Module 1

Enumerations, Autoboxing and Annotations(metadata): Enumerations, Enumeration fundamentals, the values() and valueOf() Methods, java enumerations are class types, enumerations Inherits Enum, example, type wrappers, Autoboxing, Autoboxing and Methods, Autoboxing/Unboxing occurs in Expressions, Autoboxing/Unboxing, Boolean and character values, Autoboxing/Unboxing helps prevent errors, A word of Warning. Annotations, Annotation basics, specifying retention policy, Obtaining Annotations at run time by use of reflection, Annotated element Interface, Using Default values, Marker Annotations, Single Member annotations, Built-In annotations.

Inheritance

Inheritance serves code **reusability** and **extensibility**

Reusability: data & functions present in base class can be reused
Extensibility: functions present in base can be extended (re-structured) to meet the needs of child class.

Functions present in base class exhibits GENERALIZED activity
Functions inherited and extended in child class exhibits SPECIALIZED activity.

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

Object class contains methods which can be categorized into 2
- final methods
- non-final methods

Inheritance

final methods that are present in base class Object can be used by the child class

Ex: public final class getClass( );

returns the Class **class** object of this object. The Class **class** can further be used to get the metadata of this class.

non-final methods that are present in base class Object can be further restructured for the purpose of child class.

Ex: public boolean equals(Object obj)

compares the given object to this object.

GENERALIZED activity of equals is comparing reference equality (meaning checking whether 2 references are pointing to the same object)

public String toString()

returns the string representation of this object.

Inheritance

Further base class reference is to be considered as a generic reference in class hierarchy.

Ex: Object class is the supreme base class in Java language and its properties are inherited automatically to all the built-in and user-defined classes in java.

Hence Object reference can be used to point to any object/instance present in java.

Ex: Object obj = new Complex();
Complex is an user-defined class and Object is built-in supreme class.

System is a final class in java.lang package
out is an instance of PrintStream type, which is public and static member field of System class.

Interface

Consider the simulation of stack in java for integer, float and char variables.

Further the implementation details of these 3 classes can be **aggregated** by an interface.

interface will have the necessary member functions that must be compulsorily implemented by the classes which simulate the task of integer-stack, float-stack or char-stack.

Interfaces are used to achieve multiple inheritance in Java.

Java Interface also represents the **"IS-A"** relationship.

A single class can inherit multiple interfaces.

A single class **cannot** inherit multiple abstract classes or classes.

"implements" is the keyword used to inherit the interface.

Interface

G.F:

```
<access-specifier>   interface   <interface-name>
 {
        return-type method-name1(parameter-list); //method declaration
        return-type method-name2(parameter-list);
        …….
        type final varname1 = value;   // final and static variables which must be initialized
        type final varname2 = value;
}
```

Ex:

```
interface   test
{   }

class <calss-name> [extends classname] [implements interface [,interface...]]
{
 ....//class body
}
```

If a class **inherits or implements** an interface “test”, then all the methods in the test must be compulsorily defined by the inherited class.

All methods and variables are implicitly public.

NOTE: An interface can extend (inherit) another interface.

```java
interface disp      {
        public void display();
        int i=90; // by default i is public final static           }

class cmp implements disp      {
        public void cmp_method()
        {   System.out.println("method of cmp");    }

        @Override
        public void display()
        {   System.out.println("in cmp display");   }  }

class student implements disp   {
        public void student_method()
        {   System.out.println("method of student");   }

        @Override
        public void display()
        {   System.out.println("in student display");   }  }

class test {
 public static void main(String[] args) {
        disp d = new student();            d.display();  //d.student_method( ) CTE
         d = new cmp();            d.display();     //d.cmp_method( )   CTE  }  }
```

A single base class or interface inherited by many derived classes - **HIERARCHICAL** inheritance

HI support “**one interface multiple implementation**” technique in OOP

Using a **reference of type base class or an interface**, methods that are inherited and overridden or defined in the inheritance hierarchy can be invoked.

But, the members which belong solely to the derived class cannot be accessed by base class or interface reference.

Enumerations

Enumeration is a list of named constants.

Enumeration is created using the enum keyword.
Ex: enum color { RED, GREEN, BLUE};
enum declarations are not allowed inside main function.

RED, GREEN, BLUE are termed as enumeration constants.

Each is declared as a public, static final member of **color**.
(final members are like constants in java)

Enumeration constants type, is the type of enumeration in which they are declared, which is **color** in this case.

