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

1. **Enumerations:**
    - Enumerators contain a list of constant values that apply to a certain type of data, or object.
    - They can be useful in setting a scope of values for a particular object.
    - An enumeration defines a class type.
    - An enumeration can have constructors, methods, and instance variables.
    - An enum is actually a new type of class.
    - You can declare them as inner classes or outer classes.
    - You can declare variables of an enum type.
    - Each declared value is an instance of the enum class.
    - Enums are implicitly public, static, and final.
    - enums extend java.lang.Enum and implement java.lang.Comparable.
    - Supports **equals, "==", compareTo, ordinal,** etc.
    - Enums override toString() and provide valueOf(…), name().

Points to remember for Java Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

1.1 Enumeration fundamentals :

enumeration is a special kind of class that includes a list of constant values. The values in the enumeration list define the values an object can have

Creating Enumerations

- When creating an enumeration list, we don't use the keyword class and when you create a enumeration object, we don't use the keyword new
- To create an enumeration list we need to import **java.util.Enumeration**
- An enumeration is created using the **enum** keyword followed by the variable Name we want associated with the list

Syntax:

```
public enum variableName{
   ITEM1
```

ITEM2
ITEMN
}

example:

```
public enum Gender {
MALE, FEMALE, UNKNOWN
}
```

Creating an enumeration object

To declare an enumeration object, use the variable name associated with the list followed by the name of the object.

Syntax: variableName object;

example: Gender gen;

Assigning values to the enumeration object

To assign values to the object, must use the enumeration name, a dot and one of the values in the list.

Syntax: object = variableName.ITEM;

example: Gender gen=Gender.MALE;

Assigning default values

We can also assign the default value to list of item in enumeration class. In order to assign default value to list in enumeration class, should contain the enumeration constructor,fields,methods.

example:

```
public enum Gender {
MALE(1), FEMALE(2), UNKNOWN(0)
}
```

Simple Program:

```
enum Gender {
        MALE, FEMALE, UNKNOWN
    }

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

```
 {
  Gender S;
  S=Gender.MALE;
  System.out.println("Gender IS="+S);

 }

}
```

OUTPUT:

Gender IS=MALE

Java enum in CONTROL STATEMENTS:

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. In Java control statements are categorized into selection control statements, iteration control statements and jump control statements.

- **Java's Selection Statements:**

  These statements allow us to control the flow of program execution based on condition.

  Types of control statements are:

  1)Simple if Statement

  2)if else

  3)else if

  4)switch

  1.Simple if Statement:

  The Java if statement tests the condition. It executes the *if block* if condition is true.

  Syntax: if(condition)code to be executed

  Program:

```
enum Gender {
        MALE, FEMALE, UNKNOWN;


    }

class cont
{
```

```
 public static void main( String args[] )
 {
  Gender s=Gender.FEMALE;
 if(s==Gender.MALE)
                  System.out.println("Both are not equal");


        }

        }
```

program 2: To find Smallest of given number.

```
enum Value {
                a(10), b(20);

                int a1;
                int getValue(){ return a1;}
                Value(int value)
                {
                        this.a1=value;
                }
        }

class Enu
{
 public static void main( String args[] )
 {
  int s=Value.a.getValue();

 if(s<Value.b.getValue())
          System.out.println("a value is small");
        }

        }
```

2)if else: The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax: if(condition) //code if condition is true

else

```
 //code if condition is false
 }
```

program 1:

```
enum Gender {
                MALE, FEMALE, UNKNOWN;
```

```
        }

class cont
{
 public static void main( String args[] )
 {
   Gender s=Gender.FEMALE;
 if(s==Gender.MALE)
        System.out.println("both are equal");
 else
        System.out.println("Both are not equal");

 }
}
```

output: Both are not equal

Program 2:To find Smallest of two numbers

```
 enum Value {
                a(10), b(20);

                int a1;
                int getValue(){ return a1;}
                Value(int value)
                {
                        this.a1=value;
                }
        }

class Enu
{
 public static void main( String args[] )
 {
   int s=Value.a.getValue();

 if(s<Value.b.getValue())
        System.out.println("a value is small");
 else
        System.out.println("b value is small");


