Untitled.txt");
}
}
```

However, the code is incomplete because in order for the exception to be handled we need a place for it to be caught. This happens in the `catch` block.

The catch Block

The `catch` block(s) provide a place to handle the exception thrown by the statements within a `try` block. The `catch` block is defined directly after the `try` block.
It must specify the type of exception it is handling.

For example,

the `FileReader` object defined in the code above is capable of throwing a `FileNotFoundException` or an `IOException`. We can specify two `catch` blocks to handle both of those exceptions:

```
 public static void main(String[] args){
FileReader fileInput = null;

try
{
//Open the input file
fileInput = new FileReader("Untitled.txt");
}
catch(FileNotFoundException ex)
{
//handle the FileNotFoundException
}
catch(IOException ex)
{
```

```
//handle the IOException
}
}
```

In the `FileNotFoundException catch` block we could place code to ask the user to find the file for us and then try to read the file again.

The finally Block

The statements in the finally block are always executed. This is useful to clean up resources in the event of the try block executing without an exception and in the cases when there is an exception. In both eventualities, we can close the file we have been using.

The finally block appears directly after the last catch block:

```
public static void main(String[] args){
FileReader fileInput = null;

try
{
//Open the input file
fileInput = new FileReader("Untitled.txt");
}
catch(FileNotFoundException | IOException ex)
{
//handle both exceptions
}
finally
{
//We must remember to close streams
//Check to see if they are null in case there was an
//IO error and they are never initialised
if (fileInput != null)
{
fileInput.close();
}
}
}
```

Question: Describe the five keywords that manage Java exception handling. Exam-2013 CSE-2014
Answer:

- try
- catch
- finally
- throw
- throws

Although we have covered every keyword individually, let us summarize each keyword with few lines and finally one example covering each keyword in a single program

try block:

- The code which ***might raises exception*** must be ***enclosed within try block***
- try block must be followed by either catch block or finally

catch block:

- Contains ***handling code*** for any exception raised from ***corresponding try block*** and it must be enclosed within catch block
- catch block takes one argument which should be of ***type Throwable*** or ***one of its sub-classes*** i.e.; class-name followed by a variable
- ***Variable*** contains exception information for exception raised from try block

finally block:

- finally block is used to perform ***clean-up activities*** or ***code clean-up*** like closing database connection & closing streams or file resources, etc
- finally block is always ***associated*** with ***try-catch block***
- With finally block, there can be ***2 combinations***

Throw clause:

Sometimes, programmer can also *throw/raise exception explicitly at runtime* on the basis of some business condition

Pseudo code:

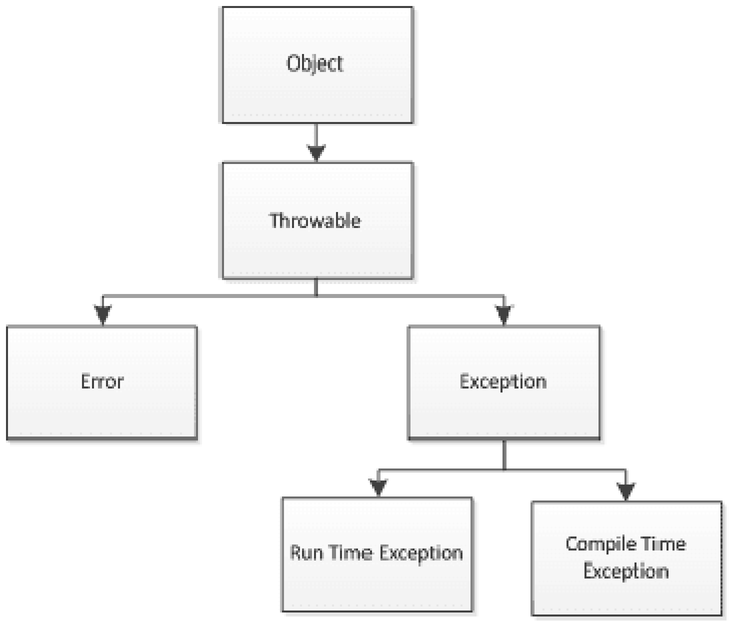
```
try {

        // some valid Java statements
        throw new RuntimeException();
    }

catch(Throwable th) {

        // handle exception here
        // or re-throw caught exception
```

throws keyword or throws clause:

- ***throws keyword*** is used to ***declare the exception*** that might raise during program execution

- ***throws clause*** is applicable for ***methods*** & ***constructor*** but strictly not applicable to classes
- It is mainly used for checked exception, as unchecked exception by default propagated back to the caller (i.e.; up in the runtime stack)

The exception is object created at the time of exceptional/error condition which will be thrown from the program and halt normal execution of the program. Java exceptions object hierarchy is as below:



Java's exceptions can be categorized into two types:

1. Checked exceptions
2. Unchecked exceptions

Unchecked exceptions come in two types:

I. Errors
II. Runtime exceptions

Checked Exceptions

Checked exceptions are the type that programmers should anticipate and from which programs should be able to recover. All Java exceptions are checked exceptions except those of the Error and RuntimeException classes and their subclasses.

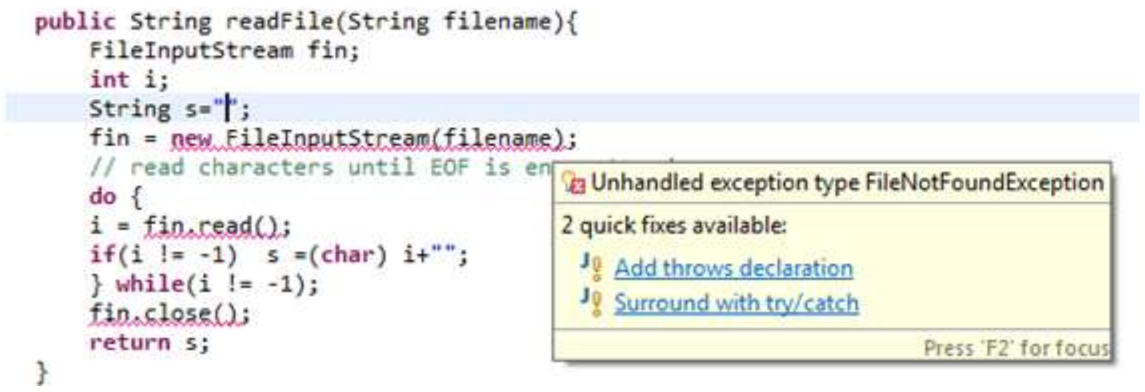
These could include subclasses of FileNotFoundException, UnknownHostException, etc.

Popular Checked Exceptions:

Name	**Description**
IOException	While using file input/output stream related exception
SQLException.	While executing queries on database related to SQL syntax
DataAccessException	Exception related to accessing data/database

ClassNotFoundException	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing .class file
InstantiationException	Attempt to create an object of an abstract class or interface.

Below example program of reading, file shows how checked exception should be handled.



```
public String readFile(String filename){
    FileInputStream fin;
    int i;
    String s="";
    fin = new FileInputStream(filename);
    // read characters until EOF is en
    do {
    i = fin.read();
    if(i != -1)  s =(char) i+"";
    } while(i != -1);
    fin.close();
    return s;
}
```

Unhandled exception type FileNotFoundException

2 quick fixes available:

Add throws declaration

Surround with try/catch

Press 'F2' for focus

Unchecked Exceptions

Unchecked exceptions inherit from the Error class or the RuntimeException class.

When an unchecked exception is thrown, it is usually caused by a misuse of code - passing a null or otherwise incorrect argument.

Popular Unchecked Exceptions:

Name	Description
NullPointerException	Thrown when attempting to access an object with a reference variable whose current value is null
ArrayIndexOutOfBound	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array)
IllegalArgumentException.	Thrown when a method receives an argument formatted differently than the method expects.
IllegalStateException	Thrown when the state of the environment doesn’t match the operation being attempted,e.g., using a Scanner that’s been closed.
NumberFormatException	Thrown when a method that converts a String to a number receives a String that it cannot convert.
ArithmaticException	Arithmetic error, such as divide-by-zero.

Question: Define runtime exception and IO exception with example Exam-2015

Runtime Exception

An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime.

I also use Runtime Exception for the default case of a switch statement if there is no better way of handling it.

