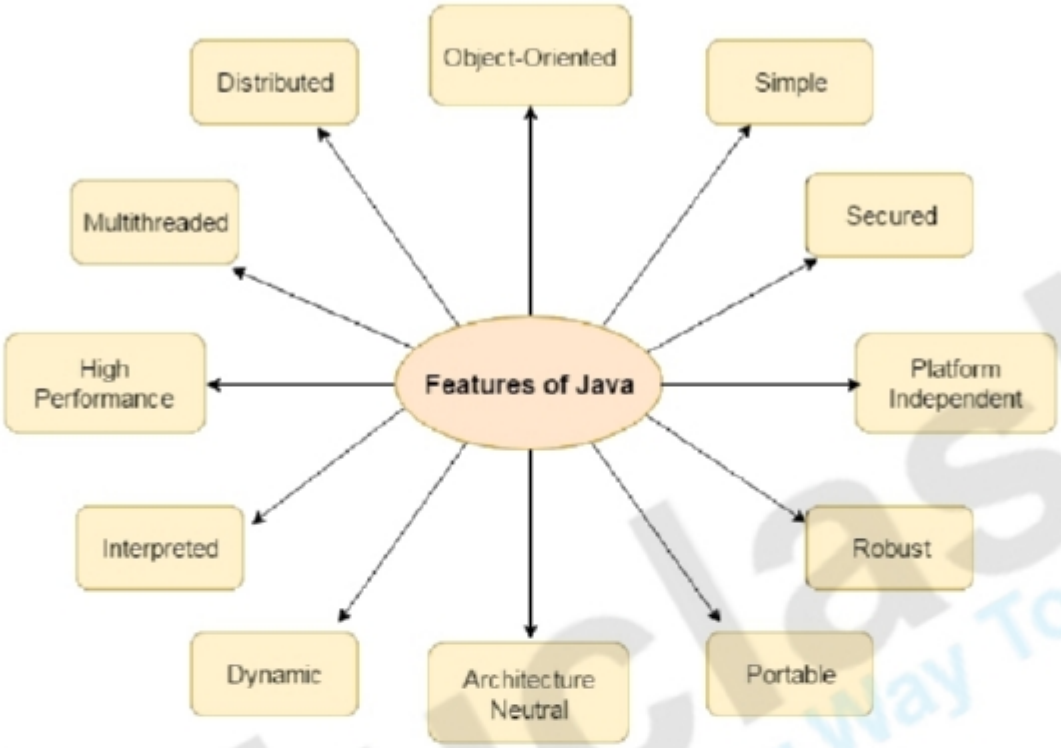
1. Write a features of java programming or java?

Ans: There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

According to Sun, Java language is simple because:

syntax is based on C++ (so easier for programmers to learn it after C++).

removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

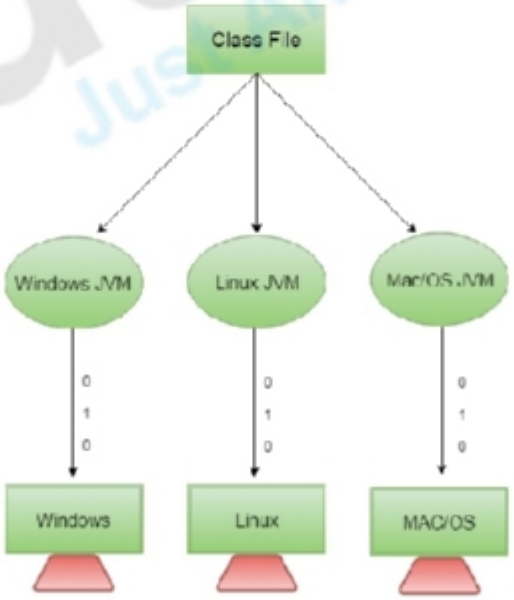
Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent



A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

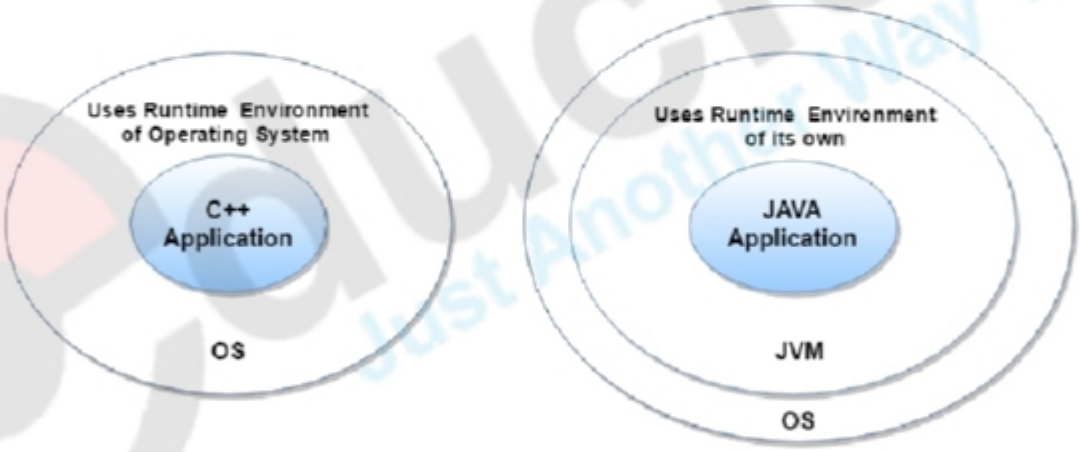
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is secured because:

- **No explicit pointer**
- **Java Programs run inside virtual machine sandbox**



- **Classloader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.

These securities are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

There is no implementation dependent features e.g. size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable

We may carry the java bytecode to any platform.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

2. Explain the JVM(JAVA VIRTUAL MACHINE) architecture?

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

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

JVM is the engine that drives the java code.

Mostly in other Programming Languages, compiler produce code for a particular system but Java compiler produce Bytecode for a Java Virtual Machine.

Bytecode is an intermediary language between Java source and the host system.

It is the medium which compile Java code to bytecode which get interpret on different machine and hence it makes it Platform/Operating system independent.

What is JVM

It is:

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

What it does

The JVM performs following operation:

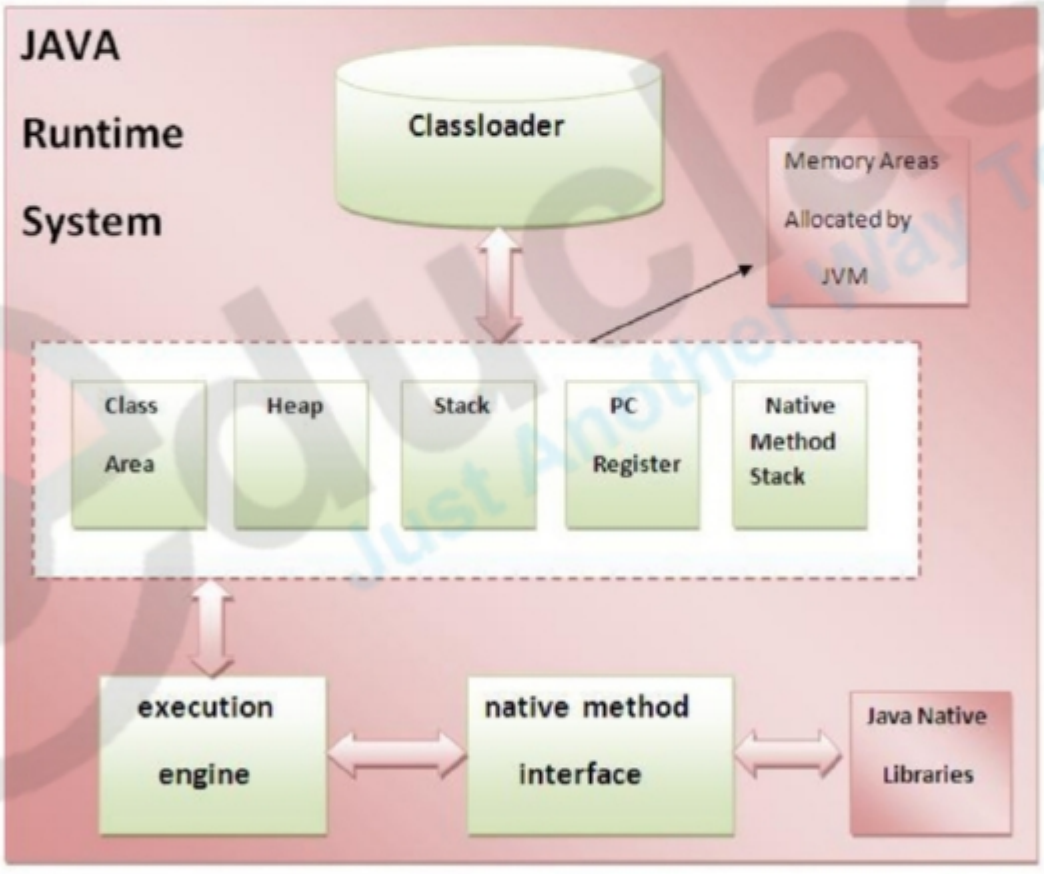
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

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

Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

How JVM is created(Why JVM is virtual):

When JRE installed on your machine, you got all required code to create JVM. JVM is created when you run a java program, e.g. If you create a java

program named FirstJavaProgram.java. To compile use – java FirstJavaProgram.java and to execute use – java FirstJavaProgram. When you run second command – java FirstJavaProgram, JVM is created. That's why it is virtual.