A variable of type enumeration can be defined.
Ex: color c;

Enumerations

c is of type color, the only value that can be assigned are those defined by enumeration.

```
Ex:  c = color.RED;  //RED is preceded by color enum type.
     System.out.println(c);  // prints RED
```

Two enumeration constants can be compared for equality using the == relational operator.
(no other relational operators can be used on enum constants except == & !=)

```
Ex:    color c = color.RED, d=color.BLUE;
       if ( c != d )
            System.out.println("c != d");
```

An enumeration value can be used to control a switch statement, where in all of the case statements must use constants from the same enum as that used by the switch expression.

```
Ex:    color c=color.RED;
       switch(c)  {
            case RED:    System.out.println("R selected");   break;
            case GREEN: System.out.println("G selected");   break;
            case BLUE:   System.out.println("B selected");    break;     }
```

Enumerations

values( ) and valueOf( ) methods

All enumerations automatically contain 2 predefined methods: values( ) and valueOf( ).

prototype is:

```
public static enum-type[ ] values( )
public static enum-type valueOf(String str)
```

values( ) method returns an array that contains a list of the enumeration constants.

valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.

Ex:
```
enum color {RED, GREEN, BLUE};

color d[] = color.values();

for (color i : d)
  System.out.println(i);  // prints RED GREEN BLUE
```
OR
```
for(color i : color.values() )
  System.out.println(i);  // prints RED GREEN BLUE
```

Enumerations - Java Enumerations are class types

Java enumeration types are class types.

Even though enum instances are not instantiated using new, properties are similar to class.

Considering an enum as a class helps the language to attach more supporting methods to it.
(Wrapper classes - slide 15)

By considering enum as a class the following facilities can be added, such as constructors, add instance variables and methods, and even implement interfaces.

Each enumeration constant is an object of its enumeration type.

Hence, when a constructor is defined for an enum, the constructor is called when each enumeration constant is created.

Each enumeration constant has its own copy of any instance variables defined by the enumeration.

Enumerations - Java Enumerations are class types

Ex: enum color {

Red(10), Blue(40), Green(60); private int ctype;

// instance variable, only becz of this value of constants can be stored and used. When constructor is called the values //of constants are stored in ctype. Each constant has its own copy of ctype.

color (int p) {

System.out.print(p); ctype=p;

System.out.println(this.getClass()); }

int getColor() { return ctype; } }

public class Main {

public static void main(String[] args) {

color c = color.Green; // only if an instance of type is created the constructors of enum will be called

System.out.println(c.getColor() + " " +c); } }

Output:

10 class color *System.out.println(getClass()); to prove that Red, Blue and Green are instantiated or created.*

40 class color

60 class color

60 Green

Enumerations - Java Enumerations are class types

class color adds 3 things

1. Instance variable of ctype, which can hold the value of color type.
2. Constructor, which is passed the value of a color.
3. Method getcolor( ), which returns the value of color constant.

When a variable c is defined **and initialized** with a value of type enum, the constructor for color is called once for each constant that is specified inside the enum.

Numerical values of constants (inside parenthesis) are passed as parameter to the constructor and assigned to ctype. Constructor is called once for each constant.

Each enumeration constant has its own value in ctype. These values can be accessed using getcolor( ) method.
Ex: color.Red.getcolor()

Overloaded constructors are also allowed in enum

Enumerations - Java Enumerations are class types

Ex:
```
enum color {
            Red(10), Blue(40), Green;

            private int ctype;  // instance variable
            color (int p) { System.out.println("Param C");  ctype=p; }
            color() { System.out.println("Zero PC"); ctype=0;}
            int getcolor() { return ctype; }
      }
      public class Main {
            public static void main(String[] args)  {
            color c=color.Green;  System.out.println(c.getcolor());
       }  }
```

Output:

```
Param C
Param C
Zero PC
0
```

Enumerations - Java Enumerations are class types

Green is assigned a value 0

Restrictions that apply to enumerations are

1. Enumeration cannot inherit another class
2. enum cannot be a superclass. (enum cannot be extended)

3 important properties associated with an enumeration are as follows

1. The **position of enumerations constant's in the list** can be obtained using the method **ordinal( )**, termed as ordinal value.

   final int ordinal( )

   This method returns the ordinal value of the invoking constant. Ordinal values begin from zero.

   Ex: System.out.println(color.Red.ordinal( )); // 0

Enumerations - Java Enumerations are class types

2. Ordinal values of the two constants can be compared, which belong to the same enumeration by using **compareTo( )** method.

   final int compareTo(enum-type e)

   Enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant.