 }
}
```

output: a value is small

3)else if:

This statement perform a task depending on whether a condition is true or false.

Syntex: if(condition)

Statement

Else if(condition)

Statement

Else

Statement

program 1:

```
enum Gender {
                MALE, FEMALE, UNKNOWN;


        }

class cont
{
 public static void main( String args[] )
 {
   Gender s=Gender.FEMALE;
  if(s==Gender.MALE)
          System.out.println("both are equal");
  else if (s==Gender.UNKNOWN)
          System.out.println("Both are  equal");
  else
          System.out.println("Both are not equal");


 }
}
```

program 2:

```
 enum Value {
                a(10), b(20), c(30);

                int a1;
                int getValue(){ return a1;}
```

```
        Value(int value)
        {
                this.a1=value;
        }
    }

class Enu
{
 public static void main( String args[] )
 {
  int s=Value.a.getValue();

 if(s<Value.b.getValue())
        System.out.println("a value is small compare to b");
 else if(s<Value.c.getValue())
        System.out.println("a value is small comapare to c");
 else
        System.out.println("a value is grater");

 }
}
```

output: a value is small compare to b

Switch Statement:

When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax: switch (expression)

```
  {  case value1:  //statement sequence
       break;
    case value2:  //statement sequence
       break;
    case valueN:  //statement sequence
       break;
    default:  //default statement sequence
  }
```

program 1: To check the gender of person

```
 enum Gender {
        MALE, FEMALE, UNKNOWN;
```

```
        }

class cont
{
 public static void main( String args[] )
 {
   Gender s=Gender.FEMALE;
  switch(s)
  {
  case MALE:System.out.print("Gender is mail");
            break;
  case FEMALE:System.out.print("Gender is femail");
  break;
  case UNKNOWN:System.out.print("Gender is unknown");

  break;
  default:System.out.print("NON of these");
  }
 }
}
```

output: Gender is femail

Java's Iteration Statements: Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops

Types of looping statements are:

1)while

2)do while

3)for

1) **while Loop:** while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.

Syntax: while (condition)

```
  {
    statements;
  }
```

Program: To find Sum of given number:

```
enum Value {
                NUM(10);

                int a1;
                int getValue(){ return a1;}
                Value(int value)
                {
                        this.a1=value;
                }
        }

class Enu
{
 public static void main( String args[] )
 {
  int n=Value.NUM.getValue();
  int sum=0,i=0;
  while( i<n)
  {
           sum+=i;
           i++;
  }
  System.out.println("sum of given number is="+sum);
}
}
```

output: sum of given number is=45

2)do while Loop: do…while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do…while loop is also called as exit control loop.

Syntax: do

```
  {
    statements;
  } while (condition);
```

Program: To find sum of given number:

```
enum Value {
                NUM(10);

                int a1;
                int getValue(){ return a1;}
                Value(int value)
```

```
                {
                        this.a1=value;
                }
        }

class Enu
{
 public static void main( String args[] )
 {
  int n=Value.NUM.getValue();
  int sum=0,i=0;
  do
  {
          sum+=i;
          i++;
  }while( i<n);
  System.out.println("sum of given number is="+sum);
}
}
```

output: sum of given number is=45

3.for Loop: The for loop is also same as do…while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)

```
  {   statements;
  }
```

Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

program : To find the sum of given number.