Lifetime of JVM:

When an application starts, a runtime instance is created. When application ends, runtime environment destroyed. If n no. of applications starts on one machine then n no. of runtime instances are created and every application run on its own JVM instance.

Main task of JVM:

- 1. Search and locate the required files.
- 2. Convert byte code into executable code.
- 3. Allocate the memory into ram
- 4. Execute the code.
- 5. Delete the executable code.

3. Explain the primitive & non-primitive datatype in java?

Ans: A data type is a classification of data, which can store a specific type of information. Data types are primarily used in computer programming, in which variables are created to store data. Each variable is assigned a data type that determines what type of data the variable may contain.

The term "data type" and "primitive data type" are often used interchangeably. Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types. Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "lastname" and define it as a string data type. The variable will then store data as a string of characters.

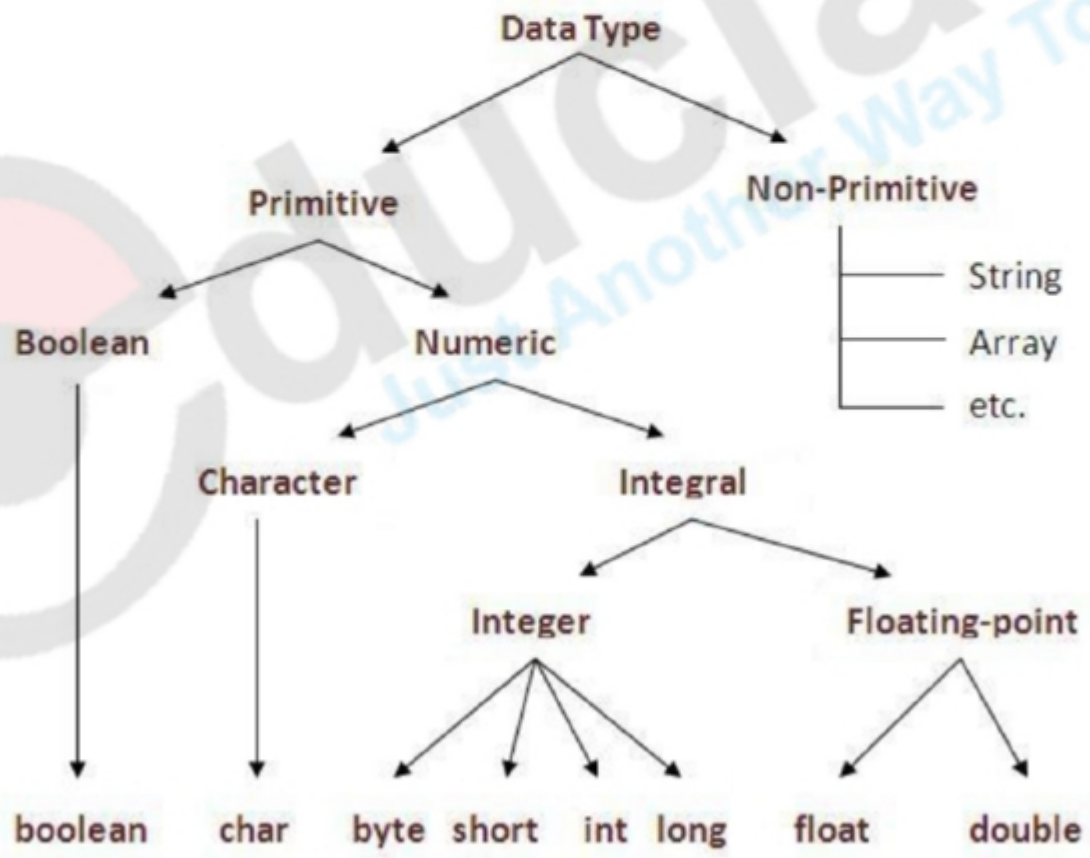
Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data. In the Java programming language, non-primitive data types are simply called "objects" because they are created, rather than predefined. While an object may contain any type of data, the information referenced by the object may still be stored as a primitive data type.

A variable is a named memory location or a name given to a memory location. Like other languages in java also used to store the data.For example, If you want store a value 10 in memory location, one write as

Example: int a=10

Here 'a' is the name give to the memory location where the value of '10' is located and the word ' int' tells that what type of data we are storing in that memory location. So, it is called " data type". Just like this, in java it is necessary to tell what type of data we are handling to JVM and to this we have data types.

" java is a strongly types languages" that in java it is necessary to tell what type of data we are handling before it can be used and it also tells what types of operation that can be carries out on this. There mainly two types of data types are available in java.



1. Java Primitive Data Types (OR) Java Basic data types

2. Java Non-Primitive Data Types (OR) Derived data types

Primitive Or Basic Data Types In Java

java define eight primitive data types namely byte, short, int,long, char, float, double and boolean. These are also called as intrinsic or built-in types. The integer group includes byte, short, int, and long and the floating group includes float and double.

Byte: A byte data type is a 8 bit signed two's complement integer. It can be used as alternative to integer where large amount of arrays what to store information. The maximum value is -128 and the minimum value is +127

Short: short is a signed 16 bit type. It has a range from -32,768 to 32,767. It is probably the least used data type.

Int: The most commonly used data types is integer.It is signed 32-bit type that has a range from 2^{63} and a maximum value of 2^{63}-1. In Java SE 8 and later, you can use the `long` data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 2^{64}-1

long: long is a signed 64-bit type and is useful for those occasions where an integer type is not enough to hold the desired value.We make integers long by appending "L" or "l" at the end of the number as

123L or 123l

integer types can hold only whole numbers and therefore we use another type known as floating point type to hold numbers containing fractional parts.

float: the type float specifies a single precision value that uses 32 bits of storage. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision. For example, float can be useful when representing dollars and cents.

double: Double precision, as denoted by the double keyword, uses 64 bits to store a value.

char: In C/C++ the character data type can store 8 bits. But it is not the case in java. Because java follows unicode system to represent all the characters found in all the human languages. So, it requires 16 bits.

boolean: a boolean is a 1 bit data type used to represent two values TRUE or FALSE.

Default Values

It is always not necessary to store value in variable. When it is the case it can store some default as either zero or null depending on the data type. Here are basic primitive data type and their default values.

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

In java we can use "String" as a data type. It represents a group of characters like "hello", "world" ext..The simplest way to create String is by storing a group of characters into a String type variable as:

String str=" hello";

What is the difference between integer and int in java ?

In java integer is a class and int is primitive data type.

What is the difference between float and double?

float can represent 7 digits accurately after decimal point, where as double can represent up to 15 digits accurately after decimal point.

What is a Unicode system?

Unicode system is an encoding system standard that provides a unique number for every character, no matter what the platform, program, or language is. Unicode uses 2 bytes to represent a single character.

Non Primitive Data Types In Java

Also refred to as derived types. java supports non primitive data types classes, interfaces, and arrays ..ext and which will be covered in later topics.

4. What is abstract class? explain with example?

Ans: **Abstract class in Java**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

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

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
1. abstract class A{}
```

---

abstract method

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

Example abstract method

```
1. abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely..");}
6. public static void main(String args[]){
7.  Bike obj = new Honda4();
8.  obj.run();
9. }
10.}
```

Output: `running safely..`

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
```

```
6. void draw(){System.out.println("drawing rectangle");}
7. }
8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. class TestAbstraction1{
13. public static void main(String args[]){
14. Shape s=new Circle1();//In real scenario, object is provided through method e.g. getSh
    ape() method
15. s.draw();
16. }
17. }
```

Output: `drawing circle`

Points to Remember

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
2. An abstract class may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared abstract.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

5. What is Wrapping class? Explain with example?

Ans: A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

Example:

```
// Java program to demonstrate Autoboxing


import java.util.ArrayList;

class Autoboxing

{

    public static void main(String[] args)

    {

        char ch = 'a';;
```

```
        // Autoboxing- primitive to Character object conversion
        Character a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);

        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Output:

```
25
```

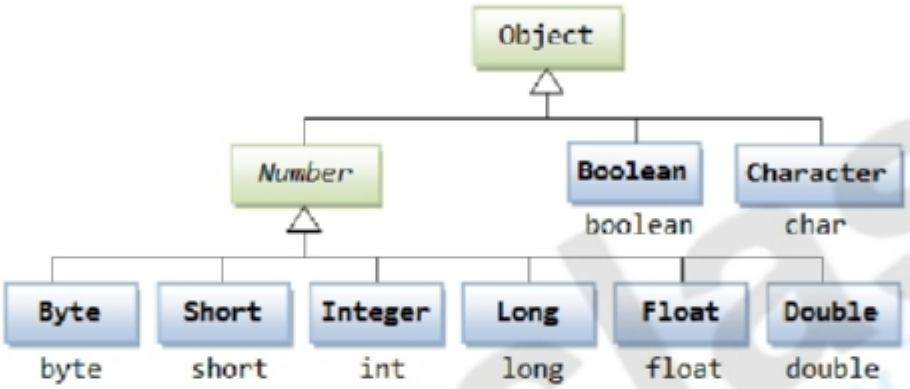
Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;

class Unboxing
{
    public static void main(String[] args)
    {
        Character ch = 'a';

        // unboxing - Character object to primitive conversion
        char a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);

        // unboxing because get method returns an Integer object
```

```
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Output:

```
24
```



Wrapper classes in Java

Implementation

```
// Java program to demonstrate Wrapping and UnWrapping
// in Java Classes
class WrappingUnwrapping
{
    public static void main(String args[])
    {
        //  byte data type
        byte a = 1;

        // wrapping around Byte object
        Byte byteobj = new Byte(a);

        // int data type
        int b = 10;
```

```
//wrapping around Integer object
Integer intobj = new Integer(b);

// float data type
float c = 18.6f;

// wrapping around Float object
Float floatobj = new Float(c);

// double data type
double d = 250.5;

// Wrapping around Double object
Double doubleobj = new Double(d);

// char data type
char e='a';

// wrapping around Character object
Character charobj=e;

//  printing the values from objects
System.out.println("Values of Wrapper objects (printing as objects)");
System.out.println("Byte object byteobj:  " + byteobj);
System.out.println("Integer object intobj:  " + intobj);
System.out.println("Float object floatobj:  " + floatobj);
System.out.println("Double object doubleobj:  " + doubleobj);
System.out.println("Character object charobj:  " + charobj);

// objects to data types (retrieving data types from objects)
// unwrapping objects to primitive data types
byte bv = byteobj;
int iv = intobj;
```

```
        float fv = floatobj;
        double dv = doubleobj;
        char cv = charobj;

        // printing the values from data types
        System.out.println("Unwrapped values (printing as data types)");
        System.out.println("byte value, bv: "+ bv);
        System.out.println("int value, iv: "+ iv);
        System.out.println("float value, fv: "+ fv);
        System.out.println("double value, dv: "+ dv);
        System.out.println("char value, cv: "+ cv);
    }
}
```

Output:

```
Values of Wrapper objects (printing as objects)
Byte object byteobj:  1
Integer object intobj:  10
Float object floatobj:  18.6
Double object doubleobj:  250.5
Character object charobj: a
Unwrapped values (printing as data types)
byte value, bv: 1
int value, iv: 10
float value, fv: 18.6
double value, dv: 250.5
char value, cv: a
```

6. Explain super method & super keyword?

Ans: The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables: This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

/* Driver program to test */
class Test
{
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Run on IDE

Output:

```
Maximum Speed: 120
```

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
```

```
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Run on IDE

Output:

```
This is student class
This is person class
```

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

3. **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, "super' can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}
```

```
/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Run on IDE

Output:

```
Person class Constructor

Student class Constructor
```

In the above example we have called the superclass constructor using keyword 'super' via subclass constructor.

Other Important points:

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*..

7. Explain the reference in java?

Ans: A reference is a value that refers to a another value. Its a convenient way to pass around objects, which are actually collections of many values. You need a single address that you can point to and say, "This is where all the object data lives." While C++ has a specific type of reference called a pointer, which points to a memory address, Java references are a higher level abstraction.

In Java, a reference is a type category. Reference types include: *classes*, *interfaces*, *arrays*, *enums*, and *annotations*. Java is a pass-by-value language, but references are themselves a kind of value. This means that when you pass a reference type to a method, that method has access to the data and can change it. But it does not have access to the original reference itself, and cannot change it to point elsewhere.

8. What is garbage collection? explain the use of GC Class? what is finalize method?

Ans:

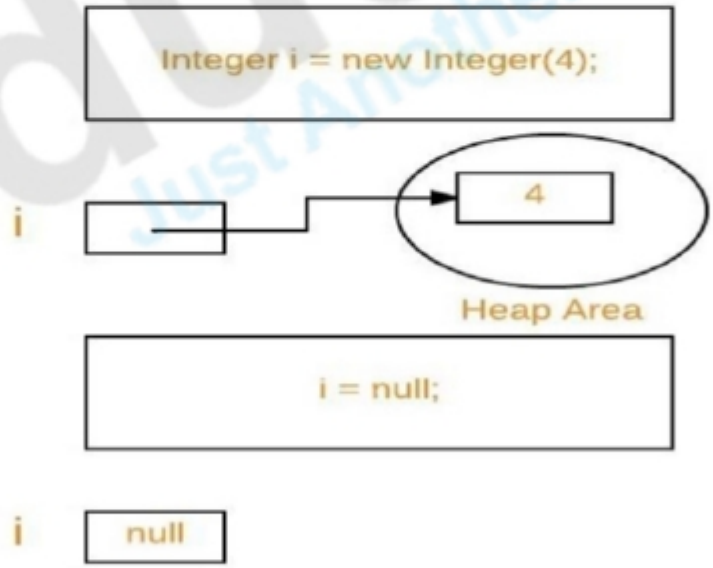
Garbage Collection in Java:

- In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.
- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
- Garbage collector is best example of Daemon thread as it is always running in background.
- Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

Important terms :

1. **Unreachable objects :** An object is said to be unreachable iff it doesn't contain any reference to it. Also note that objects which are part of island of isolation are also unreachable.

```
2. Integer i = new Integer(4);
3. // the new Integer object is reachable  via the reference in 'i'
4. i = null;
5. // the Integer object is no longer reachable.
```



6. **Eligibility for garbage collection :** An object is said to be eligible for GC(garbage collection) iff it is unreachable. In above image, after *i = null;* integer object 4 in heap area is eligible for garbage collection.

Ways to make an object eligible for GC

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
- There are generally four different ways to make an object eligible for garbage collection.
  1. Nullifying the reference variable
  2. Re-assigning the reference variable
  3. Object created inside method
  4. Island of Isolation

  All above ways with examples are discussed in separate article : How to make object eligible for garbage collection

Ways for requesting JVM to run Garbage Collector

- Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
- We can also request JVM to run Garbage Collector. There are two ways to do it :
  1. **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
  2. **Using *Runtime.getRuntime().gc()* method** : Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

```
// Java program to demonstrate requesting

// JVM to run Garbage Collector

public class Test

{

    public static void main(String[] args) throws InterruptedException

    {

        Test t1 = new Test();

        Test t2 = new Test();


        // Nullifying the reference variable

        t1 = null;


        // requesting JVM for running Garbage Collector

        System.gc();


        // Nullifying the reference variable

        t2 = null;


        // requesting JVM for running Garbage Collector

        Runtime.getRuntime().gc();
```

```
    }

    @Override
    // finalize method is called on object once
    // before garbage collecting it
    protected void finalize() throws Throwable
    {
        System.out.println("Garbage collector called");
        System.out.println("Object garbage collected : " + this);
    }
}
```

Output:

```
Garbage collector called
Object garbage collected : Test@46d08f12
Garbage collector called
Object garbage collected : Test@481779b8
```

- **Note :**
  1. There is no guarantee that any one of above two methods will definitely run Garbage Collector.
  2. The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()*

Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.
- *finalize()* method is present in Object class with following prototype.

```
protected void finalize() throws Throwable
```

Based on our requirement, we can override *finalize()* method for perform our cleanup activities like closing connection from database.

Note :

1. The *finalize()* method called by Garbage Collector not JVM. Although Garbage Collector is one of the module of JVM.

2. Object class *finalize()* method has empty implementation, thus it is recommended to override *finalize()* method to dispose of system resources or to perform other cleanup.
3. The *finalize()* method is never invoked more than once for any given object.
4. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored and finalization of that object terminates.

finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. Its the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

```
protected void finalize()
{
 //finalize-code
}
```

Some Important Points to Remember

1. `finalize()` method is defined in **java.lang.Object** class, therefore it is available to all the classes.
2. `finalize()` method is declare as **proctected** inside Object class.
3. `finalize()` method gets called only once by a Daemon thread named GC (Garbage Collector)thread.

---

gc() Method

gc() method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in **System** and **Runtime** class.

Example for gc() method

```
public class Test
{


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

```
        Test t = new Test();
        t=null;
        System.gc();
    }
    public void finalize()
    {
        System.out.println("Garbage Collected");
    }
}
```

Output :

```
Garbage Collected
```

9. Explain the inner class?

Ans:

Java inner class is defined inside the body of another class. Java inner class can be declared private, public, protected, or with default access whereas an outer class can have only public or default access.