```
public class RuntimeException
extends Exception
```

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

Example;

IoExcepton:

In general, I/O means Input or Output. Those methods throw the IOException whenever an input or output operation is failed or interpreted.

Note that this won't be thrown for reading or writing to memory as Java will be handling it automatically. Here are some cases which result in IOException.

IO Exception may occur –

- when you read some file from hard disk,
- You are trying to read/write a file and don't have permission,
- You were writing a file and disk space is not available anymore, etc.
- so consider the simple scenario, you are trying to read a file from hard disk - so you provided exact path of that file, but the file doesn't exist! so java would throw IO Exception.

RuntimeException vs Checked Exception in Java

Java Exceptions are divided in two categories RuntimeException also known as unchecked Exception and checked Exception.

Checked Exception	**RuntimeException**
, It is mandatory to provide try catch or try finally block to handle checked Exception and failure to do so will result in compile time error,	while in case of RuntimeException this is not mandatory
mandatory exception handling is not requirement for them	Any Exception which is subclass of RuntimeException are called unchecked
. . Popular example of checked Exceptions are ClassNotFoundException and IOException and that's the reason you need to provide a try catch finally block while performing file operations in Java as many of them throws IOException	Some of the most common Exception like NullPointerException, ArrayIndexOutOfBoundException are unchecked and they are descended from java.lang.RuntimeException

Question: With necessary code snippet, explain the difference between the keywords and throws. Exam-2015

Code Snipt:

Snippet is a programming term for a small region of re-usable source code, machine code, or text. Ordinarily, these are formally defined operative units to incorporate into larger programming modules.

Difference between the throw keywords and throws:

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

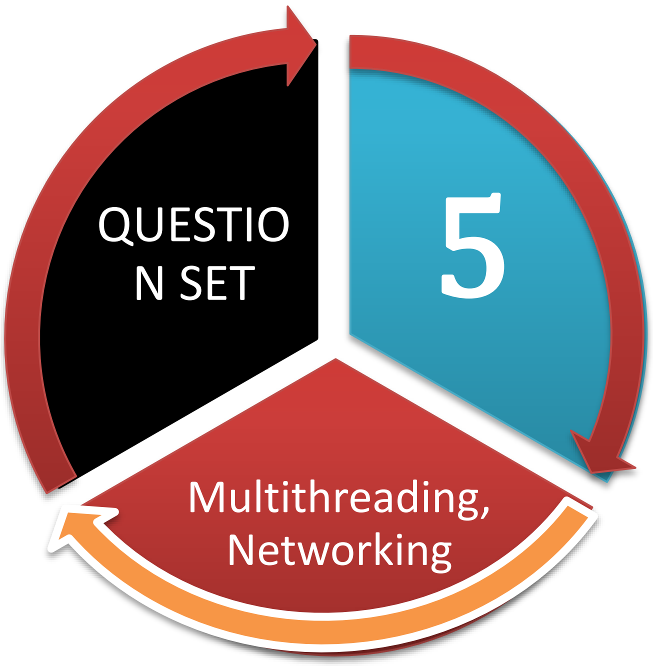
No.	**throw**	**throws**
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.

3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Throw vs Throws in java

1. **Throws clause** is used to declare an exception, which means it works similar to the try-catch block. On the other hand **throw** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.
For example:



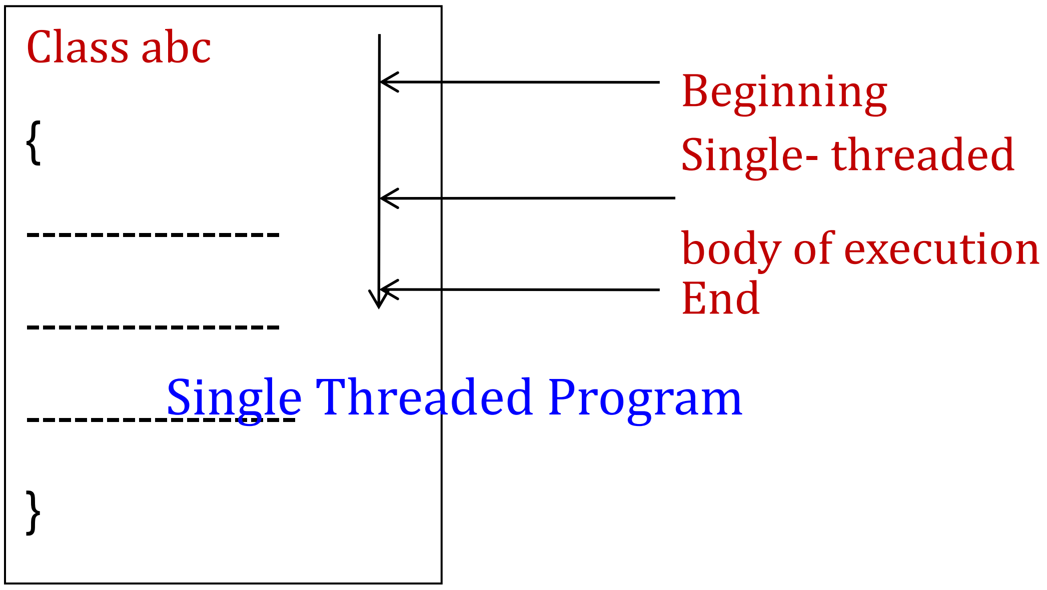
Question:What is Thread? Discuss Java Thread model? Exam-2015

Thread

A Thread is similar to a program that has a single flow of control On the other hand, threads are independent processes that can be run simultaneously

Example of Java Thread:

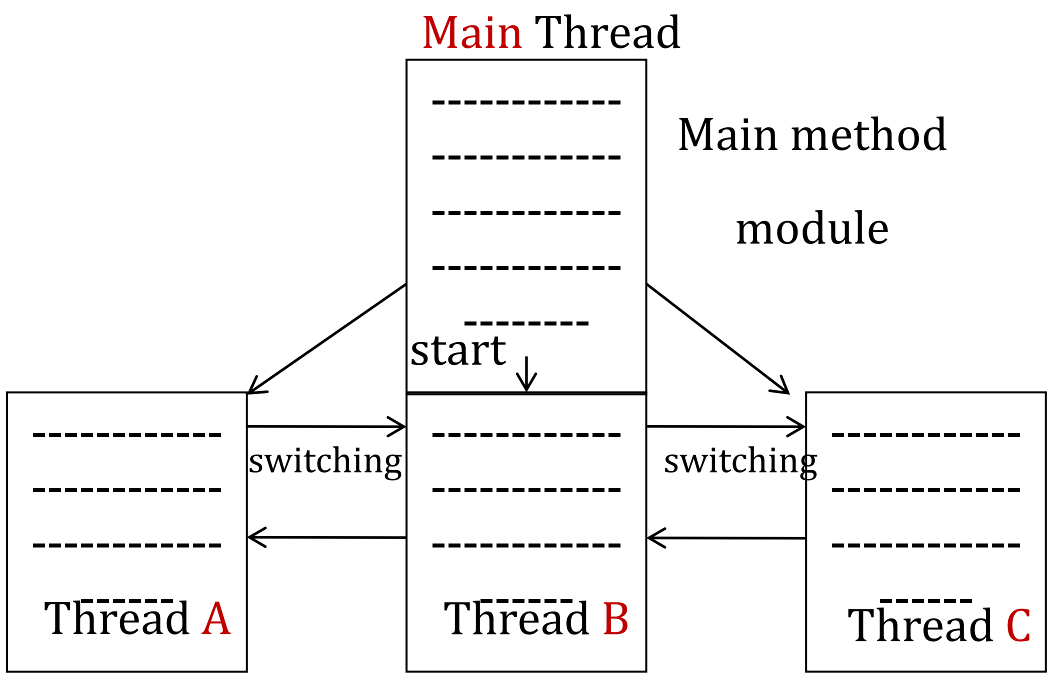
It has A beginning A body and an end Executes command sequentially



Question: Define multithread programming? CSE-2014

Multithread programming

- 🕮 A program that contains multiple flows of control is known a multiplethreaded program



Question: Discuss Different ways of creating Thread in Java?

Creating Threads:
Threads are implemented in the form of objects that contain a method called run()
The run method is the heart and soul of any thread

There have two way to create a Thread:

1. Implementing the runnable interface
2. Extending the thread class

Creating Threads

1. **Runnable**
   This is the barebones thread implementation
   There is only a single method named run() that need to be implemented

Example of Implementing the Runnable Interface

```
class NewThread implements Runnable{
Thread t;

NewThread() {
t = new Thread (this,"Demo Thread");
System.out.println ("Child thread:" +t);
t.start () ;//start the thread
}

public void run(){
try{
for (int i=5; i>0; i--){
System.out.println("child thread:"+i);
Thread.sleep(500);
}
}catch (InterruptedException e){
System.out.println("child interrupted.");
}

System.out.println("exiting child thread.");
}
}
```

2. **Extending the thread class**
   - 🕮 It can make the class runnable as a thread by extending the class java.lang.Thread
   - 🕮 It includes the following steps
     - ❖ Declare the class as extending the tread class
     - ❖ Implement the run() method that is responsible for the sequence of code that the thread will execute
     - ❖ Create a thread object and call the start() method to initiate the tread execution

```
Class MyThread extends Thread
{
------
}
Public void run()
{
------
}
MyThread aThread = newMyThread();
aThread.start(); //invoke run() method
```

Example of Extending the Thread Class

```
class A extends Thread
{
public void run()
{
for(int i=1; i<=5; i++)
{
System.out.println ("\t from yhread A: i="+i);
}
System.out.println("exit from A");
}
}

class B extends Thread
{
public void run()
{
for(int j=1; j<=5; j++)
{
System.out.println ("\t from yhread B: j="+j);
}
System.out.println("exit from B");
}
}
class C extends Thread
{
public void run()
{
for(int k=1; k<=5; k++)
{
System.out.println ("\t from yhread C:ki="+k);
}
System.out.println("exit from C");
}
}

class wc
{
public static void main(String args[])
{
new A ().start();
new B ().start();
new C ().start();
}
}
```

Stopping and blocking a thread
Stopping a thread

- If we want to stop a thread from running further, we may do so calling its stop() method a Thread.stop()

- 🕮 This statement causes the thread to move to the dead state
- 🕮 A thread will also move to the dead state automatically when it reches the end of its method
- 🕮 The stop() method may be used when the premature death of a thread is desired

Blocking a thread

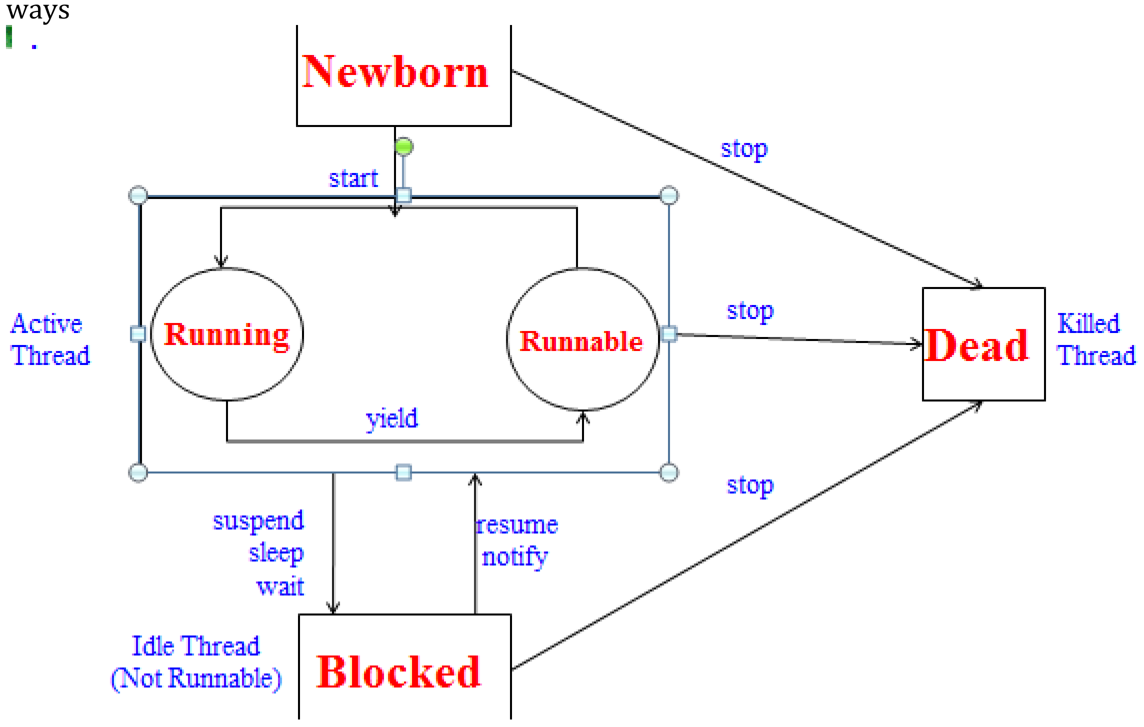
- 🕮 A thread can also be temporarily suspended or blocked from entering in to the runnable and subsequentially running state by using the following thread method
- 🕮 Sleep() //block from the specified time
- 🕮 Suspended() //block until further order specified time
- 🕮 Wait() //block until certain condition occurs

Life Cycle of a Thread

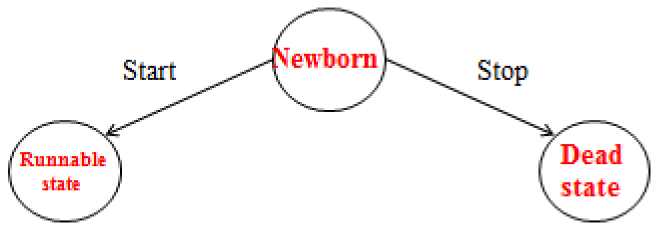
During the life time of a thread, there are many states it can enter. They include

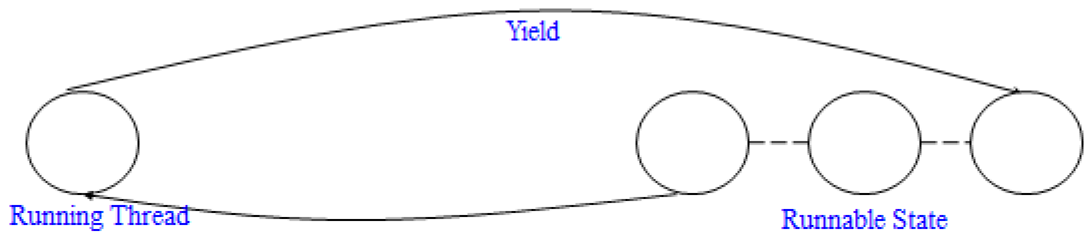
- 🕮 Newborn state
- 🕮 Runnable state
- 🕮 Running state
- 🕮 Blocked state
- 🕮 Dead state

A thread is always in one of these five states,It can move from one state to another via a variety of ways



Runnable State:

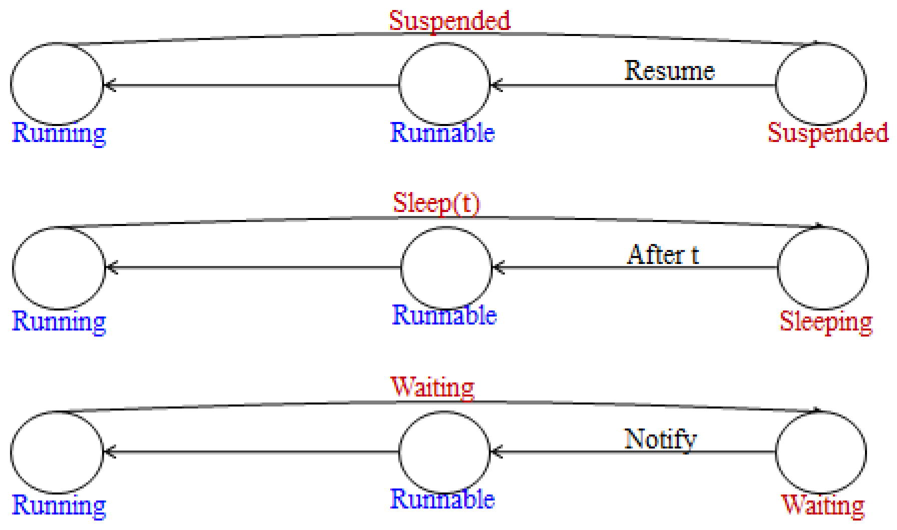




- 🕮 The runnable state means that the thread is ready for execution and is waiting for the availability of the processor
- 🕮 If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e, first come first serve manner
- 🕮 This process of assigning time to thread is known as time slicing

Running State

- 🕮 Running means that the processor has given its time to the thread for its execution
- 🕮 The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread
- 🕮 A running thread may relinquish its control in one of the following



Blocked and Dead State

Blocked state

- ❖ A thread is said to be blocked when it is prevent from entering into the runnable state and subsequently the running state

Dead state

- ❖ A running thread ends its life when it has completed executing its run() method. It is a natural death

Example