   Invoking constant and e must be of the same enumeration type.

   If the invoking constant has an ordinal value less than e's, then compareTo( ) returns a negative value. **If the two ordinal values are the same, then zero is returned (NOT POSSIBLE)**. If the invoking constant has an ordinal value greater than e's, then positive value is returned.

Ex:

```
if ( (color.Red).compareTo(color.Green) < 0 )
        System.out.println("Red is less than green");
```

Enumerations - Java Enumerations are class types

3. Two enumeration **references** can be compared using ==

Ex: color c=color.Red;   color d=color.Green; //& color d = color.Red;

```
if (c==d)
   System.out.println("c & d are pointing to the same instance");
```

Two enumeration constant values can be compared by equals method

Ex: color c=color.Green; color d=c;

```
if (c.equals(d))  // constant value comparison
System.out.println("c & d are holding same values");
```

equals( ) cannot be overridden in enum class.

Only Generalized task of equals (which is reference comparison) can be used.

Basically equals( ) and == perform the same redundant task ???!!!

Enumerations - Java Enumerations are class types

```
import java.util.*;
enum branch {   CSE, ISE, CV, MECH  }
public class Main   {
    public static void main(String[] args) {
        System.out.println("Most opted branch in engg");
        System.out.println("1. CSE");    System.out.println("2. ISE");
        System.out.println("3. CV");      System.out.println("4. MECH");
        System.out.println("Ans: ");      Scanner sc = new Scanner(System.in);
        int ans = sc.nextInt();
                branch ar[] = branch.values();
        if (ar[ans-1] == branch.CSE)     // if ( ar[ans-1].equals(branch.CSE))
        System.out.println("Correct answer");
        else
        System.out.println("Wrong answer");
    }
}
```

type wrappers

Java supports primitive types such as int, double etc.,, to hold the basic information types.

Using objects for these values will add an overhead for even the simplest calculations.

Primitive types are not part of the object hierarchy, and they will not inherit from **Object**.

Despite the performance benefit offered by the primitive types, there are times when primitive types are needed in an object manner.

Ex: Primitive types cannot be passed as reference to a method.

Many of the data structures (collections) implemented by java operate on objects.

To handle these situations java provides type wrappers, which are classes that encapsulate primitive types with an object.

The type wrappers are **Double, Float, Long, Integer, Short, Byte, Character** and **Boolean**.

type wrappers

Character

Character is a wrapper class for char. The constructor for Character is

Character(char ch)

ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, method charValue() has to be used.

char charValue( )

this returns the encapsulated character.

Boolean

Boolean is a wrapper around boolean values. It defines these constructors.

Boolean(boolean boolValue)

Boolean(String boolString)

In the first constructor, boolValue must be either **true** or **false.**

In the second constructor, if boolString contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, method booleanValue() can be used.

boolean booleanValue( ) returns the boolean equivalent of the invoking object..

type wrappers
Most commonly used type wrappers are those that encapsulate numeric values. These are **Byte, Short, Integer, Long, Float** and **Double.**

All numeric type wrappers inherit the **abstract class Number.**

Number declares methods that return the value of an object in each of the different number formats.

byte	byteValue( )
double	doubleValue( )
float	floatValue( )
int	intValue( )
long	longValue( )
short	shortValue( )

These methods are implemented by each of the numeric type wrappers.

All numeric type wrappers define constructors that allow an object to be constructed from a **value** or a **string** representation of that value.

type wrappers

Ex: constructors defined for Integer wrapper classes are

Integer(int num)

Integer(String str)

If str does not contain a valid numeric value, then a **NumberFormatException** is thrown.

All of these type wrappers override **toString( )**.

```
Integer iob = new Integer(100);
System.out.println(iob.toString() + " "+iob); // 100

int i=iob.intValue();
System.out.println(i);  // 100
```

Autoboxing

Autoboxing is a process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

No need to explicitly construct an object. Assign a primitive value to a type-wrapper reference, Java automatically constructs the object.

Ex: Integer iob = 100; // autobox an int

```
public class Main {
    public static void autoboxing(Integer i)
    {   System.out.println(i);    }

    public static void main(String[] args)
    {    autoboxing(10);  }
}
```

Autoboxing & Auto-unboxing

No **explicit object** is created through new, Java handles this automatically.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted or unboxed from a type wrapper when its value is needed. No need for calling method intValue( ) and doubleValue( ).