Java Nested classes are divided into two types.

1. static nested class

If the nested class is static, then it's called static nested class. Static nested classes can access only static members of the outer class. Static nested class is same as any other top-level class and is nested for only packaging convenience.

Static class object can be created with following statement.

```
OuterClass.StaticNestedClass nestedObject =
      new OuterClass.StaticNestedClass();
```

java inner class

Any non-static nested class is known as inner class in java. Java inner class is associated with the object of the class and they can access all the variables and methods of the outer class.

Since inner classes are associated with instance, we can't have any static variables in them.

1. Object of java inner class are part of the outer class object and to create an instance of inner class, we first need to create instance of outer class.

   Java inner class can be instantiated like this;

```
OuterClass outerObject = new OuterClass();

OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
```

There are two special kinds of java inner classes.

1. local inner class

If a class is defined in a method body, it's known as local inner class.

Since local inner class is not associated with Object, we can't use private, public or protected access modifiers with it. The only allowed modifiers are abstract or final.

A local inner class can access all the members of the enclosing class and local final variables in the scope it's defined.

Local inner class can be defined as:

```
public void print() {
        //local inner class inside the method
        class Logger {
            String name;
        }
        //instantiate local inner class in the method to
use
        Logger logger = new Logger();
```

2. anonymous inner class

A local inner class without name is known as anonymous inner class. An anonymous class is defined and instantiated in a single statement.

Anonymous inner class always extend a class or implement an interface. Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class.

Anonymous inner classes are accessible only at the point where it is defined. It's a bit hard to define how to create anonymous inner class, we will see it's real time usage in test program below.

Here is a java class showing how to define java inner class, static nested class, local inner class and anonymous inner class.

`InnerClassTest.java`

```
package com.journaldev.nested;


import java.util.Arrays;

//nested classes can be used in import for easy instantiation

import com.journaldev.nested.OuterClass.InnerClass;

import com.journaldev.nested.OuterClass.StaticNestedClass;


public class InnerClassTest {


    public static void main(String[] args) {

        OuterClass outer = new OuterClass(1,2,3,4);


        //static nested classes example

        StaticNestedClass staticNestedClass = new
StaticNestedClass();

        StaticNestedClass staticNestedClass1 = new
StaticNestedClass();


        System.out.println(staticNestedClass.getName());
```

```
        staticNestedClass.d=10;

        System.out.println(staticNestedClass.d);

        System.out.println(staticNestedClass1.d);


        //inner class example

        InnerClass innerClass = outer.new InnerClass();

        System.out.println(innerClass.getName());

        System.out.println(innerClass);

        innerClass.setValues();

        System.out.println(innerClass);


        //calling method using local inner class

        outer.print("Outer");


        //calling method using anonymous inner class

System.out.println(Arrays.toString(outer.getFilesInDir("src/c
om/journaldev/nested", ".java")));



System.out.println(Arrays.toString(outer.getFilesInDir("bin/c
om/journaldev/nested", ".class")));

    }


}
```

Benefits of Java Inner Class

1. If a class is useful to only one class, it makes sense to keep it nested and together. It helps in packaging of the classes.
2. Java inner classes implements encapsulation. Note that inner classes can access outer class private members and at the same time we can hide inner class from outer world.

3. Keeping the small class within top-level classes places the code closer to where it is used and makes code more readable and maintainable.

10. What is package? Define a package syntax in java?

Ans: **Packages in Java** is a mechanism to encapsulate a group of classes, interfaces and sub packages. Many implementations of Java use a hierarchical file system to manage source and class files. It is easy to organize class files into packages. All we need to do is put related class files in the same directory, give the directory a name that relates to the purpose of the classes, and add a line to the top of each class file that declares the package name, which is the same as the directory name where they reside.

In java there are already many predefined packages that we use while programming.

For example: java.lang, java.io, java.util etc.
However one of the most useful feature of java is that we can define our own packages

Advantages of using a package

Before discussing how to use them Let see why we should use packages.

- Reusability: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- Easy to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to "name-space collisions". Packages are a way of avoiding "name-space collisions".

Types of package:
1) User defined package: The package we create is called user-defined package.
2) Built-in package: The already defined package like java.io.*, java.lang.* etc are known as built-in packages.

Defining a Package:
This statement should be used in the beginning of the program to include that program in that particular package.
`package <package name>;`

Example:

```
package tools;
public class Hammer {
   public void id ()
   {
       System.out.println ("Hammer");
   }
}
```

Points to remember:
1. At most one package declaration can appear in a source file.
2. The package declaration must be the first statement in the unit.

Naming conventions:
A global naming scheme has been proposed to use the internet domain names to uniquely identify packages. Companies use their reversed Internet domain name in their package names, like this:
`com.company.packageName`

How to Use a Package:

1. We can call it by its full name. For instance,

```
com.myPackage1.myPackage2 myNewClass = new com.myPackage1.myPackage2();
```

However this method is very time consuming. So normally we use the second method.

2. We use the "import" keyword to access packages. If you say import `com.myPackage1.myPackage2`, then every time we type "myPackage2", the compiler will understand that we mean `com.myPackage1.myPackage2`. So we can do:

```
import com.myPackage1.myPackage2;
class myClass {
   myPackage2 myNewClass= new myPackage2 ();
   …
   …
   …
}
```

There are two ways of importing a package:
Importing only a single member of the package

```
//here 'subclass' is a java file in myPackage2
import com.myPackage1.myPackage2.subClass;
class myClass {
   subClass myNewClass= new subClass();
   …
   …
   …
}
```

Importing all members of a package.

```
import com.myPackage1.*;
import java.sql.* ;
```

Also, when we use *, only the classes in the package referred are imported, and not the classes in the sub package.

The Java runtime automatically imports two entire packages by default: The java.lang package and the current package by default (the classes in the current folder/directory).

Points to remember:

1. Sometimes class name conflict may occur. For example:

There are two packages myPackage1 and myPackage2.Both of these packages contains a class with the same name, let it be `myClass.java`. Now both this packages are imported by some other class.

```
import myPackage1.*;
import myPackage2.*;
```

This will cause compiler error. To avoid these naming conflicts in such a situation, we have to be more specific and use the member's qualified name to indicate exactly which `myClass.java` class we want:

```
myPackage1.myClass myNewClass1 = new myPackage1.myClass ();
myPackage2.myClass myNewClass2 = new myPackage1.myClass ();
```

2. While creating a package, which needs some other packages to be imported, the package statement should be the first statement of the program, followed by the import statement.

Compiling packages in java:

The java compiler can place the byte codes in a directory that corresponds to the package declaration of the compilation unit. The java byte code for all the classes(and interfaces) specified in the source files `myClass1.java` and `myClass2.java` will be placed in the directory named `myPackage1/myPackage2` , as these sources have the following package declaration

```
package  myPackage1.myPackage2;
```
The absolute path of the myPackage1/myPackage2 directory is specified by using the –d (destination directory) option when compiling with the javac compiler.

Assume that the current directory is /packages/project and all the source files are to be found here,the command,

```
javac -d .file1.java file2.java
```
Issued in the working directory will create ./ myPackage1/myPackage2(and any sub directories required) under the current directory, and place the java byte code for all the classes(and interfaces) in the directories corresponding to the package names. The dot (.) after the –d option denotes the current directory. Without the –d option, the default behaviors of the java compiler is to place all the class files in the current directory rather than the appropriate sub directories.

How do we run the program?
Since the current directory is /packages/project and we want to run file1.java,the fully qualified name of the file1 class must be specified in the java command,

```
java myPackage1.myPackage2.file1
```
Classpath :
It is a environmental variable, which contains the path for the default-working directory (.).
The specific location that java compiler will consider, as the root of any package hierarchy is, controlled by Classpath

Access Specifiers

- private: accessible only in the class
- no modifier: so-called "package" access — accessible only in the same package
- protected: accessible (inherited) by subclasses, and accessible by code in same package
- public: accessible anywhere the class is accessible, and inherited by subclasses

Notice that private protected is not syntactically legal.

Access By	private	package	protected	public
the class itself	yes	yes	yes	yes
a subclass in same package	no	yes	yes	yes
non-subclass in same package	no	yes	yes	yes
a subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

11. Define Interface & implement it, give example?

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

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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

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

However, an interface is different from a class in several ways, including –

- You cannot instantiate an interface.

- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

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

Example

Following is an example of an interface –

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\
}
```

Interfaces have the following properties –

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

Example

```
/* File name : Animal.java */
interface Animal {
   public void eat();
```

```
   public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

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

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

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

   public int noOfLegs() {
      return 0;
   }

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

This will produce the following result –

Output

```
Mammal eats
Mammal travels
```

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

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

When implementation interfaces, there are several rules –

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

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}
```

```
// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as –

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener

interface in the java.awt.event package extended java.util.EventListener, which is defined as –

Example

```
package java.util;

public interface EventListener

{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces –

Creates a common parent – As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class – This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

12. How interface is different from abstract class?

Ans:

The main difference between abstract class and interface is driven by abstract methods. An abstract class may or may not have abstract methods, while all methods declared inside an interface must be abstract (an abstract method is a method that is declared without an implementation -- without braces, and followed by a semicolon). Abstract classes and interfaces have their own application uses, and cannot be used interchangeably. Here, a tabular comparison is presented between abstract classes and interfaces.

Table 1: Differences between abstract class and interface in Java

Abstract Class	Interface
An abstract class is a class that is declared `abstract`. It may or may not have abstract methods.	An interface in Java is implicitly abstract and

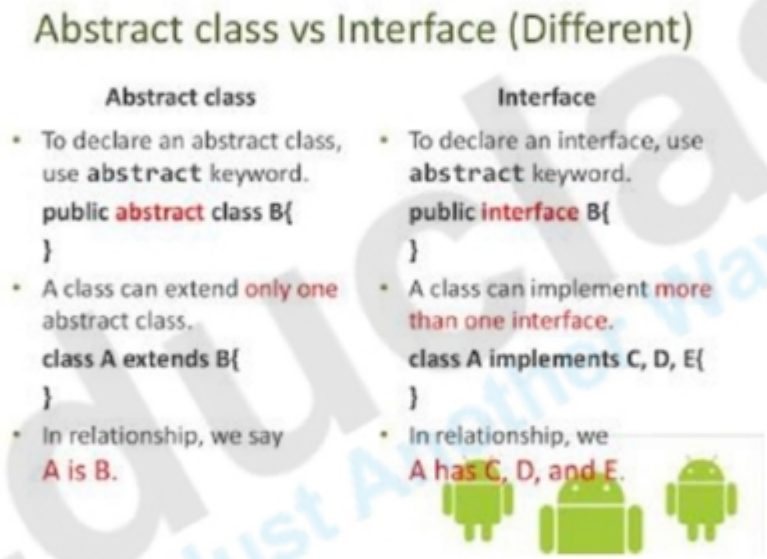
	adding that modifier is considered redundant and makes no difference.
An abstract class can have both abstract and non-abstract methods. Non-abstract methods in an abstract class are used to implement default behavior.	Methods declared in an interface are by default abstract and public; therefore, they cannot have implementation. It means to say that an interface cannot contain non-abstract methods.
Non-abstract methods of an abstract class can be declared `static` because to call a non-abstract static method there is no need to create an instance of the class. Of course, a method cannot be declared `abstract` and `static` both in abstract class.	On the other hand, interface methods must not be static because all the methods of an interface are implicitly abstract, and an abstract method can never be declared static.
An abstract class can declare instance variables as well, along with constants.	All variables defined in an interface must be `public`, `static`, and `final` - in other words, interfaces can declare only constants, not instance variables.
Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.	Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
Members of an abstract class can be `public`, `private` or `protected` depending upon the required visibility level.	All members of an interface are by default `public`.
An abstract class is extended using keyword `extends`.	An interface is implemented using keyword `implements`.
An abstract class can extend another class and implement one or more interfaces.	An interface can extend one or more other interfaces. An interface cannot extend anything but another interface.

When to Use Abstract Class Instead of Interface

When to use abstract class: In some situations, the superclass does not directly relate to a "thing" in the real world, and because of this we do not instantiate the superclass. For example, all four and two-wheelers are vehicles but there is no real world object as "vehicle" it is a conceptual entity, which has no real world existence. In that case we can declare an abstract class called "Vehicle" and place all common attributes and functionalities inside that class. Later we can create a subclass "Car" which inherits all common attributes and methods from "Vehicle". Now an object of class "Car" can be created and it can be assigned to the reference of "Vehicle".

In above explained situation we should use abstract class rather than interface because this subclass-superclass relationship is genuinely an "is a" relationship.

When to use interface: On the other hand we interfaces should be used when a class promises some behaviors to provide. Interfaces form a contract between the class and the outside world. Moreover, iterface should be used when the subclass needs to inherit from another class.



13. What is generic? Explain generic classes & generic methods?

Ans:

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –

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

Example

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

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

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

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

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

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

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

This will produce the following result –

Output

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

Array doubleArray contains:
1.1 2.2 3.3 4.4

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

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects –

```
public class MaximumTest {
   // determines the largest of three Comparable objects

   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

This will produce the following result –

Output

```
Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

Following example illustrates how we can define a generic class –

```
public class Box<T> {
   private T t;
   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }
   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}
```

This will produce the following result –

Output

```
Integer Value :10
String Value :Hello World
```

14. Explain the Exception handling? Explain different type of exception handling?

Ans:

What is an exception?

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn't continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

When an exception can occur?
Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

Reasons for Exceptions
There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

Difference between 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 –

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.
- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Why to handle exception?
If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex:-Take a look at the below system generated exception

An exception generated by the system is given below

```
 Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
 ExceptionDemo : The class name
 main : The method name
 ExceptionDemo.java : The filename
 java:5 : Line number
```

For a novice user the above message won't be easy to understand. In order to let them know that what went wrong we use exception handling in java program. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Types of exceptions

There are two types of exceptions

1)Checked exceptions
2)Unchecked exceptions

Below is a brief about each however if you want a detailed tutorial with examples then you can refer Checked and Unchecked exceptions in Java.

Checked exceptions
All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, it will give compilation error.

Examples of Checked Exceptions :-

ClassNotFoundException
IllegalAccessException
NoSuchFieldException
EOFException etc.

Unchecked Exceptions

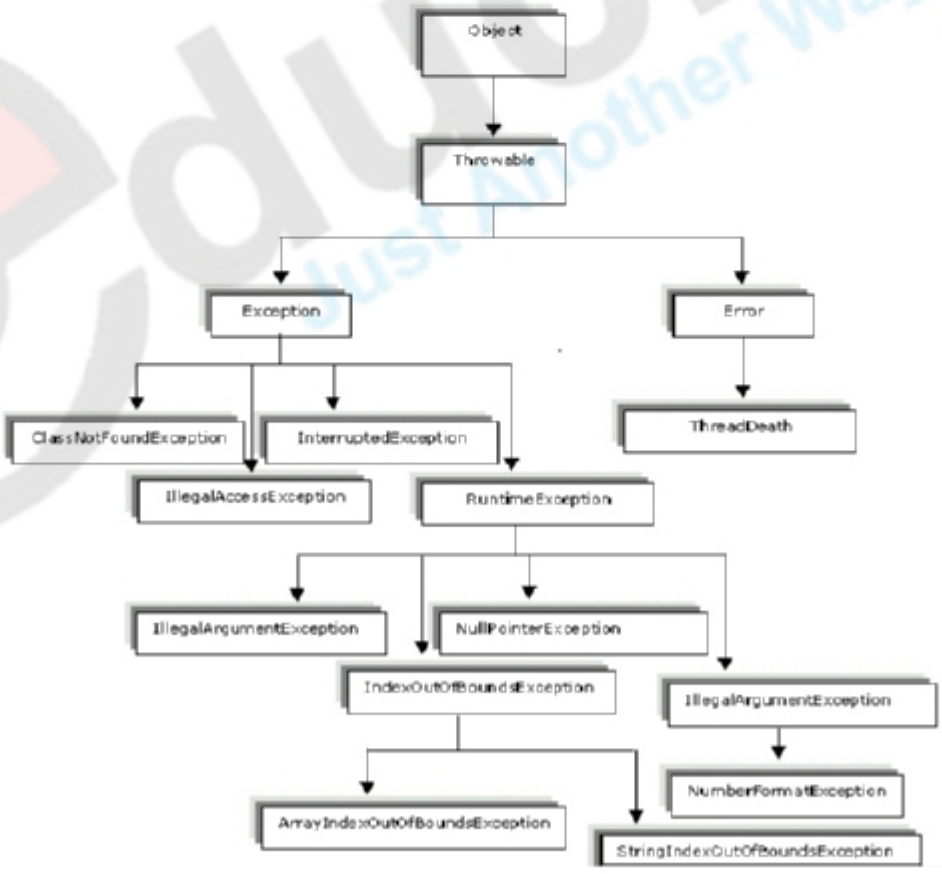
Runtime Exceptions are also known as Unchecked Exceptions as the compiler do not check whether the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.

These exceptions need not be included in any method's throws list because compiler does not check to see if a method handles or throws these exceptions.

Examples of Unchecked Exceptions:-

ArithmeticException
ArrayIndexOutOfBoundsException
NullPointerException
NegativeArraySizeException etc.

Exception hierarchy



Exception handling in Java

1. Try-catch in Java

The try block contains a block of program statements within which an exception might occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must followed by a Catch block or Finally block or both.

Syntax of try block

```
try{
   //statements that may cause an exception
}
```

What is Catch Block?

A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
}
```

Flow of try catch block

1. If an exception occurs in try block then the control of execution is passed to the catch block from try block. The exception is caught up by the corresponding catch block. A single try block can have multiple catch statements associated with it, but each catch block can be defined for only one exception class. The program can also contain nested try-catch-finally blocks.
2. After the execution of all the try blocks, the code inside the finally block executes. 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.