```
try{
sleep (1000);
}
catch (Exception e)
{ } }
System.out.println ("Exit from C");
} }
class lifecycle{
public static void main (String args[]){
A  threadA= new A();
B  threadB=new B ();
C  threadC= new C ();
System.out.println ("Start thread A");
threadA.start();
System.out.println ("Start thread B");
threadB.start();
System.out.println ("Start thread C");
threadC.start();
System.out.println ("End of main thread");
}
}
class A extends Thread{
public void run(){
for (int i=1; i<=5; i++){
if (i==1) yield ();
System.out.println ("\tFrom Thread A : i=" +i);
}
System.out.println ("Exit from A");
}}
class B extends Thread{
public void run (){
for (int j=1; j<=5; j++){
System.out.println ("\tFrom Thread B : j="+j);
if (j==3) ~~stop ();~~
}
System.out.println ("Exit from C");
```

```
} }
class C extends Thread{
public void run(){
for (int k=1; k<=5; k++){
System.out.println ("\tFrom Thread C:  k="+k);
if (k==1)
```

Synchronization

- 🕮 Java supports multiple threads to be executed
- 🕮 This may cause two or more threads to access the same fields or objects
- 🕮 Synchronization control the access the multiple threads to a shared resources
- 🕮 Synchronization is the process of allowing threads to execute one after another

- 🕮 When a method is declared as synchronized
  - ❖ The thread holds the monitor for that method's object if another thread is executing the synchronized method, that thread is blocked until that thread releases the monitor

This is the general form of the synchronized statement:

```
synchronized(object)
{
 // statements to be synchronized
}
```

Example

```
// File Name : Callme.java
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted"); }
    System.out.println("]");
  } }

// File Name : Caller.java
class Caller implements Runnable {
  String msg;  Callme target;  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();  }
```

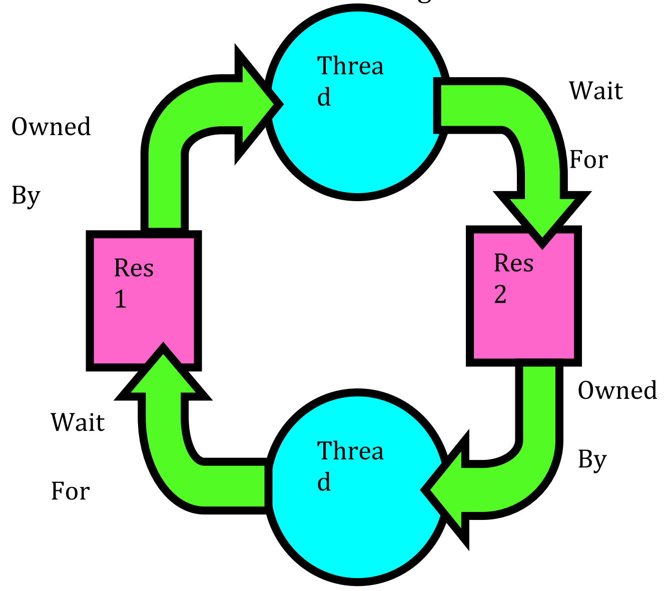
```
   // synchronize calls to call()
  public void run() {
//synchronized(target) { // synchronized block
      target.call(msg);
   // }
} }
// File Name : Synch.java
class sync {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }   }  }
```

Deadlock

Deadlock: circular waiting for resources

Thread A owns Res 1 and is waiting for Res 2

Thread B owns Res 2 and is waiting for Res 1



Consider mutexes 'x' and 'y':

Thread A Thread B

```
x.P();      y.P();
y.P();      x.P();
```

Thread scheduling

In general, the runnable thread with the highest priority is active (running)

- 🕮 Java is priority-preemptive
  - ❖ If a high-priority thread wakes up, and a low-priority thread is running
  - ❖ Then the high-priority thread gets to run immediately
- 🕮 Allows on-demand processing

```
public class MultithreadedApp extends Thread{
static int a=10;
int b =5;
int mult;
MultithreadedApp(intm){
mult=m;}
a=a*mult;
b=b*mult;
public static void main(String args[]){
MultithreadedApps testObj1=newMultithreadedApp(10);
MultithreadedApps testObj2=new MultithreadedApp(5);
testObj1.start();
try{testObj1.join();
}
catch(Exception exp){System.out.println("Exception");}
System.out.println("Final value of a is:"+MultithreadedApp.a);
System.out.println("Final value of b is:"+testObj1.b);
System.out.println("Final value of a is:"+testObj2.b);
}
}
```

i)Write down the output of the program.

ii)Make necessary addition in the main method so that the out becomes

final value of a is:500
final value of b is:50
final value of b is:25

1. What o you understand by a multi-threaded program? Explain the two different ways of implementing multiple threads in a Java program. Exam-2013
2. Describe the life cycle of a thread. Exam-2013
3. Consider the following Java program. Exam-2013

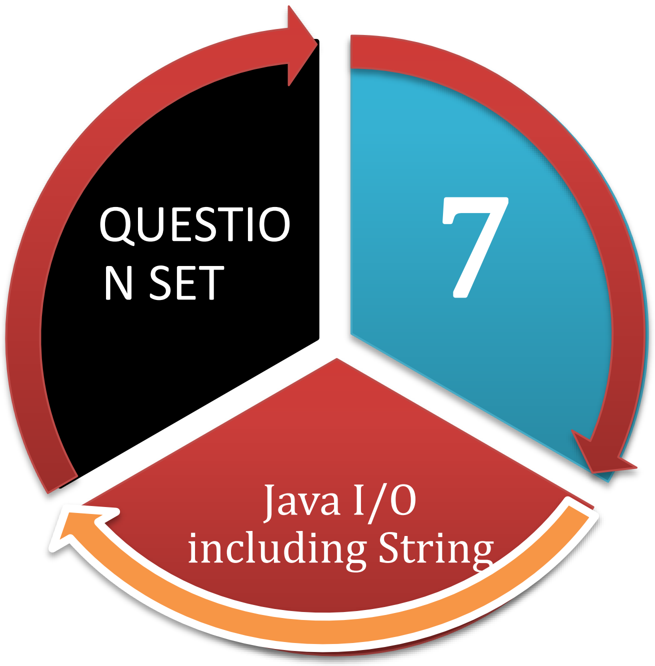
```
public class ThreadTest extends Thread
{
static int a=10;
int mult;
public ThreadTest(int m)
{
mult=m;
}
public void  run()
{a=a*mult;
System.out.println("New value of a is:"+a);
}
public static void main(String args[])
{ThreadTest testObj1=new ThreadTest(10);
ThreadTest testObj2=new ThreadTest(5);
testObj1.start();
try{testObj1.join();
}
catch(Exception exp)
{System.out.println("exception occured:"+exp);
}
System.out.println("Final value of a is:"+ThreadTst.a);
}
}
```

i)Write down the output of the program?

ii)Make necessary addition within the try block of the main method so that the output becomes as follows:

```
New value of a is:100
New value of a is:500
Final value of a is:500
```

QUESTION SET 6: Lambda Expression



- **Java's File Management Techniques**

File management

- File management is the manner in which files are monitored and controlled for standard I/O access
- The File class provides
  - A constructor to create a file handle
  - This file handle is then used by various file class methods to access the properties of a specific file or by file stream constructors to open files
- The File class also provides
  - An appropriate platform dependent directory and
  - File separator symbol using either File.separator or File.separatorChar

- Java's *File* Class
- Java's *RandomAccessFile* Class

Java's *File* Class

- The java.io package includes a class known as the File class that provides support for creating files and directories
- The class includes different several constructors for instantiating the File objects

- This class also contains several methods for supporting the operations such as
  - Creating a file
  - Opening a file
  - Closing a file
  - Deleting a file
  - Getting the name of the file
  - Getting the size of the file
  - Renaming a file. etc….

Creating a file

- If we want to create and use a disk file, we need to decide the following about the file and its intended purpose
  - Suitable name for the file
  - Data type to be stored
  - Purpose (reading, writing, or updating)
  - Method of creating the file
- A filename is a unique string of characters that helps identity a file on the disk
- A filename may contain two parts
  - A primary name
  - An optional period with extension
    Ex: input.dat, student.txt

Java's RandomAccessFile Class

- File class can use either for "read only" or for "write only" operations and not for both purpose simultaneously
- To read and write files, we can use one of two approaches
  - The extremely powerful stream classes
  - Class RandomAccessFile
- RandomAccessFile class supported by the java.io package allows us to create files that can be used for reading and writing data with random access
- RandomAccessFile, is easy to work with but has severe limitations
- Class *RandomAccessFil*e does I/O only on files

- A program can start reading or writing a random-access file at any place and read or write any number of bytes at a time
- *Random access files* allow files to be accessed at a specific point in the file
- They can also be opened in read/write mode which allows updating of a current file
- The constructor is *RandomAccessFile(FilefileObject, String accessMethod)* where the access method is either "r" or "rw"

Reading/Writing Characters:

- The two subclasses used for handling characters in files are FileReader and FileWriter
- The program below uses these two file classes to copy the contents of a file name "input.dat" in to a file called "output.dat"

