Index

:History:

Java became life as programming language in the name of OAK. Oak was developed by the members of green project in year june-1991 which mainly initiated by **James Gosling**, **Mike Sheridan**, and **Patrick Naughton**. They create it for smart electronic market but as internet came into picture, the team wanted to have a new way of computing based on power of network and wanted to run same software on different types of computer, consumer gadgets, etc. hence java was developed. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies. Java is an island of Indonesia where first coffee was produced (called java coffee).Notice that Java is just a name not an acronym. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)

3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

:Properties of java:

Sun micro system has define java as simple, object oriented, distributed, interpreted, robust, secure, architectural-neutral, portable, high performance, multi threaded & dynamic language.

1. **Simple:** java was designed to be easy for those programmers to learn and use who have learned concept of object oriented programming. For hem java is easy to use and understand. In java there are numbers of clearly define ways to complete a given task.
2. **Object oriented:** java defines data as objects with method that supports object. Java is purely object oriented & provides features like abstraction, encapsulation, inheritance, polymorphism, etc. even the most basic program has class. Any code that you write in java is inside a class. The object model in java is simple and easy to extend.
3. **Distributed:** java is turned into web; java programmer can access data across the web as easily as they access the data from local system. Java handles TCP/IP protocols. In fact accessing resources using URL is not much different from accessing a file. Java includes features of intra-address, Space-messaging, etc. This allows object on 2 computers to execute the method remotely. Java revive this interface in package called RMI(Remote Method Invocation). This feature bring client server programming level.
4. **Interpreted:** The java code is compiled to byte code that is binary and it is platform independent. When the program has to be executed, the code is fetched into the memory and interpreted in user's machine as an interpreted language. Java has simple syntax, as java enables crossed platform program. Java pipe code can be interpreted on any system that provides JVM(java virtual machine). Although java is a interpreted language, it is faster than basic, perl, etc.
5. **Robust:** java restrict in few key areas to find mistake in early stage of development. Java checks the error both during compile time as well as execution time. Hence program will behave in a predictable way in almost all condition.

   Main reason for program failure is memory management mistake or mishandled exceptional condition. Ex. Memory management in c++, you can allocate or de-allocate memory as and when require. Sometime during development you may forget to de-allocate the memory or you may de-allocate the memory which was used in later part of the program. Java handle this memory management problem very easily. Hence java is robust.
6. **Secure:** java is used in internet and there is chance of viral infection for normal program in internet but java has to be interpreted. But java has to be interpreted. If it does not understand byte code the machine totally refuse to accept the code in machine. Hence you can say java is secure. Java provides firewall between a network application and your computer by guarding against some programs which gather's information like credit card no, bank account, balance, etc.

7. **Architectural neutral:** O.S. up-gradation, processor upgrades and changes in core system resource all combined to make a program malfunction. Java designer have to make an attempt so that whatever be the changes in the machine, java program will have no effect and it will execute in desired way. Which means changes in the machine will hardly effect java program execution.
8. **Portable:** as most of the computer throughout the world has been connected through the world has been connected through internet for programming to be dynamically downloaded to all the various types of platforms connected to it. Hence some means to generating portable executable code is needed. This will help in ensuring security.
9. **High-performance:** java are comparable in speed to programs written in other complier based language like c and c++. Java is faster than other interpreter language like basic, perl, etc. since it is complied and interpreted, security of the program is more, considering security, platform independence, speed, we can say that java is a high performance language.
10. **Multi-threading:** it is the ability of the application to perform multiple tasks at the same time. Java is designed to meet the real word requirement of creating interactive networked program. The java runtime system comes with the sophisticated solution for multi process synchronization which helps the program to run smoothly. Core of the java is also multi threaded.
11. **Dynamic language:** java program can consist of many modules that are written by many programmers. These modules may undergo many changes. Java makes inter-connection between the modules at runtime which easily avoids the problem cost by the changes of the code use by your program thus you can say java is dynamic. Java has runtime capability which makes it possible to dynamically link the code in safe manner.

:Data types:

Data stored in memory can be of many types. Ex. Person age is stored as a numeric value where as his name and address is taken as character. The data types are used to define the possible operation on them & store value for more manipulation. The data types in java can be classified as: 1) Primitive data type or standard data type

2) Abstract data type or derived data type

The primitive data type can hold negative values however the keyword “Unsign” can be used to restrict the range of the value to only for the positive numbers. The data type Boolean can hold only the values true or false. And character can contain only single character.

The abstract or derived data types are based on primitive data types and have more functionality than primitive data types. **String** is an abstract data type that can store letters, digits and other characters like /,( ),$ but you can’t perform calculation on a variable of the string data type even if it is storing the digits.

:Dynamic initialization:

java allows variable to initialize dynamically using any expression or value valid at the time of variable declaration. Ex. int z1=0; int z = 2 * z1;

:Scope or life time of variable:

all variables used in java program does have scope within the block of java modules. The open ({) of the block or declaring variable specify beginning of the scope and closing bracket (}) indicate the end of the scope.

Ex:
```
public class demo()
    {
        public static void main(String a[])
        {
         int x=5;
         void cal()
            {
            int x=10;
            System.out.println(" x=" + x); // print value 10
            }
         System.out.println("outer x=" + x); // print value 5
         }
}
```

:Literals:

Literals are constant value assigning to variable which contains different types of data.

int literal: it can have decimal, octal or hexa decimal value as it can contain whole numbers.
Ex: int x =10; int x=012; int x=0xA;
float literal: floating point number represent decimal value with fractional components. The standard notation of floating point is 10.123 and in the scientific notation it will be 1.0123E+03. Where the first part is called mantissa and other part is called exponent. (part after E is other part) float z=12.45;
boolean literal: the boolean literal can take only 2 values true and false.
char literal: it takes only single char and that has to be given within the single quote.
Ex. char code= 'a';
String literal: also called as character array. It should be enclosed within double quotes.
Ex. String name = "arth";