An example of Try catch in Java

```
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.
```

Multiple catch blocks in Java

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.

Example of Multiple catch blocks

```
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.

2. Nested Try Catch:

3. 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.

4. If no catch block matches, then the java run-time system will handle the exception. Lets see the syntax first then we will discuss this with an example.

5. Syntax of Nested try Catch

```
....
//Main try block
try
{
   statement 1;
   statement 2;
   //try-catch block inside another try block
   try
   {
      statement 3;
      statement 4;
   }
   catch(Exception e1)
   {
```

```
        //Exception Message
    }
    //try-catch block inside another try block
    try
    {
        statement 5;
        statement 6;
    }
    catch(Exception e2)
    {
        //Exception Message
    }
}
catch(Exception e3) //Catch of Main(parent) try block
{
      //Exception Message
}
....
```

38. Nested try catch example - explanation

```
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..
```

The above example shows Nested try catch use in Java. You can see that there are two try-catch block inside main try block's body. I've marked them as `block 1` and `block 2` in above example.

Block1: I have divided an integer by zero and it caused an arithmetic exception however the catch of block1 is handling arithmetic exception so `"Exception: e1"`got printed.

Block2: In block2 also, `ArithmeticException` occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump back to Main try-catch(parent) body. Since catch of parent try block is handling this exception that's why "Inside parent try catch block" got printed as output.

Parent try Catch block: Since all the exception handled properly so program control didn't get terminated at any point and at last "Next statement.." came as output.

Note: 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.

Consider this example:

Here we have deep (two level) nesting which means we have a try-catch block inside a child try block. To make you understand better I have given the names to each try block in comments like try-block2 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the try-block3. Refer the explanation which is given at the end of this code.

```
class NestingDemo{
   public static void main(String args[]){
      //main try-block
      try{
          //try-block2
        try{
```

```
            //try-block3
            try{
                  int arr[]= {1,2,3,4};
                  /* I'm trying to display the value of
                   * an element which doesn't exist. The
                   * code should throw an exception
                   */
                  System.out.println(arr[10]);
              }catch(ArithmeticException e){
                  System.out.print("Arithmetic Exception");
                  System.out.println(" handled in try-block3");
              }
          }
          catch(ArithmeticException e){
             System.out.print("Arithmetic Exception");
             System.out.println(" handled in try-block2");
          }
      }
      catch(ArithmeticException e3){
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in main try-block");
      }
      catch(ArrayIndexOutOfBoundsException e4){
          System.out.print("ArrayIndexOutOfBoundsException");
          System.out.println(" handled in main try-block");
      }
      catch(Exception e5){
          System.out.print("Exception");
          System.out.println(" handled in main try-block");
       }
   }
}
ArrayIndexOutOfBoundsException handled in main try-block
```

As you can see that the ArrayIndexOutOfBoundsException has occurred in the grand child try-block3. Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in try-block2. Since the try-block2 is also not handling that exception, the control got transferred to the main grand parent try-block where it found the appropriate catch block for exception. This is how the **routing of exception is done in nested structure**.

3. Checked and unchecked exceptions:

There are two types of exceptions: checked exceptions and unchecked exceptions. In this tutorial we will learn both of them with the help of examples. The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time.

Lets understand this with this **example**: In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where an checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

```
4. import java.io.*;
5. class Example {
6.     public static void main(String args[])
7.     {
8.    FileInputStream fis = null;
9.    /*This constructor FileInputStream(File filename)
10.    * throws FileNotFoundException which is a checked
11.    * exception*/
12.         fis = new FileInputStream("B:/myfile.txt");
13.   int k;
14.
15.   /*Method read() of FileInputStream class also throws
16.    * a checked exception: IOException*/
17.   while(( k = fis.read() ) != -1)
18.   {
19.           System.out.print((char)k);
20.   }
21.
22.   /*The method close() closes the file input stream
23.    * It throws IOException*/
24.   fis.close();
25.    }
26.}
```

27. Output:

```
28.Exception in thread "main" java.lang.Error: Unresolved compilation
   problems:
29.Unhandled exception type FileNotFoundException
30.Unhandled exception type IOException
31.Unhandled exception type IOException
```

32. **Why this compilation error?** As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn't handled/declared the exceptions, our program gave the compilation error.

How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.
As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. Th reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too like this `public static void main(String args[]) throws IOException, FileNotFoundException`.

```
import java.io.*;
class Example {
   public static void main(String args[]) throws IOException
   {
      FileInputStream fis = null;
      fis = new FileInputStream("B:/myfile.txt");
      int k;

      while(( k = fis.read() ) != -1)
      {
     System.out.print((char)k);
      }
      fis.close();
   }
}
```

File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.

The above approach is not good at all. It is not a best exception handling practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
  FileInputStream fis = null;
  try{
      fis = new FileInputStream("B:/myfile.txt");
  }catch(FileNotFoundException fnfe){
            System.out.println("The specified file is not " +
                  "present at the given path");
   }
  int k;
  try{
```

```
61.       while(( k = fis.read() ) != -1)
62.       {
63.           System.out.print((char)k);
64.       }
65.       fis.close();
66.   }catch(IOException ioe){
67.       System.out.println("I/O error occurred: "+ioe);
68.    }
69.    }
70.}
```

1.Finally block in Java:

1. A finally statement must be associated with a try statement. It identifies a block of statements that needs to be executed regardless of whether or not an exception occurs within the try block.

2. After all other try-catch processing is complete, the code inside the finally block executes. 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 parts of the block.

3. In normal execution the finally block is executed after try block. When any exception occurs first the catch block is executed and then finally block is executed.

4. An exception in the finally block, exactly behaves like any other exception.

5. The code present in the **finally block** executes even if the try or catch block contains control transfer statements like return, break or continue.

To understand above concepts better refer the below examples.

Syntax of Finally block

```
try
{
    //statements that may cause an exception
}
finally
{
   //statements to be executed
}
```

Cases when the finally block doesn't execute

The circumstances that prevent execution of the code in a finally block are:
– The death of a Thread
– Using of the System. exit() method.
– Due to an exception arising in the finally block.

Finally block and Return statement

Finally block executes even if there is a return statement in try-catch block. PFB the example –

```
class JavaFinally
{
   public static void main(String args[])
   {
      System.out.println(JavaFinally.myMethod());
   }
   public static int myMethod()
   {
      try {
        return 112;
      }
      finally {
        System.out.println("This is Finally block");
        System.out.println("Finally block ran even after return statement");
      }
   }
}
```

Output of above program:

```
This is Finally block
Finally block ran even after return statement
112
```

Finally and Close()

Close() is generally used to close all the open streams in one go. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

E.g.

```
....
try{
    OutputStream osf = new FileOutputStream( "filename" );
    OutputStream osb = new BufferedOutputStream(opf);
    ObjectOutput op = new ObjectOutputStream(osb);
    try{
       output.writeObject(writableObject);
    }
```

```
    finally{
       op.close();
    }
}
catch(IOException e1){
     System.out.println(e1);
}
...
```

Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

```
...
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
       try {
         in.close();
       }catch (IOException exp) {
           System.out.println(exp);
        }
    }
}
...
```

Finally block and System.exit()

System.exit() statement behaves differently than **return statement**. Unlike return statement whenever System.exit() gets called in try block then **Finally block** doesn't get executed. Refer the below example to understand it better –

```
....
try {
   //try block
   System.out.println("Inside try block");
   System.exit(0)
}
catch (Exception exp) {
   System.out.println(exp);
}
finally {
   System.out.println("Java finally block");
}
....
```

In the above example if the **System.exit(0)** gets called without any exception then finally won't execute. However if any exception occurs while calling **System.exit(0)** then finally block will be executed.

Handling try-catch-finally block

- Either a try statement should be associated with a catch block or with finally.
- Since catch performs exception handling and finally performs the cleanup, the best approach is to merge both of them.