```
import java.io.*;
class CopyCharacters {
public static void main(String args[]){
File inFile = new File ("input.dat");
File outFile = new File ("output.dat");
FileReader ins = null;
FileWriter outs = null;
try {
ins = new FileReader (inFile);
outs = new FileWriter (outFile);
int ch;
while((ch = ins.read())!= -1) {
outs.write(ch);
}
}

catch(IOException e)
{
System.out.println("hello");
System.exit(-1);
}
finally {
try {
ins.close();
outs.close();
}
catch(IOException e){
}
}
}
}
```

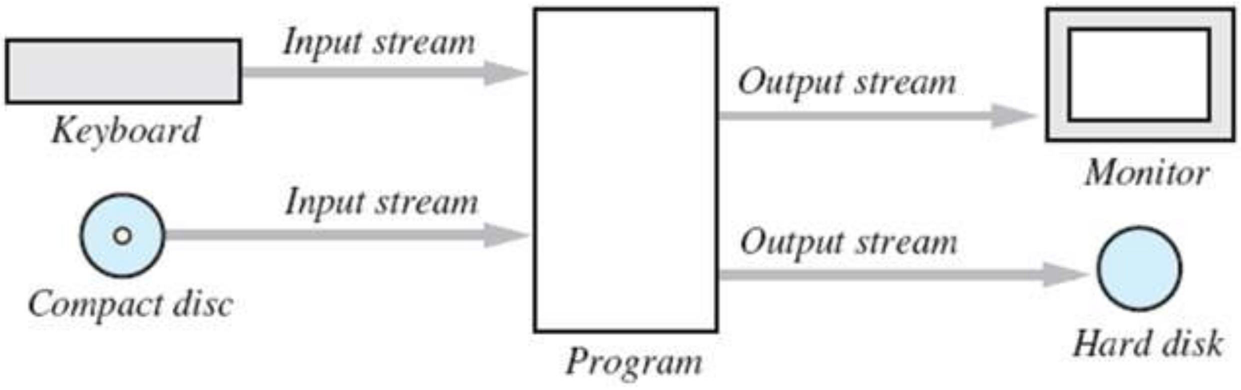
Reading/Writing using a random access file

```
import java.io.*;
class CopyCharacters {
public static void main(String args[]){
RandomAccessFile file = null;
try {
file = new RandomAccessFile ("rand.dat","rw");
file.writeChar('X');
file.writeInt(555);
file.writeDouble(3.222);
file.seek(0);
System.out.println(file.readChar());
System.out.println(file.readInt());
System.out.println(file.readDouble());
file.seek(2);
System.out.println(file.readInt());
file.seek(file.length());
file.writeBoolean(false);
file.seek(4);
System.out.println(file.readBoolean());
file.close();
}catch (IOException e) {
System.out.println(e);
}
}
}
```

- Stream manipulation classes

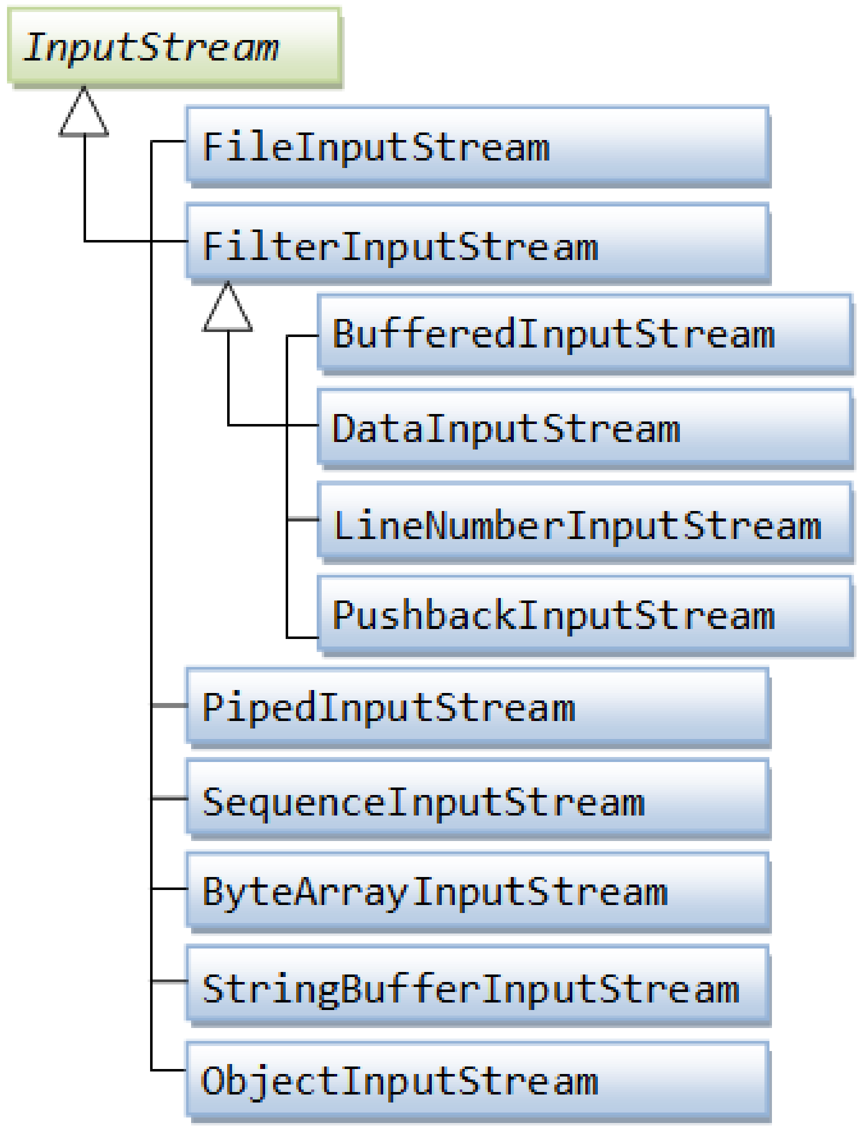
Question: Define streams?What are the standard stream classes in Java? Exam-2013
Question: Define streams and show the hierarchy of standard stream classes. Exam-2014
Answer:

- ***Stream*: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)**
  - **It acts as a buffer between the data source and destination**



Stream Classes:
- Java's stream-based I/O is built upon four abstract classes
  - **InputStreams**
  - **OutputStreams**
  - **Reader**
  - **Writer**
- InputStream and OutputStream are designed for byte streams
  - Use the byte stream classes when working with byte or other binary objects
- Reader and Writer are designed for character streams
  - Use the character stream classes when working with characters or strings
- ***Input stream***: A stream that provides input to a program
  - *System.in* is an input stream
  - All input streams can read bytes of data from some kind of data source
- ***Output stream***: A stream that accepts output from a program
  - *System.out* is an output stream

The InputStream class hierarchy



- *FileInputStream, ByteArrayInputStream, SequenceInputStream, ObjectInputStream,* and *PipedInputStream* classes data sources differentiates from the others
- All the *FilterInputStream* descendants have very different purposes

Output Streams:

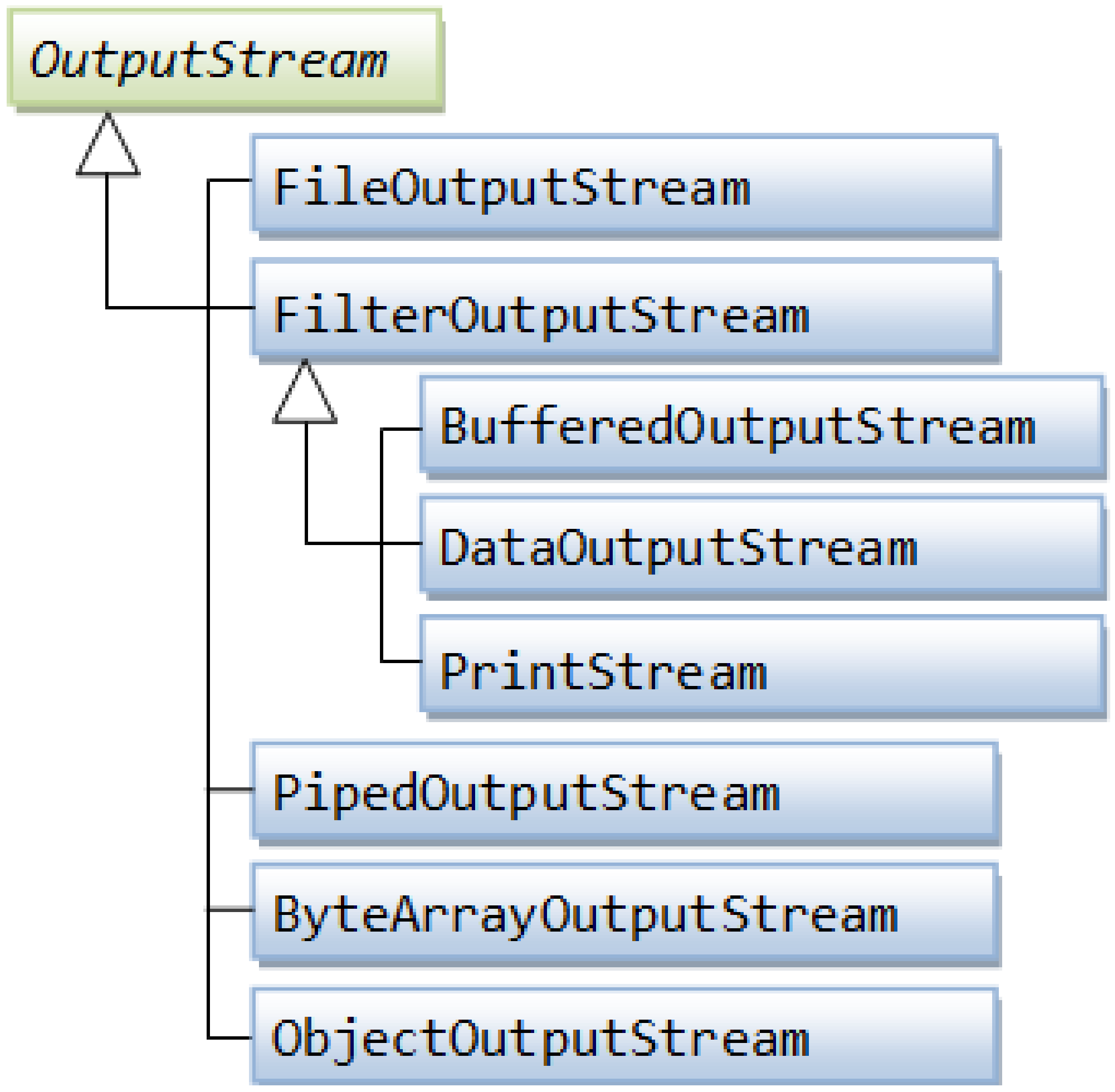
- Output streams are even simpler than input streams
- The core functionality of this class involves the capability to write bytes or characters one at a time or in blocks

OutputStream comprises three varieties of the write() method

public abstract void write(int b) throws IOException

public void write(byte b[]) throws IOException public void write(byte b[], int off, int len) throws IOException

The OutputStream class hierarchy:



FileInput/FileOutput Stream:

- The FileInputStream class creates an InputStream that use to read bytes from a file

<u>Two most common constructors</u>
  - FileInputStream(String filepath) //filepath is the full name of a file
  - FileInputStream(File fileobj) //fileobj is a File object that describes the file
- FileOutputStream creates an OutputStream that use to write bytes to a file

<u>Most common constructors</u>
  - FileOutputStream(String filepath)
  - FileOutputStream(File fileobj)
  - FileOutputStream(String filepath, boolean append)// if append is true, the file is opend in append mode

ByteArrayInput/ByteArrayOutput Stream:

```
import java.io.*;
public class ByteArrayInputStreamDemo {
public static void main(String args[]) throws IOException {
String tmp = "abcdefghijklmnoprstuvwxyz";
byte b[] = tmp.getBytes();
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArrayInputStream(b);
```

Example:

```
import java.io.*;
public class wc {
public static void main(String args[]) throws IOException {
String tmp = "abc";
byte b[] = tmp.getBytes();
ByteArrayInputStream in = new ByteArrayInputStream(b);
for (int i=0;i<2;i++){
int c;
while ((c = in.read())!=-1){
                if (i==0){
System.out.print((char)c);
}
else {
System.out.print(Character.toUpperCase((char)c));
}
}
System.out.println();
in.reset();
}
}
}
```

ObjectInput/OutputStream:

- Object input and output streams support object serialization

SequenceInputStream:

- Class *SequenceInputStream* allows to concatenate multiple InputStreams
- A *SequenceInputStream* constructor uses either a pair of *InputStream* or an Enumeration of *InputStream* as its arguments

Constructors

```
public SequenceInputStream(InputStream s1, InputStream s2)
public SequenceInputStream (Enumeration e)
```

FilterInput/OutputStream

- Filter streams are simply wrappers around underlying input or output streams that transparency provide some extended of functionality
- These streams are typically accessed by methods that are expecting a generic stream, which is a super class of the filtered stream
- Filter byte streams are FilterInputStream and FilterOutputStream

Constructor

- FilterOutputStream(OutputStream os)
- FilterInputStream(OutputStream is)

BufferedInput/OutputStream:

- Class BufferedInputStream enhances the bare-bones InputStream by adding to it a buffer of bytes, which usually improves reading performance significantly
    - BufferedInputStream(InputStream InputStream)
    - BufferedInputStream(InputStream InputStream, int bufSize) //the size of the buffer is passed in bufSize
- A BufferedOutputStream is similar to any OutputStream with the exception of an added flush() method that is used to ensure that data buffers as physically written to the actual output device
    - BufferedOutputStream(OutputStream OutputStream)
    - BufferedOutputStream(OutputStream OutputStream, int bufSize)

- **Classes PushbackInputStream** - adds ability to "push back" or "unread" one byte from stream

  Constructor
  PushbackInputStream(InputStream InputStream)
  PushbackInputStream(InputStream InputStream, int numBytes)

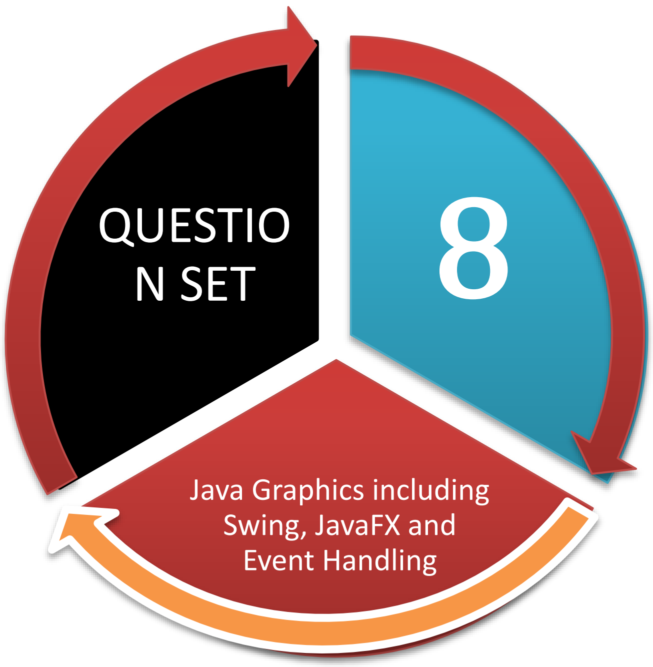
PrintStream:

- Class PrintStream resembles class DataOutputStream because the methods it defines mirror the type of write() methods provided by DataOutputStream
- The difference is that they come in two flavors:
    - print(..)
    - println(..)

There are three PrintStream constructors available
public PrintStream (OutputStream out)
public PrintStream (OutputStream out, boolean autoFlush)
public PrintStream (OutputStream out, boolean autoFlush, String encoding)

<u>**I/O Data Streams:**</u>

<u>DataInputStream - enables reading primitive Java data types from an underlying input stream</u>
<u>DataOutputStream – enables writing primitive Java data types to an output stream</u>



Question: What is a user interface?
Answer:

The user interface is that part of a program that interacts with the user of the program. User interfaces take many forms. These forms range in complexity from simple command-line interfaces to the point-and-click graphical user interfaces provided by many modern applications

There are two types of GUI elements:
Component: Components are elementary GUI entities, such as Button, Label, and TextField.
Container: Containers, such as Frame and Panel, are used to hold components in a specific layout (such as FlowLayout or GridLayout). A container can also hold sub-containers.

AWT Packages

AWT is huge! It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 1.8). Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.

The java.awt package contains the core AWT graphics classes:

- GUI Component classes, such as Button, TextField, and Label,
- GUI Container classes, such as Frame and Panel,
- Layout managers, such as FlowLayout, BorderLayout and GridLayout,
- Custom graphics classes, such as Graphics, Color and Font.

The java.awt.event package supports event handling:

- Event classes, such as ActionEvent, MouseEvent, KeyEvent and WindowEvent,
- Event Listener Interfaces, such as ActionListener, MouseListener, KeyListener and WindowListener,
- Event Listener Adapter classes, such as MouseAdapter, KeyAdapter, and WindowAdapter.
- AWT provides a platform-independent and device-independent interface to develop graphic programs that runs on all platforms, including Windows, Mac OS, and Unixes.

2.3 AWT Container Classes:

Question: Define top-level and secondary container .Write about flow layout with syntax used in Java. Exam-2014

Top-Level Containers: Frame, Dialog and Applet

Each GUI program has a *top-level container*. The commonly-used top-level containers in AWT are `Frame`, `Dialog` and `Applet`:

- 🕮 A `Frame` provides the "main window" for the GUI application, which has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area. To write a GUI program, we typically start with a subclass extending from `java.awt.Frame` to inherit the main window as follows:

Secondary Containers: Panel and ScrollPane

Secondary containers are placed inside a top-level container or another secondary container. AWT also provide these secondary containers:

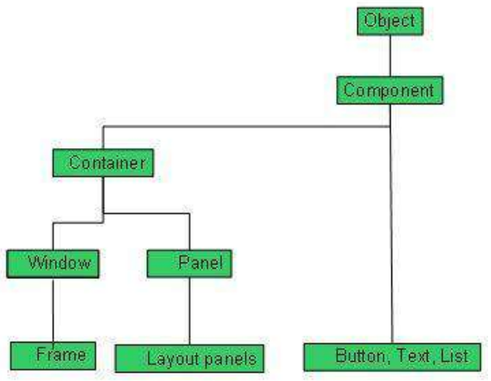
- 🕮 `Panel`: a rectangular box under a higher-level container, used to *layout* a set of related GUI components in pattern such as grid or flow.
- 🕮 `ScrollPane`: provides automatic horizontal and/or vertical scrolling for a single child component.

Basic Terminologies

Term	Description
Component	Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.
Container	The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container Examples: panel, box
Frame	A Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. Frame encapsulates window. It and has a title bar, menu bar, borders, and resizing corners.
Panel	Panel provides space in which an application can attach any other components, including other panels.
Window	Window is a rectangular area which is displayed on the screen. In different window we can execute different program and display different data. Window provide us with multitasking environment. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

Useful Methods of Component class

Every user interface considers the following three main aspects:
UI elements: These are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.
Layouts: They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in Layout chapter.
Behavior: These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.



What are the differences between 'lightweight' and 'heavyweight' GUI components in java?
Ans:
There are many differences between java awt and swing that are given below.

Java AWT Java Swing
1) AWT components are platform-dependent.
Java swing components
are platform-independent.
2) AWT components are heavyweight. Swing components
are lightweight.
3) AWT doesn't support pluggable look and
feel.
Swing supports pluggable look
and feel.
4) AWT provides less components than
Swing.
Swing provides more powerful
components such as tables, lists,
scrollpanes, colorchooser,
tabbedpane etc.
5) AWT doesn't follows MVC(Model View
Controller) where model represents data,
view represents presentation and controller
acts as an interface between model and
view.
Swing follows MVC

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	**Label:** A Label object is a component for placing text in a container.

2	**Button:** This class creates a labeled button.
3	**Check Box:** A check box is a graphical component that can be in either an on (true) or off (false) state.
4	**Check Box Group:** The CheckboxGroup class is used to group the set of checkbox.
5	List: The List component presents the user with a scrolling list of text items.
6	**Text Field:** A TextField object is a text component that allows for the editing of a single line of text.
7	**Text Area:** A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	**Choice:** A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
9	**Canvas:** A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
10	**Image:** An Image control is superclass for all image classes representing graphical images.
11	**Scroll Bar:** A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
12	**Dialog:** A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
13	**File Dialog:** A FileDialog control represents a dialog window from which the user can select a file.

Question: What is the difference between container and component in a GUI? Exam-2015
Answer:
Difference between container and component:

Container	**Component**
A container is a component that holds and manages other components. Containers display components using a layout manager.	a component is the basic user interface object and is found in all Java applications. Components include lists, buttons, panels, and windows.
To use components, we need to place them in a container.	Component important for interaction
JComponent, in turn, inherits from the Container class in the Abstract Windowing Toolkit (AWT). So Swing is based on classes inherited from AWT.	Swing components inherit from the javax.Swing.JComponent class, which is the root of the Swing component hierarchy.
Containers: Frame Window Dialog Panel	List of common Components List Scrollbar TextArea TextField Choice Button Label

Creating graphical user interfaces with AWT,

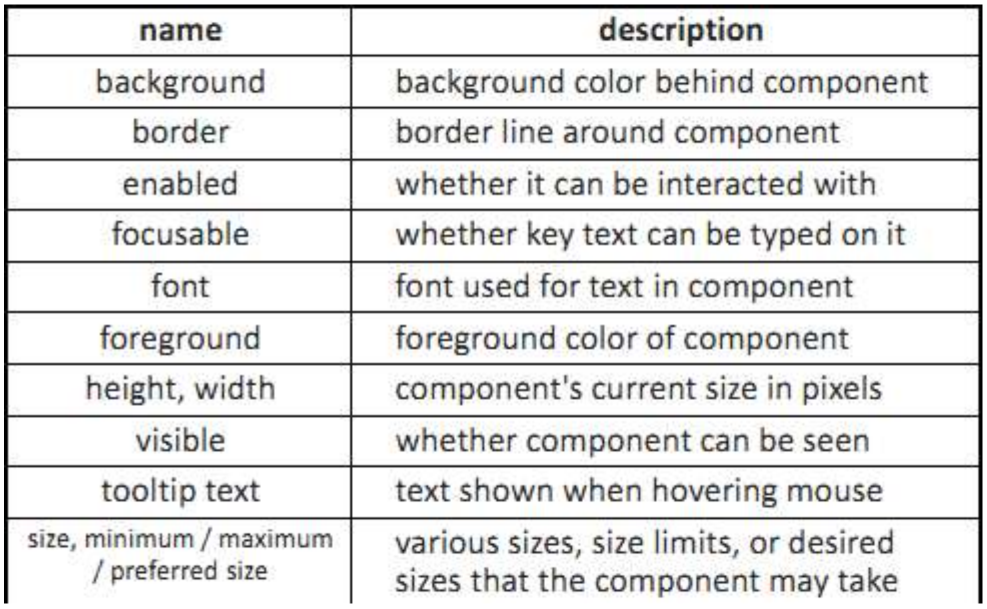
To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

1. By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

Frame Typically holds (hosts) other components Common methods:

- JFrame(String title) – constructor, title optional
- setSize(int width, int height) – set size
- add(Component c) – add component to window
- setVisible(boolean v) – make window visible or not. Don't forget this public void
- setDefaultCloseOperation(int op) Makes the frame perform the given action when it closes.
  - Common value passed: JFrame.EXIT_ON_CLOSE
  - If not set, the program will never exit even if the frame is closed.
- public void setSize(int width, int height) Gives the frame a fixed size in pixels.
- public void pack() Resizes the frame to fit the components inside it snugly.

Some Attribute of Frame:



name	description
background	background color behind component
border	border line around component
enabled	whether it can be interacted with
focusable	whether key text can be typed on it
font	font used for text in component
foreground	foreground color of component
height, width	component's current size in pixels
visible	whether component can be seen
tooltip text	text shown when hovering mouse
size, minimum / maximum / preferred size	various sizes, size limits, or desired sizes that the component may take

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
Button b=new Button("click me");
b.setBounds(30,50,80,30);
f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}}
```