:New operator:

when you create an object of a class, you need to allocate memory for it. To allocate the memory you need to use new operator.
Syntax: classname objname = new classname([argument list]);
Ex: emp e = new emp();
emp e1 = new emp(10,20);

:Type Conversion/Casting:

Every expression has a type that is determine by the component of expression.
Ex.1
double z;
int x=10;

float y=10.12;
z=x+y;
the expression to the right of "=" is solved first then resulting value stored in z. the expression to the right has variables of different data types i.e. int and float. The int value automatically promoted to higher data type float. At last this value is assign to variable z. which is of double data type. therefore the resulting value is double. Java automatically promotes value to higher data type to prevent loss of information which is called as implicit type conversion. It is done by system itself.
Ex.2
int x =(int)5 * (5/2);
in above example the expression to the right evaluate to decimal value and expression to the left of "=" is integer which can hold only whole numbers. Hence compiler will give an error specifying "incompatible type". This is because data can be loss when it is converted from higher data type to lower data type. The compiler request to type cast the given expression which is known as explicit type conversion or casting. Explicit type conversion has to be done when there is a possibility of loss of data during conversion.

: Constructor :

Constructors are methods with same name as that of class name. Constructor does not return any value. It is used for initializing the member data of class object. It gets invoked implicitly as soon as you create the object. **"A constructor with no argument is known as** default constructor **in java."**

Parameterized constructor

If the constructor takes the argument to initialized the values of member data then it is called as parameterized constructor. The value for this parameter should be given during the creation of object. The default argument is another way of giving the values to the constructor.

```
ex: class democonst
{
int ht,wd,dp;
democonst()
{
        ht=10;
        wd=5;
        dp=4;
}

democonst(int x, int y, int z)
{
        ht=x;
        wd=y;
        dp=z;
}
void volume()
{
        int vol;
        vol=ht*wd*dp;
```

```
                System. out. println("volume of box=" + vol);
}
public static void main(String args[])
{
                democonst d = new democonst();
                democonst d1 = new democonst(10,20,30);
                d.volume();
                d1.volume();
}
};
```

:Garbage Collection:

The objects are dynamically allocated by using new operator in c++. This dynamically allocated object must be manually released by use of delete operator but java takes different approach. It handles deallocation automatically. The technique used behind this is called garbage collection, in which when no reference to an object exist that object is assumed to be no longer needed. The memory occupied by the object can be released. There is no explicit way to destroy the object and java runtime approach takes value to garbage collection and this occurs when no reference is made to object later in the program. Garbage collection class can be use with the system,gc class. The value of freed object can go to system.gc.

:Finalize method:

Sometimes an object will need to perform some action when it is destroyed. Ex. if an object is holding some non java resources such as file handler or windows char-font then you might want to make sure that this resources are freed before an object is destroyed. To handle such situation java provides a mechanism called finalize. Finalize is a method which will have code to release the machine resources occupied by the object & gets invoked just before the object is destroyed.

Ex:
```
protected void finalize()
    {
                // code for finalization
    }
```

This is something similar to destructor in c++. Finalize is only called just prior to garbage collection and when garbage collection invoked we cannot know, so we even can't judge when finalized will execute.

:This Pointer:

Sometimes a method will need to refer to the object that has invoked it. To allow this, "This" keyword is used inside any methods that refer to current object. This pointer always refers to the object on which the method was invoked. You can use this pointer anywhere where reference to an object of the class type is permitted.

Ex:
```
demo(int x, int y, int z)
{
this.ht=x;
this.wd=y;
this.dp=z;
```

```
}
```

Program:

```
class student
{
int sid;
String sname;
 student(int id,String name)
{
 this.sid = id;
this.sname = name;
}
void display()
{
System.out.println(sid+" "+sname);
}
public static void main(String args[])
{
student s1 = new student(1,"Kuhu");
student s2 = new student(2,"Arth");
s1.display();
s2.display();
}
}
```

:Overloading:

In java, it is possible to define two or more method with the same name but their parameter declarations are different. When this is the case, the methods are said to be overloaded and the process is referred as method overloading. Method overloading is a process through which java implements polymorphism. When an overloaded method is invoked, java uses the type and no. of parameters as its guide to determine which version of the overloaded method actually calls.

Overloaded method must defer in type & no. of arguments while overloaded method may have different return types. The return type alone is insufficient to distinguish between two methods, as java does automatic type conversion which also play important role in overloading resolution.

Ex1. void volume(int x,int y)
void volume(int x,int y,int z)

Ex2. void volume(int x,int y)
void volume(float x, float y)

in above examples ex1 is correct but ex2 gives you an error when you call the method, as java takes automatic type conversion.

:Recursion:

Recursion is a process of defining something in terms of itself. It relates to java programming by allowing a method to call itself. This method of calling function itself is called as recursion. In this kind of programming, stack does a very important role by storing the value of the variables. Recursion process is little bit slow compare to iterative process.(do while/while/for loop)

:Access Specifier/Mode:

The accessibility of the data member or member function can be determined by different access specifier such as public, private and protected. When member of the class is define as **public** then any of the classes other than the original class can access this member function or member data. If the member of the class is specified as **private** than that member can only be access by member of the same class. Other class member cannot access it & that is why main is always defined as public. When no access specifier is defined by default in java it is public to its own package and private for other packages. **Protected** is used only when inheritance come into the picture. Where access to member is given to class where it is declared and its immediate subclasses.