Syntax:

```
try
{
     //statements that may cause an exception
}
catch (…)
{
     //error handling code
}
finally
{
    //statements to be executed
}
```

Examples of Try catch finally blocks

Example 1: Below example illustrates finally block when no exception occurs in try block

```
class Example1{
  public static void main(String args[]){
    try{
       System.out.println("First statement of try block");
       int num=45/3;
       System.out.println(num);
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("ArrayIndexOutOfBoundsException");
    }
    finally{
       System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
  }
}
```

Output:

```
finally block
Out of try-catch-finally block
```

Example 2:Below example illustrates finally block execution when exception occurs in try block but doesn't get handled in catch block.

```
class Example2{
   public static void main(String args[]){
     try{
```

```
        System.out.println("First statement of try block");
        int num=45/0;
        System.out.println(num);
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("ArrayIndexOutOfBoundsException");
     }
     finally{
        System.out.println("finally block");
     }
     System.out.println("Out of try-catch-finally block");
   }
}
```

Output:

```
First statement of try block
finally block
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at beginnersbook.com.Example2.main(Details.java:6)
```

Example 3:Below example illustrates execution of finally, when exception occurs in try block and handled in catch block.

```
class Example3{
   public static void main(String args[]){
      try{
         System.out.println("First statement of try block");
         int num=45/0;
         System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("ArithmeticException");
      }
      finally{
         System.out.println("finally block");
      }
      System.out.println("Out of try-catch-finally block");
   }
}
```

Output:

```
First statement of try block
ArithmeticException
finally block
Out of try-catch-finally block
```

5. try-catch-finally

Flow of control in try/catch blocks:

when exception doesn't occur:
When the code which is present in try block's body doesn't throw any

exceptionthen first, the body of try block executes and then the code after catch blocks. In this case catch block never runs as they are meant to be run when an exception occurs. For example-

```
.....
int x = 10;
int y = 10;
try{
   int num= x/y;
   System.out.println("next-statement: Inside try block");
}catch(Exception ex)
 {
    System.out.println("Exception");
 }
System.out.println("next-statement: Outside of try-catch");
...
```

Output:

```
next-statement: Inside try block
next-statement: Outside of try-catch
```

In the above example exception didn't occur in try block so catch block didn't run.

when exception occurs:

First have a look at the below example and then we will discuss it –

```
int x = 0;
int y = 10;
try{
   int num= y/x;
   System.out.println("next-statement: Inside try block");
}catch(Exception ex)
 {
    System.out.println("Exception Occurred");
 }
System.out.println("next-statement: Outside of try-catch");
...
```

Output:

```
Exception Occurred
next-statement: Outside of try-catch
```

Point to note in above example: There are two statements present inside try block. Since exception occurred because of first statement, the second statement didn't execute. Hence we can conclude that if an exception occurs then the rest of the try block doesn't execute and control passes to catch block.

Flow of control in try/catch/finally blocks:

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 and after that rest of the program.
2. 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**).
3. 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.**

Consider the below example to understand above mentioned points:

```
class TestExceptions {
   static void myMethod(int testnum) throws Exception {
      System.out.println ("start - myMethod");
      if (testnum == 12)
         throw new Exception();
      System.out.println("end - myMethod");
      return;
   }
   public static void main(String  args[]) {
      int testnum = 12;
      try {
         System.out.println("try - first statement");
         myMethod(testnum);
         System.out.println("try - last statement");
      }
      catch ( Exception ex) {
         System.out.println("An Exception");
      }
      finally {
         System. out. println( "finally") ;
      }
      System.out.println("Out of try/catch/finally - statement");
   }
}
```

Output:

```
try - first statement
start - myMethod
An Exception
finally
Out of try/catch/finally - statement
```

6. Throw exception in Java:

Do you know that a programmer can create a new exception and throw it explicitly? These exceptions are known as user-defined exceptions. In order

to throw user defined exceptions, throw keyword is being used. In this tutorial, we will see how to create a new exception and throw it in a program using **throw keyword.**

You can also throw an already defined exception like `ArithmeticException`, `IOException` etc.

Syntax of throw statement

```
throw AnyThrowableInstance;
```

Example:

```
//A void method
public void sample()
{
   //Statements
   //if (somethingWrong) then
   IOException e = new IOException();
   throw e;
   //More Statements
 }
```

Note:

- A call to the above mentioned sample method should be always placed in a try block as it is throwing a checked exception – `IOException`. This is how it the call to above method should be done:

```
MyClass obj =  new MyClass();
try{
      obj.sample();
}catch(IOException ioe)
 {
      //Your error Message here
      System.out.println(ioe);
  }
```

- Exceptions in java are compulsorily of type Throwable. If you attempt to throw an object that is not throwable, the compiler refuses to compile your program and it would show a compilation error.

Flow of execution while throwing an exception using throw keyword

Whenever a throw statement is encountered in a program the next statement doesn't execute. Control immediately transferred to catch block to see if the thrown exception is handled there. If the exception is not handled there then next catch block is being checked for exception and so on. If none of the catch block is handling the thrown exception then a system generated exception message is being populated on screen, same what we get for un-

handled exceptions.
E.g.

```
class ThrowDemo{
   public static void main(String args[]){
      try{
           char array[] = {'a','b','g','j'};
           /*I'm displaying the value which does not
            * exist so this should throw an exception
            */
           System.out.println(array[78]);
      }catch(ArithmeticException e){
            System.out.println("Arithmetic Exception!!");
       }
   }
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
78 at beginnersbook.com.ThrowDemo.main(Details.java:9)
```

Since the exception thrown was not handled in the catch blocks the system generated exception message got displayed for that particular exception.

Few examples of throw exception in Java

Example 1: How to throw your own exception explicitly using throw keyword

```
package beginnersbook.com;
class MyOwnException extends Exception {
   public MyOwnException(String msg){
      super(msg);
   }
}

class EmployeeTest {
   static void  employeeAge(int age) throws MyOwnException{
      if(age < 0)
         throw new MyOwnException("Age can't be less than zero");
      else
         System.out.println("Input is valid!!");
   }
   public static void main(String[] args) {
       try {
            employeeAge(-2);
       }
       catch (MyOwnException e) {
            e.printStackTrace();
       }
   }
}
```

Output:

```
beginnersbook.com.MyOwnException: Age can't be less than zero
```

Points to Note: Method call should be in try block as it is throwing an exception.

Example2: How to throw an already defined exception using throw keyword

```
package beginnersbook.com;
class Exception2{
   static int sum(int num1, int num2){
      if (num1 == 0)
         throw new ArithmeticException("First parameter is not valid");
      else
         System.out.println("Both parameters are correct!!");
      return num1+num2;
   }
   public static void main(String args[]){
      int res=sum(0,12);
      System.out.println(res);
      System.out.println("Continue Next statements");
   }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException: First parameter is not
```

7. Throws in Java:

Use of throws keyword in Java

1. The throws keyword is used in method declaration, in order to explicitly specify the exceptions that a particular method might throw. When a method declaration has one or more exceptions defined using throws clause then the method-call must handle all the defined exceptions.
2. When defining a method you must include a throws clause to declare those exceptions that might be thrown but doesn't get caught in the method.
3. If a method is using throws clause along with few exceptions then this implicitly tells other methods that – " If you call me, you must handle these exceptions that I throw".

Syntax of Throws in java:

```
void MethodName() throws ExceptionName{
    Statement1
    ...
    ...
}
```

E.g:

```
public void sample() throws IOException{
     //Statements
     //if (somethingWrong)
     IOException e = new IOException();
     throw e;
     //More Statements
}
```

Note: In case a method throws more than one exception, all of them should be listed in throws clause. PFB the example to understand the same.

```
public void sample() throws IOException, SQLException
{
    //Statements
}
```

The above method has both IOException and SQLException listed in throws clause. There can be any number of exceptions defined using throws clause.

Complete Example of Java throws Clause

```
class Demo
{
   static void throwMethod() throws NullPointerException
   {
       System.out.println ("Inside throwMethod");
       throw new NullPointerException ("Demo");
   }
   public static void main(String args[])
   {
       try
       {
          throwMethod();
       }
       catch (NullPointerException exp)
       {
          System.out.println ("The exception get caught" +exp);
       }
    }
}
```

The output of the above program is:

```
Inside throwMethod
The exception get caught java.lang.IllegalAccessException: Demo
```

Difference between throw and throws in java

1. **Throws clause** in used to declare an exception and **thow** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance variable and **throws** is followed by exception class names.

3. The keyword **throw** is used inside method body to invoke an exception and **throws clause** is used in method declaration (signature).

for e.g.

Throw:

```
....
static{
try {
throw new Exception("Something went wrong!!");
} catch (Exception exp) {
System.out.println("Error: "+exp.getMessage());
}
}
....
```

Throws:

```
public void sample() throws ArithmeticException{
 //Statements

.....

 //if (Condition : There is an error)
ArithmeticException exp = new ArithmeticException();
 throw exp;
...
}
```

4. By using **Throw keyword** in java you cannot throw more than one exception but using **throws** you can declare multiple exceptions. PFB the examples.

for e.g.

Throw:

```
throw new ArithmeticException("An integer should not be divided by zero!!")
```

```
throw new IOException("Connection failed!!")
```

Throws:

```
throws IOException, ArithmeticException, NullPointerException,
ArrayIndexOutOfBoundsException
```

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.