To unbox an object, assigning the object reference to a primitive type variable will suffice.

Ex: Integer iob=100;//autobox int i = iob; // auto unbox

```
public class Main {

  public static void autoboxing(Integer i)
  { System.out.println("Boxing "+i); }

  public static void autounboxing(int i)
  { System.out.println("UnBoxing "+i); }

    public static void main(String[] args) {
        autoboxing(10);
        autounboxing(new Integer(10));
    } }
```

Autoboxing & Auto-unboxing

Advantages of autoboxing and auto-unboxing are, it eases the job of manually boxing and unboxing values. It also helps prevent errors. Auto-boxing and unboxing is important to generics, which operates only on objects.

Autoboxing and unboxing is not limited only to assignment statements, but autoboxing automatically occurs whenever a primitive type must be converted into an object and auto-unboxing occurs whenever an object must be converted into a primitive type.

Autoboxing and auto-unboxing will happen when an argument is passed to a method or when a value is returned by a method.

Autoboxing/unboxing occurs in Expressions

Ex: Integer iob=90;

```
// the following statement automatically unboxes iob.  Performs the increment and then reboxes the result back into iob.

        ++iob;
        System.out.println("value of iob "+iob);   // 91

// In the following statement iob is unboxed, the expression is evaluated and the result is boxed and assigned to iob2.
        Integer iob2;
        iob2 = iob + (iob + 3);
        System.out.println("value of iob2 "+iob2);  // 181
```

Auto-unboxing allows different types of numeric objects to be mixed in an expression.

Once the values are unboxed, the standard type promotions and conversions are applied.

Ex:

```
        Integer iob=10;
        Double dob=10.2;

        dob = dob + iob;
        System.out.println(dob)
```

As can be seen Double object and Integer object is used in expression, and the result is boxed and stored in dob.

Because of auto-unboxing integer numeric objects can be used to control switch statement.

Ex:
```
Integer iob=11;
 switch(iob)    {
        case 10: System.out.println("iob value is 10"); break;
        case 11: System.out.println("iob value is 11"); break;     }
```

When the switch expression is evaluated, iob is unboxed and its int value is obtained.

Autoboxing/Unboxing Boolean and Character Values

Wrapper classes for boolean and char are Boolean and Character.

Autoboxing and unboxing applies to these wrappers too.

Ex:
```
Boolean b = true;
// b is auto-unboxed before used in conditional expression
if (b)
  System.out.println("b is true");
//Autobox/unbox a char
Character c = 'x';
char c1 = c;
System.out.println(c+"\n"+c1);
```

In java conditional expressions must yield a boolean value, and since b is an object, auto-unboxing will extract the boolean value stored in the object b.

Autoboxing/Unboxing helps prevent Errors

Ex:

```
Integer iob = 1000;

int i = iob.byteValue();  // manual unboxing

System.out.println(i);  // -24
```

Considering the above code snippet output will be -24, because of wrong interpretation of value stored in Integer object as byte type.

Auto-unboxing prevents this type of error because the value of iob will always auto-unbox into a value compatible with int.

In general, because autoboxing always creates proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value.

A word of warning

Ex:

```
Double a, b, c;
a=10.0; b=4.0;
c = Math.sqrt(a*a + b*b);

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

Although the above code snippet generates proper output, because of autoboxing and auto-unboxing adds overhead. (in terms of converting values from primitive to object and vice versa)

Using primitive types will reduce the amount of time taken to perform the same task.

Annotations

Annotations are used to provide supplement information about a program. Annotations start with ‘@’.

Annotations do not change the action of a compiled program.

Annotations help to associate metadata (information) to the program elements
i.e. instance variables, constructors, methods, classes, etc.
These metadata can be accessed during program execution.

Annotations are not comments as they can change the way a program is treated by the compiler.
Ex: @Override

Annotation leaves the semantics of a program unchanged.
However, this information can be used by various tools during both development and deployment.

An important factor with annotation is, **they are a standard way of defining metadata in code**.

Prior to annotations, comments were used as metadata of the code.

Annotations - Custom annotations

Annotation is a super-interface of all annotations.

It is declared within the **java.lang.annotation** package.

It overrides **equals( )**, and **toString( )**, which are defined by **Object**.

It also specifies **annotationType( )**, which returns a **Class** object that represents the invoking annotation.

After declaring an annotation, it can be used to annotate an entity (such as class, method or data members).

Any type of declaration can have an annotation associated with it.
Ex: classes, methods, fields, parameters, and **enum** constants can be annotated.

In all cases, the annotation precedes the rest of the declaration.

When an annotation is applied, **values are assigned** to its members.