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Managing graphics objects with GUI layout managers,

Question: What is layout manager? Briefly explain about the basic layout manager in Java and give the syntax of each layout manager. Exam-2013
Question: Write down the constructors of Flow Layout, Card Layout and GridBag Laoyout manager. Explain them. Exam-2015
Layout Manager:

A layout manager is an object that implements the LayoutManager interface* and determines the size and position of the components within a container.

Although components can provide size and alignment hints, a container's layout manager has the final say on the size and position of the components within the container.

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

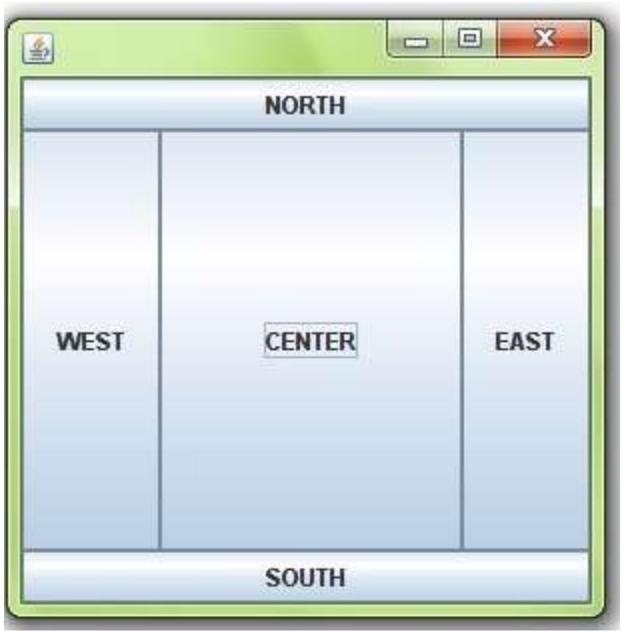
There are following classes that represents the layout managers:

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.GridLayout
- java.awt.CardLayout
- java.awt.GridBagLayout
- javax.swing.BoxLayout
- javax.swing.GroupLayout
- javax.swing.ScrollPaneLayout
- javax.swing.SpringLayout etc.

We Discuss here some AWT of layout:

- **Java BorderLayout**

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region:

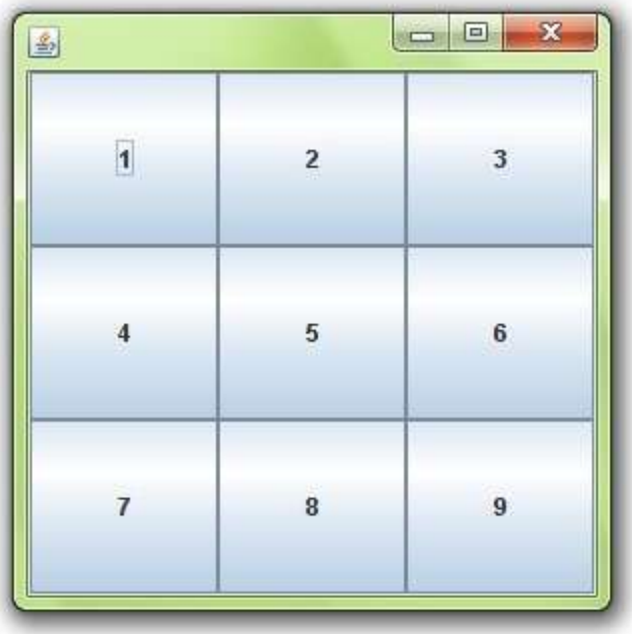


1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

```
import java.awt.*;

import javax.swing.*;

public class Border {
JFrame f;
Border(){
f=new JFrame();

JButton b1=new JButton("NORTH");;
JButton b2=new JButton("SOUTH");;
JButton b3=new JButton("EAST");;
JButton b4=new JButton("WEST");;
JButton b5=new JButton("CENTER");;

f.add(b1,BorderLayout.NORTH);
f.add(b2,BorderLayout.SOUTH);
f.add(b3,BorderLayout.EAST);
f.add(b4,BorderLayout.WEST);
f.add(b5,BorderLayout.CENTER);

f.setSize(300,300);
f.setVisible(true);
}
public static void main(String[] args) {
new Border();
}    }
```

🕮 Java GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.
Constructors of GridLayout class



1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class

```
import java.awt.*;
import javax.swing.*;

public class MyGridLayout{
JFrame f;
MyGridLayout(){
  f=new JFrame();

  JButton b1=new JButton("1");
  JButton b2=new JButton("2");
  JButton b3=new JButton("3");
  JButton b4=new JButton("4");
  JButton b5=new JButton("5");
    JButton b6=new JButton("6");
    JButton b7=new JButton("7");
  JButton b8=new JButton("8");
    JButton b9=new JButton("9");

  f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
  f.add(b6);f.add(b7);f.add(b8);f.add(b9);

  f.setLayout(new GridLayout(3,3));
  //setting grid layout of 3 rows and 3 columns

  f.setSize(300,300);
  f.setVisible(true);
}
public static void main(String[] args) {
  new MyGridLayout();
}
}
```

Java FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.
Fields of FlowLayout class

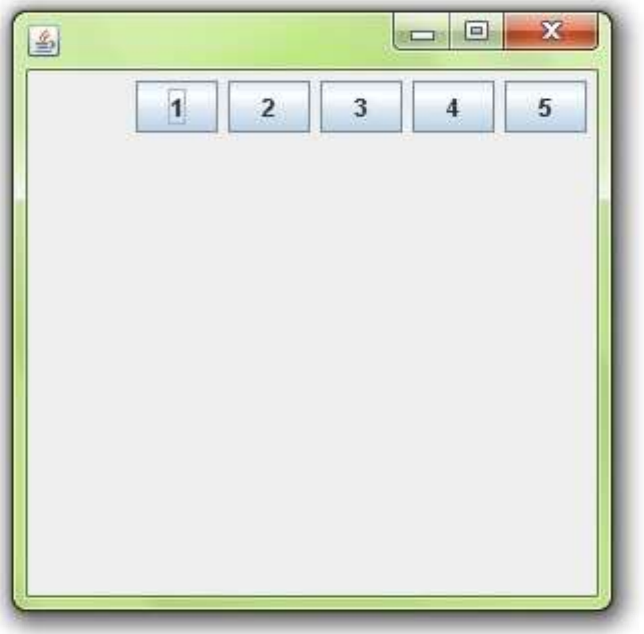
1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class

```
import java.awt.*;
import javax.swing.*;

public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
   f=new JFrame();

   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");

   f.add(b1);f.add(b2);f.add(b3);f.add(b
4);f.add(b5);

   f.setLayout(new FlowLayout(FlowLa
yout.RIGHT));
   //setting flow layout of right alignme
nt

   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyFlowLayout();
```



```
}
}
```

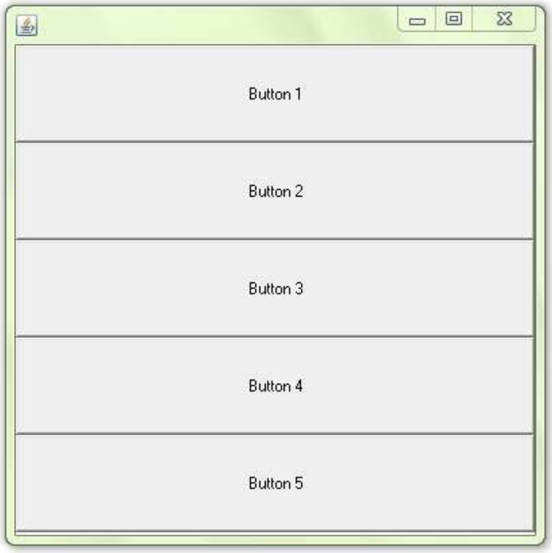
Java BoxLayout

The BoxLayout is used to arrange the components either vertically or horizontally. For this purpose, BoxLayout provides four constants. They are as follows:
Note: BoxLayout class is found in javax.swing package.
Fields of BoxLayout class