```
 enum Value {
                NUM(10);

                int a1;
                int getValue(){ return a1;}
                Value(int value)
                {
                        this.a1=value;
                }
        }

class Enu
```

```
{
 public static void main( String args[] )
 {
  int n=Value.NUM.getValue();
  int sum=0;
  for(int i=0;i<n;i++)
  {
          sum+=i;
  }
  System.out.println("sum of given number is="+sum);
}
}
```

output: sum of given number is=45

2. **Java Enumerations Are Class Types**

- Enumerations in Java can have methods, members and constructors just as any other class can have.
- Each enumeration constant is an object of its enumeration type.
- Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created.
- Also, each enumeration constant has its own copy of any instance variables defined by the enumeration.
- the enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value(default value) to the enum constants by defining fields and constructors.

Syntax:

```
enum variableName
{
 ITEM1(1), ITEM2(20), ITEM3(30);
data-typevariableName;
 data-type methodName()
{ statement;}
enumName (parameter-list) {
        statements;
}

}
```

Progarm:

```
enum Value {
        A(10), B(20), C(30);

        int a;
        int getValue(){ return a;}
```

```
        Gender(int value)
        {
                this.a=value;
        }
    }

class Enu
{
 public static void main( String args[] )
 {

  System.out.println(" value of A IS="+Value.A.getValue());


 }

}
```

output: Value of A Is= 10

3.the values() and valueOf() Methods:

There are some methods that you can use that are part of the enumeration class, these methods include **:**

1. **values()**
2. **valueOf()**
3. **ordinal()**
4. **compareTo()**
5. **toString()**

1.Values():

- Method returns an array that contains a list of the enumeration constants
- values() returns the values in the enumeration and stores them in an array. We can process the array with a foreach loop.

Systex: public static *enum-type[ ] values( )*

example:
```
Values value[] = Values.values();
        for (Values a : value)
        statement;
```

Progarm 1: To print the list of day of enumeration class

```
 enum Days {
        mon,tue,wed,thu,fri,sat,sun;
```

```
        }

class cont
{
 public static void main( String args[] )
 {
   Days d[]=Days.values();
   for(Days d1:d)
            System.out.println("today day is:"+d1);
 }
}
output:
today day is=mon
today day is=tue
today day is=wed
today day is=thu
today day is=fri
today day is=sat
today day is=sun

program 2:TO find no of days in month

enum month {
        January(31),
February(28),
        March(31),
        April(30),
        May(31),
        June(30),
        July(31),
        August(31),
        September(30),
        October(31),
        November(30),
        December(31);

        private final int days;

   month(int days) {
      this.days = days;
   }

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

```
        for(month month1 : month.values( ))
        {
                System.out.println(month1+":"+month1.days);
        }

  }}
```

output:
January:31
February:28
March:31
April:30
May:31
June:30
July:31
August:31
September:30
October:31
November:30
December:31

2.ValueOf():

- **method returns the enumeration constant whose value corresponds to the string** passed in *str*.
- method takes a single parameter of the constant name to retrieve and returns the constant from the enumeration, if it exists.

Syntax: enumerationVariable = enumerationName.valueOf("EnumerationValueInList");

Example

```
        WeekDays wd = WeekDays.valueOf("MONDAY");
        System.out.println(wd);
```

program:

```
enum Days {
        monday,tuesday;


    }

class cont
{
 public static void main( String args[] )
 {
  Days d=Days.valueOf("monday");
  System.out.println("day selected is:"+d);
 }
}
```

output: day selected is:Monday

Note: An enumeration cannot inherit another class and an enum cannot be a superclass for other class.

4.Enumerations Inherit Enum:

All enumerations in Java inherit the Enum class, java.lang.Enum, which provide a set of methods for all enumerations. The two mentioned here are ordinal( ) and compareTo( ).

1.ordinal():

- Returns the value of the constant's position in the list (the first constant has a position of zero).
- The ordinal value provides the order of the constant in the enumeration, starting with 0

Example

```
WeekDays wd = WeekDays.MONDAY;
System.out.println(wd.ordinal());
```

program: To find index of Enum List.

```
enum Days {
                mon,tue,wed;
                }
class cont
{
 public static void main( String args[] )
 {
        Days wd = Days.mon;
        System.out.println("Index of list:"+wd.ordinal());
 }
}
```

output: Index of list:0

Program 2: Using foreach loop

```
enum Days {
                mon,tue,wed;
                }
class cont
{
 public static void main( String args[] )
 {
        Days wd[] = Days.values();
```

```
        for(Days w:wd)
        System.out.println("Index of list:"+w.ordinal());
 }
}
```

output:
Index of list:0
Index of list:1
Index of list:2

2.compareTo():

- compares the ordinal value of the two enumeration objects.
- If the object invoking the method has a value less than the object being passed, a negative number is returned.
- It the two objects have the same oridnal number, a zero is returned.
- If the invoking object has a greater value than the one being passed, a positive number is returned.