:Final Variable, Method and Class:

Final variable:

If you make any variable as final, you cannot change the value of final variable(It will be constant). Ex. final int vote_min_age=18;

A final variable that is not initialized at the time of declaration is known as blank final variable. We must initialize the blank final variable in constructor of the class otherwise it will throw a compilation error.

```
class Demo{
  //Blank final variable
  final int MAX_VALUE;
  Demo(){
    //It must be initialized in constructor
    MAX_VALUE=100;
  }
```

A static final variable that is not initialized during declaration can only be initialized in static block.

```
class Example{
  //static blank final variable
  static final int ROLL_NO;
  static{
    ROLL_NO=1230;
  }
  public static void main(String args[]){
    System.out.println(Example.ROLL_NO);
  }
}
```

Output:

1230

Final method:

A final method cannot be overridden. Even though a sub class calls the final method of parent class but it cannot override it**.**

Program1:

```
class Bike
{
final void run(){System.out.println("running");}
}
class Honda  extends Bike{
void run(){System.out.println("running safely with 100kmph");}
public static void main(String args[])
{ Bike b1= new Bike();
    Honda honda1= new Honda();
//b1.run();
Honda1.run();
}
}
```

Output:

Compile Time Error

Program2:

```
class XYZ{
  final void demo(){
    System.out.println("XYZ Class Method");
  }
}
class ABC extends XYZ{
  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

XYZ Class Method

Final class:

If you make any class as final, you cannot extend it. You cannot have subclass of final class. Several classes in java are final e.g. string, integer and other wrapper classes.

Program:

```
final class XYZ{
}

class ABC extends XYZ{
  void demo(){
    System.out.println("My Method");
  }
```

```
  public static void main(String args[]){
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:
The type ABC cannot subclass the final class XYZ

Note:
1) A **constructor** cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an **interface** are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6)A final class not be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, **finally** and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during **garbage collection**.

Ref for final:
http://beginnersbook.com/2014/07/final-keyword-java-final-variable-method-class/
http://javarevisited.blogspot.in/2011/12/final-variable-method-class-java.html
http://www.javatpoint.com/final-keyword

:Static:

Static variable

Normally a class member must be accessed with the object however it is possible to create a member that can be used by itself without referring to any object. To create such member you should precede its declaration with the word static. When a member is declared as static it can be access before any object of its class are created. When a variable is declared with the keyword "static", its called a "**class variable**". All objects share the same copy of the variable. A class variable can be accessed directly with the class, without the need to create a object.

Program of counter without static variable

```
class Counter{
int count=0;//will get memory when instance is created
  Counter()
{
count++;
System.out.println(count);
}
  public static void main(String args[])
{
  Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
  }
```

```
    }
```
Output:
```
        1
        1
        1
```

Program of counter with static variable

```
class Counter{
static int count=0;//will get memory only once and retain its value

Counter(){
count++;
System.out.println(count);
}
  public static void main(String args[]){

Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```
Output:
```
        1
        2
        3
```

Static method:

Both member data and function can be made as static. Method declared as static have several restrictions.

1. They must access only static data.
2. They can call only static methods.
3. They can't refer to 'this' or 'super'.
4. static method can be **accessed directly** by the **class name** and doesn't need any object

When a member data is declared as static it gets only 1 memory location independent of any no. of object.

Program static method

```
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    static void change(){
                            college = "BBDIT";
                            }
     Student(int r, String n)
        {
```

```
        rollno = r;
        name = n;
        }
 void display ()
        {
        System.out.println(rollno+" "+name+" "+college);
        }
 public static void main(String args[])
{
   Student.change();
   Student s1 = new Student (111,"Karan");
   Student s2 = new Student (222,"Aryan");
   Student s3 = new Student (333,"Sonoo");
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Output:
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

Program Ref. http://www.javatpoint.com/static-keyword-in-java

:Inheritance:

When a class has more properties and it differs from another class by 1 or 2 properties, you can make a class called super class which will have generalized property in it and the subclasses will have the additional information but which are different from other subclasses. This property of object orientation is called as inheritance. To have inheritance, there should be some relationship between super and subclass.

In short **Inheritance** is a mechanism in which one object acquires all the properties and behaviors of parent object Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship. We use inheritance for 2 main purposes

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

Note: To reduce the complexity and simplify the language, **"multiple inheritance" is not supported in java**.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class hence java provide compile time error if you inherit 2 classes.

Syntax: **class** Subclass-name **extends** Superclass-name { //methods and fields }

Program1:

```
class Employee{
 float salary=40000;
}
```

```
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Example ref: http://www.javatpoint.com/inheritance-in-java

Program2:

```
class Calculation
{
   int z;

   public void addition(int x, int y){
      z = x+y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void Substraction(int x,int y){
      z = x-y;
      System.out.println("The difference between the given numbers:"+z);
   }
  }
public class My_Calculation extends Calculation{

   public void multiplication(int x, int y){
      z = x*y;
      System.out.println("The product of the given numbers:"+z);
   }

   public static void main(String args[]){
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.Substraction(a, b);
      demo.multiplication(a, b);
   }
}
```

Example Ref: http://www.tutorialspoint.com/java/java_inheritance.htm

:Super:

whenever a subclass needs to refer to its immediate super class. It can do so by the use of keyword super. Super has 2 general forms.

1) It is used to **invoke the superclass** constructor from subclass. (Calls to super class constructor)
2) It is used to **differentiate the members** of superclass from the members of subclass, if they have same names. (To access members of super class which are hidden by members of subclass.)

1)Using super to call super class constructor:

A subclass can call a constructor defined by super class using super([list of arguments])
Here argument list specifies parameter list needed by the constructor in super class. Super() must always be the first statement execute inside a subclass constructor.

Program1:

```
class Employee
{
        Employee()
        {
        System.out.println("Employee class Constructor");
        }
}
class HR extends Employee
        {
        HR()
        {
        super(); //will invoke or call parent class constructor
        System.out.println("HR class Constructor");
        }
}
class Supercons
{
        public static void main(String[] args)
        {
        HR obj=new HR();
        }
}
```

Program1 Ref. http://www.tutorial4us.com/java/java-super-keyword

2)To access super class members(variable and function):

The 2dn form of super somewhat act like this pointer except that it always refer to super class of subclass in which it is used. Syntax: super.member
Here member can be either member data(super class variable) or member function. This form of super is most applicable in which member names of subclass hides the same name members in super class.

Program2:

```
public class Super_Ex{

        void display() {

                System.out.println("Base Class method called");
        }
}
 class SubClass extends Super_Ex {

        void display() {

                super.display();
                System.out.println("Sub Class method called");
        }
}
class MainClass {

        public static void main(String args[]) {

                SubClass obj = new SubClass();
                obj.display();
        }
}
```

Program2 Ref. http://www.sampleprogramz.com/java/super.php

:Overriding:

Overriding is a part of object oriented programming in which super class and subclass may have same named function where the codes are different but signature of the method is same.

1. Name of the method should be same
2. Return type must be same
3. Argument should be same
4. Access specifier should be same
5. Exception it gives should be same

When an overridden method is called from within a subclass it will always refer to version of the method written in subclass rather than the super class method. So super class method is hiding from subclass same name method using overriding concept. This problem can be solved using keyword "super" as discussed earlier.

- Rules for overriding:

1. The argument list should be exactly the same as that of the overridden method.
2. The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
3. The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.

4. Instance methods can be overridden only if they are inherited by the subclass.
5. A method declared final cannot be overridden.
6. A method declared static cannot be overridden but can be re-declared.
7. If a method cannot be inherited, then it cannot be overridden.
8. A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
9. A subclass in a different package can only override the non-final methods declared public or protected.
10. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
11. Constructors cannot be overridden.

- Rules Ref http://www.tutorialspoint.com/java/java_overriding.htm

Late Binding/Dynamic Binding/ Dynamic Method Dispatch/ Runtime Polymorphism:

Whenever subclass and superclass has same name method then it is called overriding. It is nothing but using same name convention in the memory. Dynamic method dispatch is the mechanism by which a call to overridden method is resolved at runtime rather than compile time. (However, it can't call any of the newly added methods by the subclass but a call to an overridden methods results in calling a method of that object whose reference is stored in the super class reference.) Dynamic method dispatch is important because it is the way in which java implements runtime polymorphism in this case a super class reference variable can refer to subclass object. Java uses this fact to resolve the calls to overridden method at runtime.

When an overridden method is called through superclass reference variable, java determines which version of that method to execute based on the type of object being referred at the time call occurs. In other words it is **the type of the object being referred to** that determines the method to get executed.

Program:

```
class A
{
void callme()
{
System.out.println("Inside A's callme method");
}
};
class B extends A
{
void callme()   // override callme()
{
System.out.println("Inside B's callme method");
}
};
class C extends A
```

```
{
 void callme() // override callme()
{
 System.out.println("Inside C's callme method");
}
};
public class Dynamic_disp
{
public static void main(String args[])
{
 A a = new A(); // object of type A
 B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

Output:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Here reference of type A, called r, is declared. The program then assigns a reference to each type of object to r and uses that reference to invoke callme( ).As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

Program Ref: http://www.javatutorialprograms.com/2014/01/using-dynamic-method-dispatch-in-java.html

Abstract Class & Methods and Concrete Class & Methods:

A class which contains the **abstract** keyword in its declaration is known as **abstract class**.
An ***abstract class*** is a class which cannot be instantiated, meaning you cannot create new object of an abstract class. The purpose of an abstract class is to act as a place holder or base class from which other classes are derived. If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the **method in the parent class as abstract**.
The derived class is expected to provide implementations for the methods that are not implemented in the base class. A derived class that implements all the missing functionality of abstract class is called a ***concrete class***. A concrete class has **concrete methods**, i.e., with code and other functionality. This class a may extend (inherit) an abstract class or implements an interface.
Rules for abstract class:

1. Abstract classes may or may not contain *abstract methods* ie., methods without body. Ex. public void get();

2. But, if a class has at least one **abstract method**, then the class **must** be declared **abstract class.**
3. If a class is declared abstract it cannot be instantiated.
4. To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.
5. If you inherit an abstract class you have to provide implementations to all the abstract methods in it.

- Rules Ref: http://www.tutorialspoint.com/java/java_abstraction.htm

Rules for abstract method:

1. **"abstract"** keyword is used to declare the method as abstract.
2. You have to place the **abstract** keyword before the method name in the method declaration.
3. An abstract method contains a method signature, but no method body.
4. Instead of curly braces an abstract method will have a semi colon (;) at the end

- Rules Ref: http://www.tutorialspoint.com/java/java_abstraction.htm

Program:

```
abstract class Bike
{
  Bike()
    {
    System.out.println("bike is created");
    }
  abstract void run();
  void changeGear()
    {
    System.out.println("gear changed");
    }
};
 class Honda extends Bike
{
 void run()
    {
    System.out.println("running safely..");
    }
} ;
 class TestAbstraction2
{
 public static void main(String args[])
{
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Output:
bike is created
running safely..
gear changed

Program Ref: http://www.javatpoint.com/abstract-class-in-java

Differences: abstract class vs final class.

Property	Abstract class	Final class
Subclassing	Should be subclassed to override the functionality of abstract methods	Can never be subclassed as final does not permit
Method alterations	Abstract class methods functionality can be altered in subclass	Final class methods should be used as it is by other classes
Instantiation	Cannot be instantiated	Can be instantiated
Overriding concept	For later use, all the abstract methods should be overridden	Overriding concept does not arise as final class cannot be inherited
Inheritance	Can be inherited	Cannot be inherited
Abstract methods	Can contain abstract methods	Cannot contain abstract methods
Partial implementation	A few methods can be implemented and a few cannot	All methods should have implementation
Immutable objects	Cannot create immutable objects (infact, no objects can be created)	Immutable objects can be created (eg. String class)
Nature	It is an incomplete class (for this reason only, designers do not allow to create objects)	It is a complete class (in the sense, all methods have complete functionality or meaning)
Adding extra functionality	Extra functionality to the methods can be added in subclass	No extra functionality can be added and should be used as it is

Difference Ref: http://way2java.com/oops-concepts/java-made-clear-difference-between-abstract-class-and-final-class/

:Interface:

Interface is something to class which defines what to do & not how to do. Interface are syntactically similar to classes but they lack instance variable and their methods are declared without any body. Once the interface is define, any number of classes can implement an interface & one class can implement any number of interface.
To implement an interface a class must create a complete set of methods defined by the interface. However each class is free to determine the details of its own implementation. By providing the 'interface' keyword, java allows you to have "1 interface multiple methods" which is called as property polymorphism.
Syntax:
Accesss specifier interface interface_name
{
final/static datatype variablename1 = value ;