What is Assertion?

Ans:

The **assert** keyword is used in `assert` statement which is a feature of the Java programming language since Java 1.4. Assertion enables developers to test assumptions in their programs as a way to defect and fix bugs.

Syntax of assert statement

Syntax of an assert statement is as follow (short version):

assert *expression1*;

or (full version):

assert *expression1* : *expression2*;

Where:

- *expression1* must be a `boolean` expression.
- *expression2* must return a value (must not return void).

The `assert` statement is working as follows:

-
  - If assertion is enabled, then the `assert` statement will be evaluated. Otherwise, it does not get executed.
  - If *expression1* is evaluated to `false`, an `AssertionError` error is thrown which causes the program stops immediately. And depending on existence of *expression2*:

-
  -
    - If *expression2* does not exist, then the `AssertionError` is thrown with no detail error message.
    - If *expression2* does exist, then a String representation of *expression2*'s return value is used as detail error message.

-
  - If *expression1* is evaluate to `true`, then the program continues normally.

Enable assertion
By default, assertion is disabled at runtime. To enable assertion, specify the switch `-enableassertions` or `-ea` at command line of `java` program. For example, to enable assertion for the program called `CarManager`:

```
java -enableassertions CarManager
```

or this for short:

```
java -ea CarManager
```

Assertion examples

The following simple program illustrates the short version of `assert` statement:

```
1   public class AssertionExample {
2       public static void main(String[] args) {
3           // get a number in the first argument
4           int number = Integer.parseInt(args[0]);
5
6           assert number <= 10; // stops if number > 10
7
8           System.out.println("Pass");
9
10      }
11  }
```

When running the program above with this command:

```
java -ea AssertionExample 15
```

A `java.lang.AssertionError` error will be thrown:

```
Exception in thread "main" java.lang.AssertionError

        at
AssertionExample.main(AssertionExample.java:6)
```

But the program will continue and print out “Pass” if we pass a number less than 10, in this command:

```
java -ea AssertionExample 8
```

And the following example is using the full version of `assert` statement:

```
1   public class AssertionExample2 {
2       public static void main(String[] args) {
3
4           int argCount = args.length;
5
6           assert argCount == 5 : "The number of arguments must be 5";
7
8           System.out.println("OK");
9
10      }
11  }
```

When running the program above with this command:

```
java -ea AssertionExample2 1 2 3 4
```

it will throw this error:

```
Exception in thread "main" java.lang.AssertionError:
The number of arguments must be 5

        at
AssertionExample2.main(AssertionExample2.java:6)
```

Generally, assertion is enabled during development time to defect and fix bugs, and is disabled at deployment or production to increase performance.

What is Annotations?

Java Annotations allow us to add metadata information into our source code, although they are not a part of the program itself. Annotations were added to the java from JDK 5. Annotation has no direct effect on the operation of the code they annotate (i.e. it does not affect the execution of the program).

In this tutorial we are going to cover following topics: Usage of annotations, how to apply annotations, what predefined annotation types are available in the Java and how to create custom annotations.

What's the use of Annotations?

1) Instructions to the compiler: There are three built-in annotations available in Java (`@Deprecated`, `@Override` & `@SuppressWarnings`) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.

2) Compile-time instructors: Annotations can provide compile-time instructions to the compiler that can be further used by sofware build tools for generating code, XML files etc.

3) Runtime instructions: We can define annotations to be available at runtime which we can access using java reflection and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

Annotations basics

An annotation always starts with the symbol `@` followed by the annotation name. The symbol `@` indicates to the compiler that this is an annotation.

For e.g. `@Override`
Here @ symbol represents that this is an annotation and the Override is the name of this annotation.

Where we can use annotations?
Annotations can be applied to the classes, interfaces, methods and fields. For example the below annotation is being applied to the method.

```
@Override
```

```
void myMethod() {
    //Do something
}
```
What this annotation is exactly doing here is explained in the next section but to be brief it is instructing compiler that `myMethod()` is a overriding method which is overriding the method (myMethod()) of super class.

Built-in Annotations in Java

Java has three built-in annotations:

- @Override
- @Deprecated
- @SuppressWarnings

1) @Override:

While overriding a method in the child class, we should use this annotation to mark that method. This makes code readable and avoid maintenance issues, such as: while changing the method signature of parent class, you must change the signature in child classes (where this annotation is being used) otherwise compiler would throw compilation error. This is difficult to trace when you haven't used this annotation.

Example:

```
public class MyParentClass {

    public void justaMethod() {
        System.out.println("Parent class method");
    }
}


public class MyChildClass extends MyParentClass {

    @Override
    public void justaMethod() {
        System.out.println("Child class method");
    }
}
```
I believe the example is self explanatory. To read more about this annotation, refer this article: @Override built-in annotation.

2) @Deprecated

@Deprecated annotation indicates that the marked element (class, method or field) is deprecated and should no longer be used. The compiler generates a

warning whenever a program uses a method, class, or field that has already been marked with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. Make a note of case difference with @Deprecated and @deprecated. @deprecated is used for documentation purpose.

Example:

```
/**
 * @deprecated
 * reason for why it was deprecated
 */
@Deprecated
public void anyMethodHere(){
    // Do something
}
```

Now, whenever any program would use this method, the compiler would generate a warning. To read more about this annotation, refer this article: Java – @Deprecated annotation.

3) @SuppressWarnings

This annotation instructs compiler to ignore specific warnings. For example in the below code, I am calling a deprecated method (lets assume that the method deprecatedMethod() is marked with @Deprecated annotation) so the compiler should generate a warning, however I am using @@SuppressWarnings annotation that would suppress that deprecation warning.

```
@SuppressWarnings("deprecation")
    void myMethod() {
        myObject.deprecatedMethod();
}
```

Creating Custom Annotations

- Annotations are created by using @interface, followed by annotation name as shown in the below example.
- An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.
- All annotations extends java.lang.annotation.Annotation interface. Annotations cannot include any extends clause.

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{
    int studentAge() default 18;
    String studentName();
    String stuAddress();
    String stuStream() default "CSE";
}
```

Note: All the elements that have default values set while creating annotations can be skipped while using annotation. For example if I'm applying the above annotation to a class then I would do it like this:

```
@MyCustomAnnotation(
    studentName="Chaitanya",
    stuAddress="Agra, India"
)
public class MyClass {
...
}
```

As you can see, we have not given any value to the `studentAge` and `stuStream`elements as it is optional to set the values of these elements (default values already been set in Annotation definition, but if you want you can assign new value while using annotation just the same way as we did for other elements). However we have to provide the values of other elements (the elements that do not have default values set) while using annotation.

Note: We can also have array elements in an annotation. This is how we can use them:
Annotation definition:

```
@interface MyCustomAnnotation {
    int      count();
    String[] books();
}
```

Usage:

```
@MyCustomAnnotation(
    count=3,
    books={"C++", "Java"}
)
public class MyClass {

}
```

Lets back to the topic again: In the custom annotation example we have used these four annotations: `@Documented`, `@Target`, `@Inherited` & `@Retention`. Lets discuss them in detail.

@Documented

@Documented annotation indicates that elements using this annotation should be documented by JavaDoc. For example:

```
java.lang.annotation.Documented
@Documented
public @interface MyCustomAnnotation {
  //Annotation body
}
@MyCustomAnnotation
public class MyClass {
     //Class body
}
```

While generating the javadoc for class `MyClass`, the annotation `@MyCustomAnnotation` would be included in that.

@Target

It specifies where we can use the annotation. For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
public @interface MyCustomAnnotation {

}
public class MyClass {
   @MyCustomAnnotation
   public void myMethod()
   {
       //Doing something
   }
}
```

Note: 1) If you do not define any Target type that means annotation can be applied to any element.
2) Apart from ElementType.METHOD, an annotation can have following possible Target values.
ElementType.METHOD
ElementType.PACKAGE
ElementType.PARAMETER
ElementType.TYPE
ElementType.ANNOTATION_TYPE
ElementType.CONSTRUCTOR
ElementType.LOCAL_VARIABLE
ElementType.FIELD

@Inherited

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

```
java.lang.annotation.Inherited

@Inherited
public @interface MyCustomAnnotation {

}
@MyCustomAnnotation
public class MyParentClass {
  ...
}
public class MyChildClass extends MyParentClass {
   ...
}
```

Here the class `MyParentClass` is using annotation `@MyCustomAnnotation`which is marked with @inherited annotation. It means the sub class `MyChildClass` inherits the `@MyCustomAnnotation`.

@Retention

It indicates how long annotations with the annotated type are to be retained.

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface MyCustomAnnotation {

}
```

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:

`RetentionPolicy.RUNTIME`: The annotation should be available at runtime, for inspection via java reflection.

`RetentionPolicy.CLASS`: The annotation would be in the .class file but it would not be available at runtime.

`RetentionPolicy.SOURCE`: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.