Syntax: int num = enumObject1.compareTo(enumObject2);

program: To find order of day

```
 enum Days {
                mon,tue,wed;
                }
class cont
{
 public static void main( String args[] )
 {
        Days d1,d2,d3;
        d1=Days.mon;
        d2=Days.tue;
        d3=Days.wed;
        if(d1.compareTo(d2)<0)
                System.out.println("mon comes befor tue");
        if(d2.compareTo(d3)<0)
                System.out.println("Tue comes befor wed");

                System.out.println("wed comes after tue");


 }
}
```

output:
mon comes befor tue
Tue comes befor wed
wed comes after tue

4.Type wrappers:

- Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language.
- Instead of primitive types if objects are used everywhere for even simple calculations then performance overhead is the problem.
- So to avoid this java had used primitive types.
- So primitive types do not inherit Object class
- But there are times when you will need an object representation for primitives like int and char.
- Example, you can't pass a primitive type by reference to a method.
- Many of the standard data structures implemented by Java operate on objects, which mean that you can't use these data structures to store primitive types.
- To handle these (and other) situations, Java provides *type wrappers, which are classes* that encapsulate a primitive type within an object.

The type wrappers are :

Double, Float, Long, Integer, Short, Byte, Character, and Boolean.

Character:

- Character is a wrapper around a char.
- The constructor for Character is Character(char ch) here ch is a character variable whose values will be wrapped to character object by the wrapper class
- To obtain the char value contained in a Character object, call charValue( ), shown here:

  char charValue( )

  It 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 version, *boolValue* must be either **true** or **false**. In the second version, 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, use **booleanValue( )**, shown here:

  boolean booleanValue( )

- It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- The most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the 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. These methods are shown here:

1. byte byteValue( )
2. double doubleValue( )
3. float floatValue( )
4. int intValue( )
5. long longValue( )
6. short shortValue( )

doubleValue( ): returns the value of an object as a **double**
floatValue( ): returns the value as a **float**, and so on.

- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

  Integer(int *num*) Integer(String *str*)

- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown. All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

Program : All wrapper class

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

Character c=new Character('@'); // character type
char c1=c.charValue();
System.out.println("Character wrapper class"+c1);

Boolean b=new Boolean(true);
boolean b1=b.booleanValue();
System.out.println("Boolean wrapper class"+b1);
```

```
Integer i1 = new Integer(100);  // integre type
int i = i1.intValue();
System.out.println("Integer wrapper class"+i); // displays 100 100

Float f1 = new Float(12.5);  // Float type
float f = f1.floatValue();
System.out.println("Float wrapper class"+f);

}

   }

   output:

Character wrapper class@
Boolean wrapper classtrue
Integer wrapper class100
Float wrapper class12.5
```

Autoboxing

- **Autoboxing** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.
- For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:
    - Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
    - Assigned to a variable of the corresponding **wrapper class**.
- **Auto-unboxing** is the process by which the value of a boxed object is automatically extracted(unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( ) or doubleValue( ).**
- For example conversion of Integer to int. The Java compiler applies unboxing when an object of a wrapper class is:
    - Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
    - Assigned to a variable of the corresponding **primitive type**.

Uses of Autoboxing and Unboxing

- Useful in removing the difficulty of manually boxing and unboxing values in several algorithms.
- it is very important to generics, which operates only on objects.
- It also helps prevent errors.
- autoboxing makes working with the Collections Framework
- here is the modern way to construct an **Integer object that has the value 100:**

Integer iOb = 100; // autobox an int

- Notice that no object is explicitly created through the use of **new.** Java handles this for you,automatically.
- To unbox an object, simply assign that object reference to a primitive-type variable.
- For example, to unbox **iOb**, you can use this line:

int i = iOb; // auto-unbox

Program: Simple program for autoboxing and autoUnboxing