```
final/static datatype variablename2 = value ;
:
:
datatype method1 ([argument list]);
datatype method2 ([argument list]);
:
:
}
```

Example:

```
public interface bright
{
        final int max_tv = 50;
        final int min_tv = 20;
        void increase(int x);
        void decrease(int x);
}
```

1) Here in the place of access specifier, you can either use public or no access specifier. If you declare the interface with the public access specifier, you can use this interface within package or in any other package but without access specifier the interface which is declared can be use only within that package.
2) Interface name specify any valid interface name.
3) The variables declared in the interface will be either static or final meaning they can't be change by the implementing class & they must also initialize with some constant value.
4) The methods which are declared in the interface have no bodies. They just ends with semicolon after parameter list. They are essentially abstract method that means they just have the signature of the method but no codes within body.

Implementing an interface:

Once an interface has been defined, one or more classes implement that interface. To implement the interface, include "**implements**" clause in the class definition.

Syntax:

```
Class_name implements interface1, interface2,……..
```

The methods (of interface) which are implemented by class should be declared as public during overriding although you don't have specified those methods as public during interface declaration.

Accessing the implementation through interface reference:

This kind of accessing is just because during the dynamic method dispatch or dynamic runtime polymorphism the compiler needs to have a reference of the super class where as in this case interface reference use for dynamic binding.

Syntax:

```
Interfacename object = new classname([argument list]);
```

Ex: bright t= new tv();

Where bright is an interface and tv is class.

Program:

```
interface MyInterface
```

```
{
   public void method1();
   public void method2();
}
class XYZ implements MyInterface
{
  public void method1()
  {
     System.out.println("implementation of method1");
  }
  public void method2()
  {
     System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
     MyInterface obj = new XYZ();
     obj. method1();
  }
}
```

Output:
implementation of method1

Program ref: http://beginnersbook.com/2013/05/java-interface/

Interface extension:

One interface can inherit another interface by the use of keyword “extends”. Syntax is same as for inheriting classes. When a class implements an interface that inherits another interface. It must provide implementation for all the methods define within the interface chain.

Program:

```
public interface I1
{
void method1();
}
public interface I2 extends I1
{
   void method2();
}
class Demo implements I2
{
  public void method1()
{
        System.out.println("Interface1’s method1");
  }
  public void method2()
{
     System.out.println("Interface2’s method2");
```

```
  }
};//Demo
public class IntExt
{
public static void main(String arg[])
  {
     Demo d = new Demo();
     d. method1();
     d. method2();
  }
};//IntExt
```

In above case interface I2 inherits properties of interface I1. Although class Demo implements the interface I2, it should override all the methods declared in interface I2 and I1 both.

Partial Implementation:

If a class includes an interface but does not fully implement the methods defined by the interface then it is called as partial implementation. The class which uses the partial implementation should be declared as abstract class and the subclass of an abstract class must override the methods which are not overridden in the abstract class.

Program:

```
public interface I1
{
void method1();
void method2();
}
abstract class Demo implements I1
{
  public void method1()
{
        System.out.println("method1 displayed");
  }
 };//Demo

public class IntExt extends Demo
{
public void method2()
{
        System.out.println("method2 displayed");
  }
public static void main(String arg[])
  {
     Demo d = new Demo();
     d. method1();
     d. method2();
  }
};//IntExt
```

In above case interface I1 implemented by class Demo but it only override method1. So Demo class declared as an abstract class and method2 which is not overridden by class Demo must be overridden in subclass of Demo class i.e. IntExt.

Differences: abstract class vs interface

Abstract class	**Interface**
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Note: Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Difference Ref: http://www.javatpoint.com/difference-between-abstract-class-and-interface

Difference: Concrete class vs. Abstract class vs. Interface.

1. A concrete class has concrete methods, i.e., with code and other functionality. This class a may extend an abstract class or implements an interface.
2. An abstract class must contain at least one abstract method with zero or more concrete methods
3. An interface must contain only method signatures and static members.
4. An interface does not have definitions of the methods declared in it.
5. An abstract class may have some definition and at least one abstract method.
6. A subclass of an abstract class must either implement all the abstract methods of the abstract class or declare itself as abstract.

Difference Ref.http://www.careerride.com/Java-concrete-class-abstract-class-interface.aspx

Key points: Here are the key points to remember about interfaces:
1) We can't instantiate an interface in java.
2) Interface provides complete abstraction as none of its methods can have body. On the other hand, abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
3) implements keyword is used by classes to implement an interface.
4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
5) Class implementing any interface must implement all the methods, otherwise the class should be declared as "abstract".
6) Interface cannot be declared as private, protected or transient.
7) All the interface methods are by default abstract and public.
8) Variables declared in interface are public, static and final by default.

```
interface Try
{
  int a=10;
  public int a=10;
  public static final int a=10;
  final int a=10;
  static int a=0;
}
```

All of the above statements are identical.
9) Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

```
interface Try
{
     int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.
10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static final by default and final variables can not be re-initialized.

```
Class Sample implements Try
{
 public static void main(String arg[])
 {
    x=20; //compile time error
 }
}
```

11) Any interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.
12) A class can implement any number of interfaces.
13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
  public void aaa();
}
interface B
{
  public void aaa();
}
class Central implements A,B
{
  public void aaa()
  {
      //Any Code here
  }
  public static void main(String arg[])
  {
      //Statements
   }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
  public void aaa();
}
interface B
{
  public int aaa();
}

class Central implements A,B
{

  public void aaa() // error
  {
  }
  public int aaa() // error
  {
  }
  public static void main(String arg[])
  {

  }
}
```

15) Variable names conflicts can be resolved by interface name

```
interface A
```

```
{
   int x=10;
}
interface B
{
   int x=100;
}
class Hello implement A,B
{
   public static void Main(String arg[])
   {