1. **public static final int X_AXIS**
2. **public static final int Y_AXIS**
3. **public static final int LINE_AXIS**
4. **public static final int PAGE_AXIS**

Example of BoxLayout class with Y-

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample1 extends Frame {
 Button buttons[];

 public BoxLayoutExample1 () {
   buttons = new Button [5];

   for (int i = 0;i<5;i++) {
     buttons[i] = new Button ("Button " + (i + 1));
     add (buttons[i]);
    }

setLayout (new BoxLayout (this, BoxLayout.Y_AXIS)
);
setSize(400,400);
setVisible(true);
}

public static void main(String args[]){
BoxLayoutExample1 b=new BoxLayoutExample1();

}
}
```



Java CardLayout

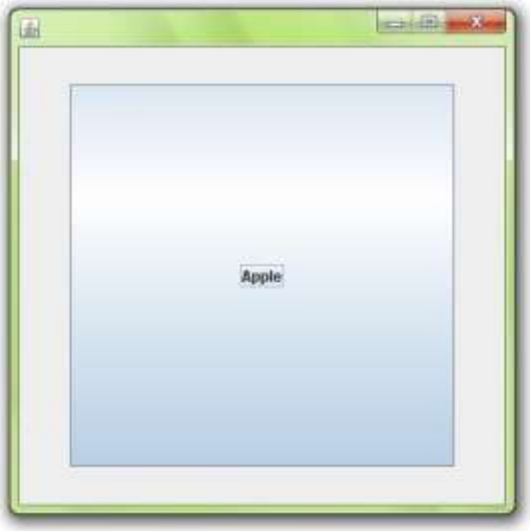
The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout. Constructors of CardLayout class

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

Commonly used methods of CardLayout class

- **public void next(Container parent):** is used to flip to the next card of the given container.
- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

Example of CardLayout class



```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class CardLayoutExample extends JFra
me implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
  CardLayoutExample(){

    c=getContentPane();
    card=new CardLayout(40,30);
//create CardLayout object with 40 hor space
 and 30 ver space
    c.setLayout(card);

    b1=new JButton("Apple");
    b2=new JButton("Boy");
    b3=new JButton("Cat");
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);

    c.add("a",b1);c.add("b",b2);c.add("c",b3);

  }
  public void actionPerformed(ActionEvent e) {
  card.next(c);
  }

  public static void main(String[] args) {
    CardLayoutExample cl=new CardLayoutExample();
    cl.setSize(400,400);
    cl.setVisible(true);
    cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
```

- **Java GridBagLayout**

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.

Summary of Layout Manager:

Sr. No.	Layout Manager & Description
1	**BorderLayout** The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.
3	**FlowLayout** The FlowLayout is the default layout.It layouts the components in a directional flow.
4	**GridLayout** The GridLayout manages the components in form of a rectangular grid.
5	**GridBagLayout** This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.

Drawing methods include:

- drawString – For drawing text
- g.drawString("Hello", 10, 10);
- drawImage – For drawing images
- g.drawImage(img,
- 0, 0, width, height,
  0, 0, imageWidth, imageHeight,
  null);

drawLine, drawArc, drawRect, drawOval, drawPolygon – For drawing geometric shapes

g2.draw(new Line2D.Double(0, 0, 30, 40));

Depending on your current need, you can choose one of several methods in the `Graphics` class based on the following criteria:

- Whether you want to render the image at the specified location in its original size or scale it to fit inside the given rectangle
- Whether you prefer to fill the transparent areas of the image with color or keep them transparent

Fill methods apply to geometric shapes and include `fillArc`, `fillRect`, `fillOval`, `fillPolygon`.

Whether you draw a line of text or an image, remember that in 2D graphics every point is determined by its x and y coordinates. All of the draw and fill methods need this information which determines where the text or image should be rendered.

For example, to draw a line, an application calls the following:

```
java.awt.Graphics.drawLine(int x1, int y1, int x2, int y2)
```

In this code *(x1, y1)* is the start point of the line, and *(x2, y2)* is the end point of the line.

So the code to draw a horizontal line is as follows:

```
Graphics.drawLine(20, 100, 120, 100);
```

For convenience's sake in this lecture the variable `g` will always refer to a preexisting object of the `Graphics` class. As with any other method you are free to use some other name for the particular `Graphics` context, `myGraphics` or `appletGraphics` perhaps.

Layout strategy for border interactors

As an example, suppose that you want to write a program that displays two buttons—**Start** and **Stop**—at the bottom of a program window. Let's ignore for the moment what those buttons actually do and concentrate instead on how to make them appear. If you use the standard layout management tools provided by the **Program** class, all you have to do is include the following code as part of the **init** method:

```
add(new JButton("Start"), SOUTH);
add(new JButton("Stop"), SOUTH);
```

Question: How to implement a listener interface. Exam-2013
Answer:
Assigning action listeners to the buttons

Creating the buttons, however, accomplishes only part of the task. To make the buttons active, you need to give each one an action listener so that pressing the button performs the appropriate action.

These days, the most common programming style among experienced Java programmers is to assign an individual action listener to each button in the form of an anonymous inner class. Suppose, for example, that you want the Start and Stop buttons to invoke methods called startAction and stopAction, respectively. You could do so by changing the initialization code as follows:

```
JButton startButton = new JButton("Start");
startButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    startAction();
  }
});
add(startButton, SOUTH);
JButton stopButton = new JButton("Start");
stopButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    stopAction();
  }
});
add(stopButton, SOUTH);
```

Question:Define actionPerfomed().What is the signature of the actionPerfomed method? Exam-2014
Answer:
Define actionPerfomed().

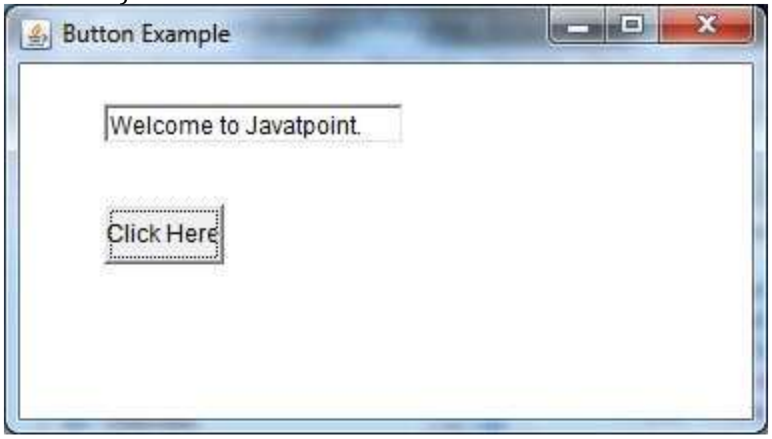
The actionPerformed() method is invoked automatically whenever you click on the registered component.

public abstract void actionPerformed(ActionEvent e);

The ActionEvent argument e is the object that represents the event. This object contains information about the event, like which component is the event source. The ActionEvent class has two methods that you might find beneficial: getActionCommand() and getSource() (which is actually inherited by the ActionEvent class from the EventObject class). The first method returns a string that identifies the action. This string is usually set by the event source, using a method called setActionCommand. The second method returns the event source object

Example:

```
import java.awt.*;
import java.awt.event.*;
public class ActionListenerExample {
public static void main(String[] args) {
   Frame f=new Frame("ActionListener Example");
   final TextField tf=new TextField();
   tf.setBounds(50,50, 150,20);
   Button b=new Button("Click Here");
   b.setBounds(50,100,60,30);

   b.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
         tf.setText("Welcome to Javatpoint.");
   }
   });
   f.add(b);f.add(tf);
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}
}
```



Signature of the actionPerfomed method:

The signature of actionPerfomed method is ActionEvent e

<u>Event handling of various components.</u>

Question: What is an event? List down the names of some of the Java AWT event listener interfaces.
Question: What is an Event?

Change button 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.

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.

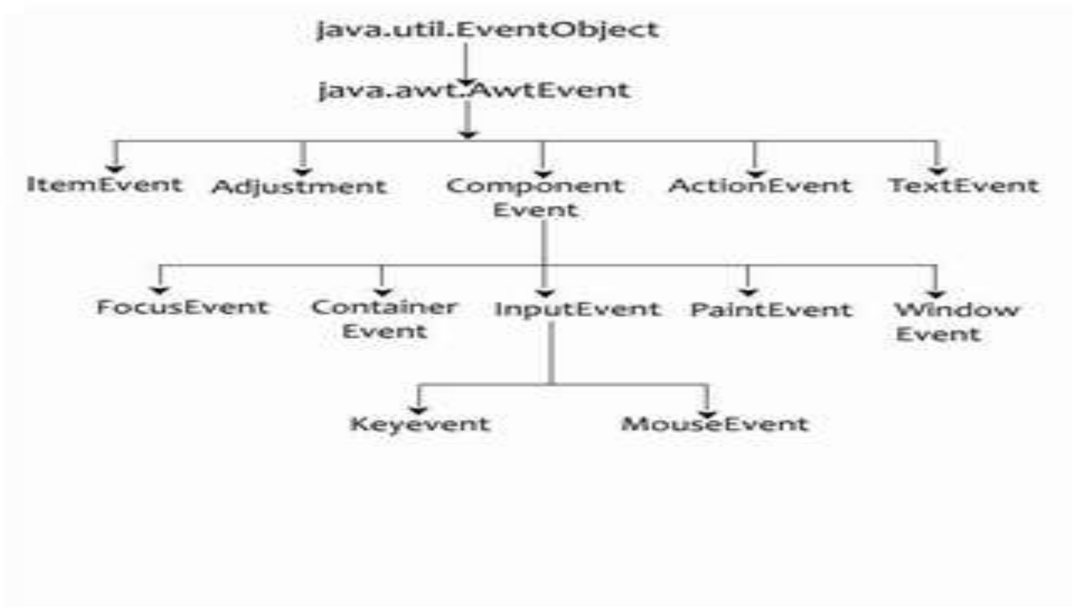
List Java AWT event listener interface:

Interface Summary	
Interface	**Description**
ActionListener	The listener interface for receiving action events.
AdjustmentListener	The listener interface for receiving adjustment events.
AWTEventListener	The listener interface for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or their subclasses.
ComponentListener	The listener interface for receiving component events.
ContainerListener	The listener interface for receiving container events.
FocusListener	The listener interface for receiving keyboard focus events on a component.
HierarchyBoundsListener	The listener interface for receiving ancestor moved and resized events.
HierarchyListener	The listener interface for receiving hierarchy changed events.
InputMethodListener	The listener interface for receiving input method events.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
MouseWheelListener	The listener interface for receiving mouse wheel events on a component.

TextListener	The listener interface for receiving text events.
WindowFocusListener	The listener interface for receiving `WindowEvents`, including `WINDOW_GAINED_FOCUS` and `WINDOW_LOST_FOCUS` events.
WindowListener	The listener interface for receiving window events.
WindowStateListener	The listener interface for receiving window state events.

List Java AWT event listener classes:

Class Summary	
Class	**Description**
ActionEvent	A semantic event which indicates that a component-defined action occurred.
ComponentEvent	A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).
ContainerAdapter	An abstract adapter class for receiving container events.
ContainerEvent	A low-level event which indicates that a container's contents changed because a component was added or removed.
FocusAdapter	An abstract adapter class for receiving keyboard focus events.
FocusEvent	A low-level event which indicates that a Component has gained or lost the input focus.
MouseEvent	An event which indicates that a mouse action occurred in a component.
MouseMotionAdapter	An abstract adapter class for receiving mouse motion events.
MouseWheelEvent	An event which indicates that the mouse wheel was rotated in a component.
PaintEvent	The component-level paint event.
TextEvent	A semantic event which indicates that an object's text changed.
WindowAdapter	An abstract adapter class for receiving window events.
WindowEvent	A low-level event that indicates that a window has changed its status.



At the root of Java event class hierarchy is EventObject which is in java.util. It is a super class for all events.

Question: What is Event Handling?
Answer:

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs.
The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provide as with classes for source object.

Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Steps involved in event handling

The User clicks the button and the event is generated.
Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

Event object is forwarded to the method of registered listener class.
the method is now get executed and returns.

Event Handling Example

Create the following java program using any editor of your choice in say **D:/ > AWT > com > tutorialspoint > gui >**

AwtControlDemo.java
package com.tutorialspoint.gui;