```
import java.util.*;
interface disp {
        default void display() {
            System.out.println("In default display ");
        }
}
class comp implements disp  {
        @Override
        public void display()   {
            disp.super.display(); // to call base class generailzed functionality
            System.out.println("In overridden display");
        }
}
class JavaApplication4  {
        public static void main(String[] args)
        {
           comp p = new comp();
           p.display();
        }
}  //super.toString( ) will call the generalized function which is present in base clas..
```

Annotations

Annotations can be classified as

1. Predefined annotations
    - @Depricated
    - @Override
    - @SuppressWarnings
    - @SafeVarargs
    - @FunctionalInterface

2. Custom annotations

3. Meta-annotations - *used only on Custom annotations*
    - @Retention
    - @Documented
    - @Target
    - @Inherited
    - @Repeatable

Annotations - Custom annotations

An annotation is created via **interface** keyword

Custom annotation can be divided into 3 types

1. Marker Annotation 2. Single valued Annotation 3. Multi valued Annotation

Ex:// A simple annotation of **Multi Value-type**.

```
@interface MyAnno
{
   String str();
   int val();
}
```

Points regarding custom annotation.

1. Method should not have any throws clauses
2. Method should return one of the following, primitive data types
   int, float, String, Class, enum or array.
3. Method should not have any parameter.
4. Symbol @ must be attached, before the interface keyword to define annotation.
5. Default value can be assigned to the method.

Annotations - Custom annotations

Ex:

```
// A simple annotation type.
    @interface MyAnno
    {
        String str();
        int val();
    }
```

All annotations consist solely of method declarations.

Ex: the two members **str( )** and **val( )**.

No statements are provided for the methods str( ) and val ( ).

Moreover, the methods act much like fields. Java implements these methods automatically.

An annotation cannot include an **extends** clause.

Annotations - Custom annotations

Ex: // Annotate a method.

```
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { …  }
```

This annotation is linked with the static method **myMeth( )**.

The name of the annotation is, preceded by an **@**, is followed by a parenthesized list of member initializations.
An annotation member name is assigned with a value, which constitutes metadata of method.
Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**.

Specifying a Retention Policy
A retention policy determines the **lifetime of an annotation**.

Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration.

They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

Annotations - Specifying a Retention Policy

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**.

Its general form is: **@Retention(retention-policy)**
Here, retention-policy must be one of the previously discussed enumeration constants.

If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

Ex:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
    String str();
    int val();
}
```

Annotations - Obtaining Annotations at Run Time by Use of Reflection

Annotations are used mostly by other deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of reflection.

Reflection is the feature that helps to retrieve metadata information of a program during runtime.

The reflection API is contained in the **java.lang.reflect** package.

The first step in using reflection is to obtain a **Class** object that represents the class whose annotations has to be obtained.

Class is one of Java's built-in classes and is defined in **java.lang**.

One of the ways (3 ways are there) to obtain a class object is to call **getClass( )**, which is a method defined by **Object**. Its general form is shown here:

final Class getClass( ); returns the **Class** object that represents the invoking object.

Class objects are constructed automatically by the Java Virtual Machine. (explicit creation not necessary)

Ex; Class c = new Class( ); //generates CTE

It is a final class, hence cannot be extended. Instances of Class represent names of classes and interfaces in a running Java application.

Annotations - Obtaining Class object type

Ex:

```
class complex {    }

class test {
        public static void main(String [] args) {
                complex a = new complex();
                Class c = a.getClass( );
                System.out.println(c);
        }   }
```

Output: class complex   or   class <package-name>.complex

The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the keyword *void* are also represented as Class types.

System.out.println(**int.class**);   //prints int

Only the primitive, data types can be retrieved using **.class** method, like **float.class, char.class** etc.,

Primitive types has no public or private fields and methods.

Annotations - Obtaining Class object type

```
class complex {    }

class Main
{
        public static void main(String [] args)
        {
                complex a = new complex();
                Class c = a.getClass( );  //obtaining type using object/instance
                System.out.println(c);

                c = Main.class; // not a built-in class
                System.out.println(c); //obtaining type using class name

                c=int.class;
                System.out.println(c); //obtaining type using primitive type
        }

  }
```

Annotations - Obtaining Annotations at Run Time by Use of Reflection

After obtaining a **Class** object, it can be used to obtain information about the various elements declared by the class, including its annotations.