      System.out.println(x); // reference to x is ambiguous both variables are x
      System.out.println(A.x);
      System.out.println(B.x);
   }
}
```

Key points Ref: http://beginnersbook.com/2013/05/java-interface/

Advantages of interfaces:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, However by using interfaces you can achieve the same. A class can extend only one class but can implement any number of interfaces.
3. Interface provides a contract for all the implementation classes, so it's good to code in terms of interfaces because implementation classes can't remove the methods we are using.
4. Interfaces are good for starting point to define Type and create top level hierarchy in our code.
5. Since a java class can implements multiple interfaces, it's better to use interfaces as super class in most of the cases.

Disadvantages of interfaces:

1. We need to choose interface methods very carefully at the time of designing our project because we can't add or remove any methods from the interface at later point of time, it will lead compilation error for all the implementation classes. Sometimes this leads to have a lot of interfaces extending the base interface in our code that becomes hard to maintain.
2. If the implementation class has its own methods, we can't use them directly in our code because the type of Object is an interface that doesn't have those methods. For example, in above code we will get compilation error for code shape.getRadius(). To overcome this, we can use **typecasting** and use the method like this:

```
1   Circle c = (Circle) shape;
2    c.getRadius();
```

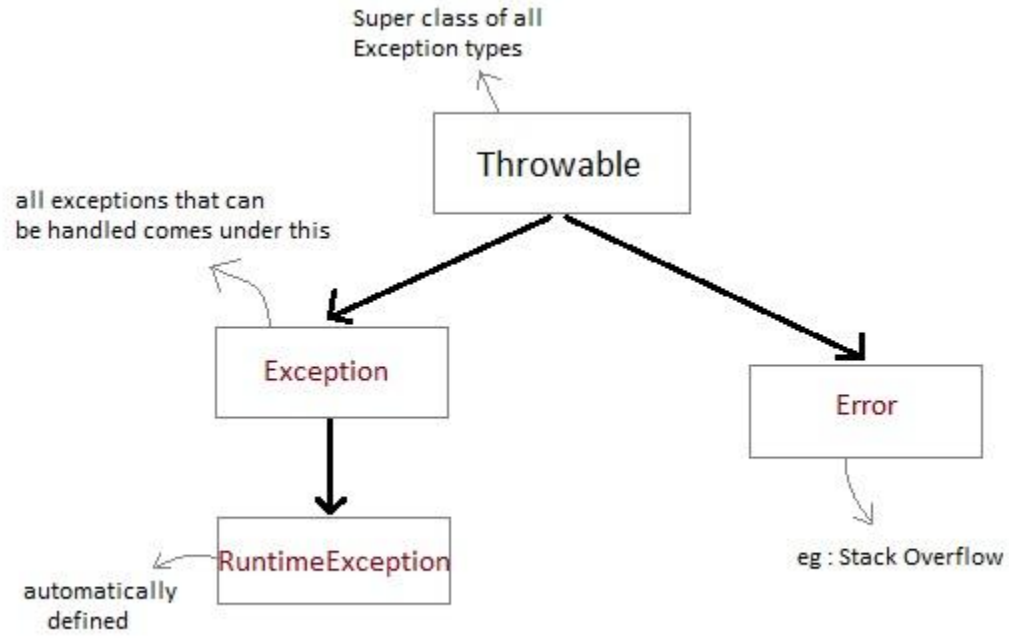
Although class typecasting has its own disadvantages.

Advantage and disadvantage Ref: http://beginnersbook.com/2013/05/java-interface/
http://www.journaldev.com/1601/what-is-interface-in-java-example-tutorial

:Exception Handling:

Exception:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled. Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).



- The 'Throwable' class is the superclass of all errors and exceptions in the Java language and is at the top of exception class hierarchy.
- Subclasses of throwable class are error and exception.
- Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.
- Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Few examples –
    1. DivideByZero exception
    2. NullPointerException
    3. ArithmeticException
    4. ArrayIndexOutOfBoundsException
- RuntimeException is a subclass of Exception. Exceptions under this class are automatically defined for programs.

Exception types:

An exception can occur for many different reasons, below given are some scenarios where exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. Based on these we have categories exception mainly in **two** types

Checked and Unchecked where **Error** is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception / Programmatic Exceptions / compile time exceptions:

1. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException, File that need to be opened is not found, ClassNotFoundException, IllegalAccessException, NoSuchFieldException EOFException etc.
2. The exception that can be predicted by the programmer.
3. Checked exceptions are checked at these are also called as compile time exceptions.
4. Compile-time exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.

Unchecked Exception / JVM Exceptions / Runtime Exceptions:

- Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit. In short Unchecked exceptions are checked at runtime & ignored at compile time.
- Unchecked exceptions are the class that extends RuntimeException. Which means RuntimeException is the parent class of all runtime exceptions.
- If we are throwing any runtime exception in a method, it's not required to specify them in the method signature throws clause.
- Runtime Exceptions are cause by bad programming. These include programming bugs, such as logic errors or improper use of an API. For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.
  Examples
  ArithmeticException
  ArrayIndexOutOfBoundsException
  NullPointerException
  NegativeArraySizeException etc.

Error

- Error is irrecoverable
- These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

- Errors are typically ignored in your code because you can rarely do anything about an error.That's why we have a separate hierarchy of errors and we should not try to handle these situations.
- Some of the common Errors are OutOfMemoryError and StackOverflowError, hardware failure, JVM crash or out of memory error, VirtualMachineError, AssertionError etc.

Java provides a robust and object oriented way to handle exception scenarios, known as **Java Exception Handling**.

Exception handler:

In java, exception handling is done using five keywords,

1. **try**
2. **catch**
3. **throw**
4. **throws**
5. **finally**

When exception occurs, the execution of a program is transfer to an appropriate exception handler

Try & catch:

- The codes which have possibility to generate exceptions are placed in the try block, when an exception occurs, that exception is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.
- try is the start of the block and catch is at the end of try block to handle the exceptions.
- catch block requires a parameter that should be of type Exception.
- Code within a try/catch block is referred to as protected code,

Syntax:

```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

Program:

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
      // Try block to handle code that may cause exception
      num1 = 0;
      num2 = 62 / num1;
      System.out.println("Try block message");
    } catch (ArithmeticException e) {
        // This block is to catch divide-by-zero error
        System.out.println("Error: Don't divide a number by zero");
     }
    System.out.println("I'm out of try-catch block in Java.");
```

```
  }
}
```

Output:

Error: Don't divide a number by zero
I'm out of try-catch block in Java.

Program Ref: http://beginnersbook.com/2013/04/try-catch-in-java/

Multiple catch blocks in Java

We can have multiple catch blocks with a try and try-catch block

Syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

1. A try block can have any number of catch blocks.
2. A catch block that is written for catching the class Exception can catch all other exceptions

Syntax:

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

3. If multiple catch blocks are present in a program then the above mentioned catch block should be placed at the last as per the exception handling best practices.
4. If the try block is not throwing any exception, the catch block will be completely ignored and the program continues.
5. If the try block throws an exception, the appropriate catch block (if one exists) will catch it
–catch(ArithmeticException e) is a catch block that can catch ArithmeticException
–catch(NullPointerException e) is a catch block that can catch NullPointerException
6. All the statements in the catch block will be executed and then the program continues.

Program:

```
class Example2{
   public static void main(String args[]){
     try{
        int a[]=new int[7];
        a[4]=30/0;
        System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
```

```
      System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
      System.out.println("Warning: Some Other exception");
    }
   System.out.println("Out of try-catch block...");
  }
}
```

Output:
Warning: ArithmeticException
Out of try-catch block...

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it has the ability to handle all exceptions. This catch block should be placed at the last to avoid such situations.

Program Ref: http://beginnersbook.com/2013/04/try-catch-in-java/

Nested try catch blocks in Java

The try catch blocks can be nested. One try-catch block can be present in the another try's body. This is called Nesting of try catch blocks. Each time a try block does not have a catch handler for a particular exception, the stack is unwound and the next try block's catch (i.e., parent try block's catch) handlers are inspected for a match.
If no catch block matches, then the java run-time system will handle the exception.
Syntax:

```
try
{
statement 1;
statement 2;
try
{
statement 3;
statement 4;
}
catch(Exception e)
{
//Exception Message
}
}//main try over
catch(Exception e) //Catch of Main(parent) try block
{
//Exception Message
}//main catch over
```

The main point to note here is that whenever the child try-catch blocks are not handling any exception, the control comes back to the parent try-catch if the exception is not handled there also then the program will terminate abruptly.

Program:

```
class Nest{
  public static void main(String args[]){
          //Parent try block
    try{
         //Child try block1
       try{
          System.out.println("Inside block1");
          int b =45/0;
          System.out.println(b);
       }
       catch(ArithmeticException e1){
          System.out.println("Exception: e1");
       }
       //Child try block2
       try{
          System.out.println("Inside block2");
          int b =45/0;
          System.out.println(b);
       }
       catch(ArrayIndexOutOfBoundsException e2){
          System.out.println("Exception: e2");
       }
      System.out.println("Just other statement");
   }
   catch(ArithmeticException e3){
          System.out.println("Arithmetic Exception");
       System.out.println("Inside parent try catch block");
   }
   catch(ArrayIndexOutOfBoundsException e4){
          System.out.println("ArrayIndexOutOfBoundsException");
       System.out.println("Inside parent try catch block");
   }
   catch(Exception e5){
          System.out.println("Exception");
       System.out.println("Inside parent try catch block");
    }
    System.out.println("Next statement..");
  }
}
```

Output:

```
Inside block1
```

```
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

Program Ref: http://beginnersbook.com/2013/04/nested-try-catch/ & http://www.studytonight.com/java/try-and-catch-block.php

Throw:

throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception. If it doesn't find match it will go for another catch & so..on. if it doesn't find match at all then a default handler halts the program and gives stack trace message. We can throw either checked or uncheked exceptions in java by throw keyword. The throw keyword is mainly used to throw custom exception or user defined exception.

Syntax :

throw Throwableobject

Creating object of Throwable class

There are two possible ways to get an instance of class Throwable,

- Using a parameter in catch block.
- Creating instance with new operator.

Example:

throw new IOException("sorry device error");

This constructs an instance of IOException with name sorry device error.

new NullPointerException("test");

This constructs an instance of NullPointerException with name test.

Program:

```
class Test
{
 static void avg()
 {
  try
  {
   throw new ArithmeticException("demo");
  }
  catch(ArithmeticException e)
  {
   System.out.println("Exception caught");
  }
 }

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

```
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

Throws:

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that caller can guard themselves from these exceptions using exception handling codes so normal flow can be maintained.A method can do so by using the throws keyword.

Syntax :

```
type method_name(parameter_list) throws exception_list
{
 //definition of method
}
```

NOTE : It is necessary for all exceptions, except the exceptions of type Error and RuntimeException, or any of their subclass.

Program:

```
class Test
{
 static void check() throws ArithmeticException
 {
  System.out.println("Inside check function");
  throw new ArithmeticException("demo");
 }
 public static void main(String args[])
 {
  try
  {
   check();
  }
  catch(ArithmeticException e)
  {
   System.out.println("caught" + e);
  }
 }
}
```

Ref: http://www.studytonight.com/java/throw-throws-and-finally-keyword.php

Advantage of throws:

Now Checked Exception can be propagated (forwarded in call stack).
It provides information to the caller of the method about the exception.

Difference between throw and throws in Java

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.