```
import java.awt.*;
import java.awt.event.*;

public class AwtControlDemo {

  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;

  public AwtControlDemo(){
    prepareGUI();
  }
  public static void main(String[] args){
    AwtControlDemo  awtControlDemo = new AwtControlDemo();
    awtControlDemo.showEventDemo();
  }
  private void prepareGUI(){
    mainFrame = new Frame("Java AWT Examples");
    mainFrame.setSize(400,400);
    mainFrame.setLayout(new GridLayout(3, 1));
    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
      }
    });
    headerLabel = new Label();
    headerLabel.setAlignment(Label.CENTER);
    statusLabel = new Label();
    statusLabel.setAlignment(Label.CENTER);
    statusLabel.setSize(350,100);
    controlPanel = new Panel();
    controlPanel.setLayout(new FlowLayout());
    mainFrame.add(headerLabel);
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }

  private void showEventDemo(){
    headerLabel.setText("Control in action: Button");
    Button okButton = new Button("OK");
    Button submitButton = new Button("Submit");
    Button cancelButton = new Button("Cancel");

    okButton.setActionCommand("OK");
    submitButton.setActionCommand("Submit");
```

```
      cancelButton.setActionCommand("Cancel");

      okButton.addActionListener(new ButtonClickListener());
      submitButton.addActionListener(new ButtonClickListener());
      cancelButton.addActionListener(new ButtonClickListener());
      controlPanel.add(okButton);
      controlPanel.add(submitButton);
      controlPanel.add(cancelButton);
      mainFrame.setVisible(true);
    }

    private class ButtonClickListener implements ActionListener{
      public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if( command.equals( "OK" ))  {
          statusLabel.setText("Ok Button clicked.");
        }
        else if( command.equals( "Submit" ) )  {
          statusLabel.setText("Submit Button clicked.");
        }
        else  {
          statusLabel.setText("Cancel Button clicked.");
        }
      }
    }
  }
```

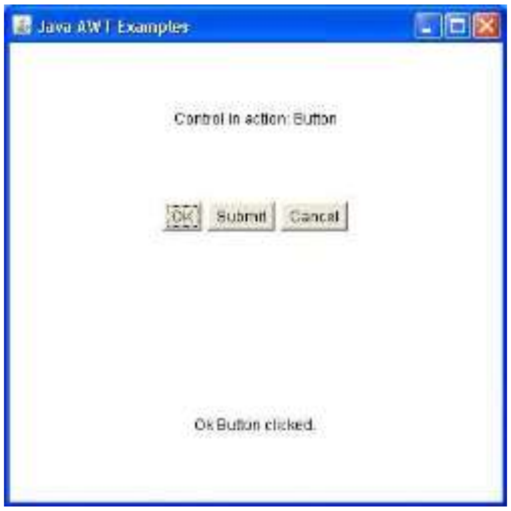
Compile the program using command prompt. Go to **D:/ > AWT** and type the following command.

```
D:\AWT>javac com\tutorialspoint\gui\AwtControlDemo.java
```

If no error comes that means compilation is successful. Run the program using following command.

```
D:\AWT>java com.tutorialspoint.gui.AwtControlDemo
```

Verify the following output



```
Control in action: Button
Ok Button clicked.
```

Question: Describe how to handle MouseEvent and KeyEvent in Java? Exam-2015
Answer:
Handling Mouse Events

Mouse events are one of the two most frequent kinds of events handled by an application (the other kind being, of course, key events). Mouse clicks—which involve a user pressing and then releasing a mouse button—generally indicate selection,

Overview of Mouse Events

Before going into the "how to" of mouse-event handling,

Table 4-1 Type constants and methods related to left-button mouse events

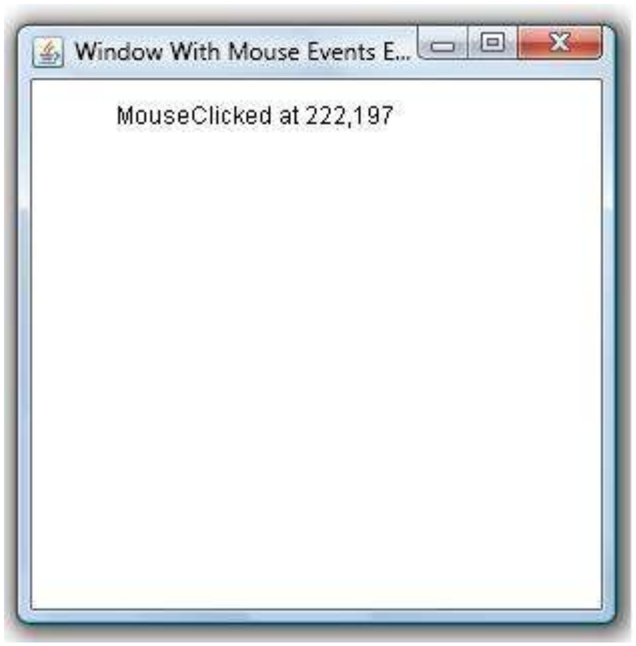
Action	Event type (left mouse button)	Mouse-event method invoked (left mouse button)
Press down the button	`NSLeftMouseDown`	`mouseDown:`
Move the mouse while pressing the button	`NSLeftMouseDragged`	`mouseDragged:`
Release the button	`NSLeftMouseUp`	`mouseUp:`
Move the mouse without pressing any button	`NSMouseMoved`	`mouseMoved:`

Mouse Event Handling in a Frame Window Example

```
/*
    Mouse Event Handling in a Frame Window Example
    This java example shows how to handle mouse events in a Frame window
    using MouseListener.
*/

import java.awt.Frame;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

/*
* To create a stand alone window, class should be extended from
* Frame and not from Applet class.
*/
```

```
public class HandleMouseListenerInWindowExample extends Frame implements MouseListener{

    int x=0, y=0;
    String strEvent = "";

    HandleMouseListenerInWindowExample(String title){

        //call superclass constructor with window title
        super(title);

        //add window listener
        addWindowListener(new MyWindowAdapter(this));

        //add mouse listener
        addMouseListener(this);

        //set window size
        setSize(300,300);

        //show the window
        setVisible(true);
    }



            public void mouseClicked(MouseEvent e) {

                strEvent = "MouseClicked";
                x = e.getX();
                y = getY();
                repaint();
            }


            public void mousePressed(MouseEvent e) {
                strEvent = "MousePressed";
                x = e.getX();
                y = getY();
                repaint();

            }


            public void mouseReleased(MouseEvent e) {
                strEvent = "MouseReleased";
                x = e.getX();
                y = getY();
                repaint();

            }
    public void mouseEntered(MouseEvent e) {
        strEvent = "MouseEntered";
        x = e.getX();
        y = getY();
        repaint();

    }


    public void mouseExited(MouseEvent e) {
        strEvent = "MouseExited";
        x = e.getX();
        y = getY();
        repaint();

    }


    public void paint(Graphics g){
        g.drawString(strEvent + " at " + x + "," + y, 50,50);
    }

    public static void main(String[] args) {

        HandleMouseListenerInWindowExample myWindow =
                new HandleMouseListenerInWindowExample("Window With Mouse Events 
Example");
    }


}

class MyWindowAdapter extends WindowAdapter{

    HandleMouseListenerInWindowExample myWindow = null;

    MyWindowAdapter(HandleMouseListenerInWindowExample myWindow){
        this.myWindow = myWindow;
    }

    public void windowClosing(WindowEvent we){
        myWindow.setVisible(false);
    }
}
```

Example Output



MouseClicked at 222,197

KeyEvent:

Key events indicate when the user is typing at the keyboard. Specifically, key events are fired by the component with the keyboard focus when the user presses or releases keyboard keys.

Notifications are sent about two basic kinds of key events:

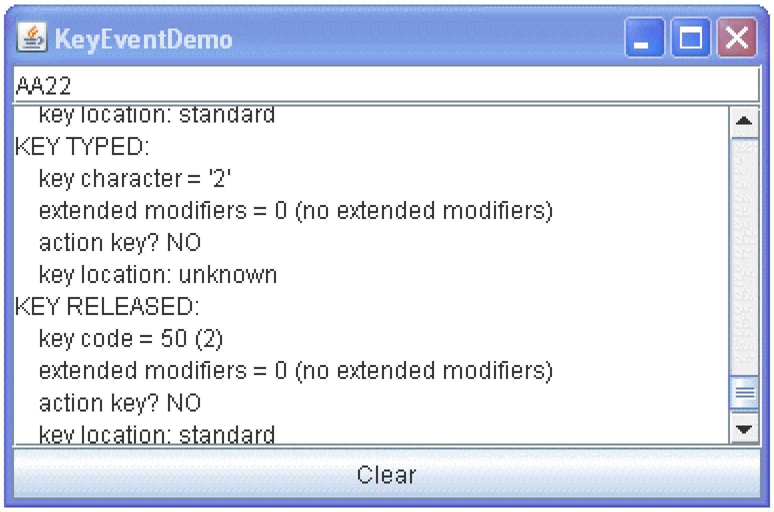
- The typing of a Unicode character
- The pressing or releasing of a key on the keyboard

The first kind of event is called a *key-typed* event. The second kind is either a *key-pressed* or *key-released* event.

To make a component get the keyboard focus, follow these steps:

1. Make sure the component's `isFocusable` method returns `true`. This state allows the component to receive the focus. For example, you can enable keyboard focus for a `JLabel` component by calling the `setFocusable(true)` method on the label.
2. Make sure the component requests the focus when appropriate. For custom components, implement a mouse listener that calls the `requestFocusInWindow` method when the component is clicked.

The following example demonstrates key events. It consists of a text field that you can type into, followed by a text area that displays a message every time the text field fires a key event. A button at the bottom of the window lets you clear both the text field and text area.



```
AA22
   key location: standard
KEY TYPED:
   key character = '2'
   extended modifiers = 0 (no extended modifiers)
   action key? NO
   key location: unknown
KEY RELEASED:
   key code = 50 (2)
   extended modifiers = 0 (no extended modifiers)
   action key? NO
   key location: standard
```