If annotations associated with a specific item within a class has to be retrieved, first an object must be obtained that represents that item.

Ex: **Class** supplies (among others) the **getMethod( )**, **getField( )**, and **getConstructor( )** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

getMethod( ) has this general form:

Method getMethod(String methName, Class ... paramTypes)

The name of the method is passed in methName.

If the method has argument/s, then **Class** objects representing those types must also be specified by paramTypes. paramTypes is a varargs parameter.

varargs is the way of passing any number of arguments.

Annotations - Obtaining Annotations at Run Time by Use of Reflection
getMethod( ) returns a **Method** object that represents the method.

If the method can’t be found, **NoSuchMethodException** is thrown.

After obtaining the object of type, **Method** or **Field** or **Constructor** object, from **Class,** annotation can be obtained associated on the same by calling **getAnnotation( )**.

Its general form is shown here:

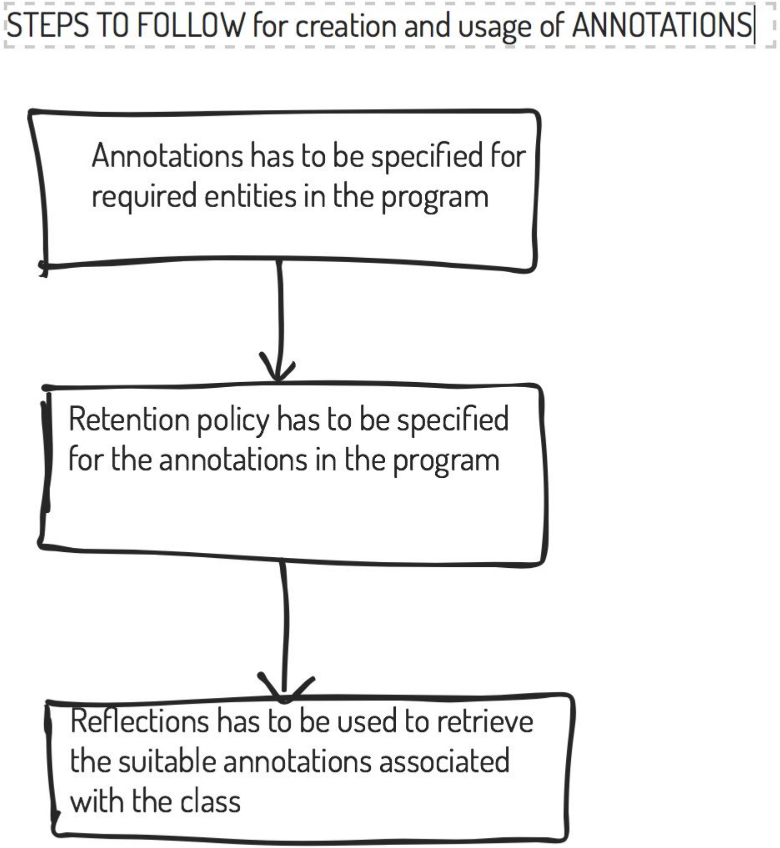
Annotation getAnnotation(Class annoType)

Here, annoType is a **Class** object that represents the annotation which has to be retrieved.
This method is commonly present in Method, Field or Constructor built-in class.

The method returns a reference to the annotation. Using this reference, the values associated with the annotations’ can be retrieved.

The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Annotations - Steps to use custom annotations



Annotations - Obtaining Annotations at Run Time by Use of Reflection

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
   String str();   // String str() default  “string value”
//default value will be considered when no value is assigned to str field.
   int val();
}

class Anno_test {
  // Annotating a method.
  @MyAnno(str = "Annotation Example", val = 100)
//str field value can be omitted only if default value for str is specified in interface declaraiton
  public static void myMeth()
  {  }
```

Annotations - Obtaining Annotations at Run Time by Use of Reflection

```
public static void main(String args[])   {
    Anno_test ob = new Anno_test();
        // First, get a Class object that represents this class.
            Class c = ob.getClass();
            Method m=null;
    try {
            // Now, get a Method object that represents this method.
            m = c.getMethod("myMeth");
        }
    catch (NoSuchMethodException  exc)
    { System.out.println("Method Not Found."); }

        // Next, get the annotation for this class.
            MyAnno anno = m.getAnnotation(MyAnno.class);

            // Finally, display the values.
            System.out.println(anno.str() + " " + anno.val());
}   }
```