1)Java throw example

```
void m(){
throw new ArithmeticException("sorry");
}
```

2)Java throws example

```
void m()throws ArithmeticException{
//method code
}
```

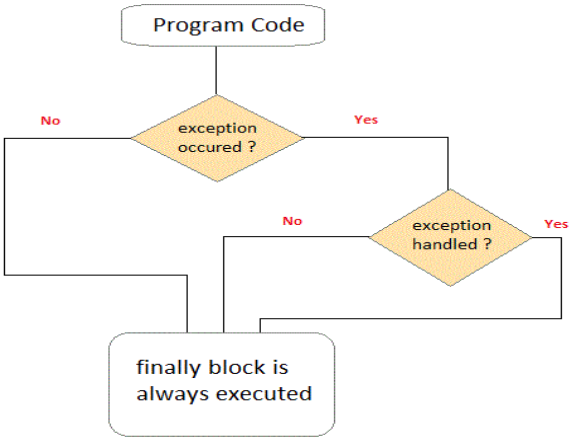
3) Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

Difference Ref: http://www.javatpoint.com/difference-between-throw-and-throws-in-java

Finally:

A finally keyword is used to create a block of code that follows a try block. A finally block of code always executes whether or not exception has occurred. A finally block appears at the end of catch block. We can have only one finally block for each try statement.



1. If exception occurs in try block's body then control immediately transferred (**skipping rest of the statements in try block**) to the catch block. Once catch block finished execution then **finally block** complete its execution and then the rest of the program.
2. It is not mandatory to include a finally **block** at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch blocks.
3. If there is no exception occurred in the code which is present in try block then first, the try block gets executed completely and then control gets transferred to finally block (**skipping catch blocks**).

4. If a **return statement** is encountered either in try or catch block. In such case also **finally runs**. Control first goes to finally and then it returned back to **return statement.**

Syntax:

```
try
{
  //Protected code
}

catch(ExceptionType e)
{
  //Catch block
}
: ….
finally
{
  //The finally block always executes.
}
```

Program:

```
Class ExceptionTest
{
 public static void main(String[] args)
 {
  int a[]= new int[2];
  System.out.println("out of try");
  try
  {
   System.out.println("Access invalid element"+ a[3]);
   /* the above statement will throw ArrayIndexOutOfBoundException */
  }
  finally
  {
   System.out.println("finally is always executed.");
  }
 }
}
```

Output:

```
Out of try
finally is always executed.
Exception in thread main java. Lang. exception array Index out of bound exception.
```

In above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

Ref: http://beginnersbook.com/2013/05/flow-in-try-catch-finally/ & http://www.studytonight.com/java/throw-throws-and-finally-keyword.php

User-defined Exceptions:

You can create your own exceptions in Java by creating your own exception sub class simply by extending java Exception class. You can define a constructor for your Exception sub class (not compulsory) and you can override the toString() function to display your customized message on catch. **User defined exceptions in java** are also known as **Custom exceptions**.

All exceptions must be a child of Throwable.

If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

If you want to write a runtime exception, you need to extend the RuntimeException class.

Program1:

```
class MyException extends Exception{
   String str1;
   MyException(String str2) {
     str1=str2;
   }
   public String toString(){
     return ("Output String = "+str1) ;
   }
}
class CustomException{
   public static void main(String args[]){
     try{
        throw new MyException("Custom");
        // I'm throwing user defined custom exception above
     }
     catch(MyException exp){
        System.out.println("Hi this is my catch block") ;
        System.out.println(exp) ;
     }
   }
}
```

Output:

```
Hi this is my catch block
Output String = Custom
```

Points to Remember

1. Extend the Exception class to create your own exception class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

Ref: http://www.tutorialspoint.com/java/java_exceptions.htm & http://www.studytonight.com/java/create-your-own-exception.php

Program2:

```
class MyOwnExceptionClass extends Exception {
    private int price;
```

```
    public MyOwnExceptionClass(int price)
          {
           this.price = price;
           }
          public String toString()
          {
           return "Price should not be in negative, you are entered" +price;
           }
 }
public class Client {
public static void main(String[] args)throws Exception
{
  int price = -120;

  if(price < 0)
    throw new MyOwnExceptionClass(price);
  else
    System.out.println("Your age is :"+price);
  }
}
```

Program2 Ref: http://www.java4s.com/core-java/how-to-create-user-defined-exception-in-java/

Program3:

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
class TestCustomException1{
    static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
  }
}
```

Program3 Ref: http://www.javatpoint.com/custom-exception

Rules:
1. We can't have catch or finally clause without a try statement.

2. A try statement should have either catch block or finally block, it can have both blocks.
3. We can't write any code between try-catch-finally block.
4. We can have multiple catch blocks with a single try statement.
5. Try-catch blocks can be nested similar to if-else statements.
6. We can have only one finally block with a try-catch statement.
7. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
8. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
9. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.
10. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
11. Only the object of Throwable class or its subclasses can be thrown.

Rules Ref: http://www.journaldev.com/1696/java-exception-handling-tutorial-with-examples-and-best-practices
http://www.studytonight.com/java/try-and-catch-block.php

:Summary:

- An exception is an event, which occurs during the execution of a program, that interrupts the normal flow of the program. It is an error thrown by a class or method reporting an error in code.
- The ***'Throwable'*** class is the superclass of all errors and exceptions in the Java language
- Exceptions are broadly classified as ***'checked exceptions'*** and ***'unchecked exceptions'***. All RuntimeExceptions and Errors are unchecked exceptions. Rest of the exceptions are called checked exceptions. Checked exceptions should be handled in the code to avoid compile time errors.
- Exceptions can be handled by using ***'try-catch'*** block. Try block contains the code which is under observation for exceptions. The catch block contains the remedy for the exception. If any exception occurs in the try block then the control jumps to catch block.
- If a method doesn't handle the exception, then it is mandatory to specify the exception type in the method signature using ***'throws'*** clause.
- We can explicitly throw an exception using ***'throw'*** clause

Difference between final, finally and finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

1)Java final example

```
class FinalExample{
public static void main(String[] args){
final int x=100;
```

```
x=200;//Compile Time Error
}}
```

2)Java finally example

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}}
```

3) Java finalize example

```
class FinalizeExample
{
protected void finalize()
{
System.out.println("finalize called");
}
public static void main(String[] args)
{
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}
}
```

Difference Ref: http://www.javatpoint.com/difference-between-final-finally-and-finalize