```
class auto
{
 public static void main(String[] args)
 {
  Integer iob = 100; //Auto-boxing of int i.e converting primitive data type
                       int to a Wrapper class Integer
  int i = iob;   //Auto-unboxing of Integer i.e converting Wrapper class
                  Integer to a primitve type int
  System.out.println("integer type="+i+" "+iob);

  Character cob = 'a';  //Auto-boxing of char i.e converting primitive data
                          type char to a Wrapper class Character
  char ch = cob;  //Auto-unboxing of Character i.e converting Wrapper class
                  Character to a primitive type char
  System.out.println("character type="+cob+" "+ch);
 }
}
```

output:

```
integer type=100 100
character type=a a
```

Autoboxing and Methods:

- Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.
- autoboxing automatically occurs whenever a primitive type must be converted into an object.autounboxing takes place whenever an object must be converted into a primitive type

Program:

```
class auto {
static int m(Integer v) {
return v ; // auto-unbox to int
}

public static void main(String args[]) {
 Integer iOb = m(100); // Auto Boxing
 System.out.println("Integer type="+iOb);
}
}
```

output:

```
Integer type=100
```

Autoboxing/Unboxing Occurs in Expressions:

- autoboxing and unboxing take place whenever a conversion into an object or from an object is required.
- This applies to expressions. Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary

Program:

```
class auto {
public static void main(String args[]) {
Integer iOb, iOb2; int i;
iOb = 100;
System.out.println("Original value of iOb: " + iOb);     //The following
                          automatically unboxes iOb, performs the increment,
                          and then reboxes the result back into iOb.
++iOb;
System.out.println("After ++iOb: " + iOb);
iOb2 = iOb + (iOb / 3);
System.out.println("iOb2 after expression: " + iOb2);
i = iOb + (iOb / 3);
System.out.println("i after expression: " + i);
}
   }
```

output:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

Autoboxing/Unboxing Boolean and Character Values:

Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too

- Character ch = 'x'; // box a char
- char ch2 = ch; // unbox a char
- Boolean b = true; here the value true is boxed in b
- if(b) System.out.println("b is true"); // here b is unboxed

Program:

```
class auto {
public static void main(String args[]) {
//Autobox/unbox a boolean.
     Boolean b = true;
     if(b)
     System.out.println("b is true");

//    Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char
```

```
    System.out.println("ch2 is " + ch2);

}
}
output:
b is true
ch2 is x
```

Autoboxing/Unboxing Helps Prevent Errors:

- Autoboxing always creates the proper object and auto unboxing always produce the proper value.
- There is no wayfor the process to produce the wrong type of object or value.

Program:

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

Integer iOb = 1000; // autobox the value 1000
int i = iOb.byteValue(); // manually unbox as byte !!!
System.out.println("unbox value:"+i); // does not display 1000 !
}
}
```

```
output:
unbox value:-24
```

- This program displays not the expected value of 1000, but –24! The reason is that the value inside **iOb** is manually unboxed by calling byteValue( ), which causes the truncation of the value stored in iOb, which is 1,000.
- This results in the garbage value of –24 being assigned to i.
- Auto-unboxing prevents this type of error because the value in iOb will always autounbox into a value compatible with int.

A Word of Warning:

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether.

```
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

Annotation :

- Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces

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.

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.

Annotation basics

- An annotation is created through a mechanism based on the **interface.**
- A Java annotation in its shortest form looks like this:

  Systex: @interface MyAnno

- The @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared The name following the @interface character is the name of the annotation.
- All annotations consist solely of method declarations.
- However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields.
- 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
}
```

- An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package.
- It overrides **hashCode( )**, **equals( )**, and **toString( )**, which are defined by **Object**. It also specifies **annotationType( )**, which returns a **Class** object that represents the invoking

annotation.

Program: To display welcome message.

```
class annu
{
   @mymethod public void show()
   {
      System.out.println("Well come to Annotation");
   }
   public static void main(String a[])
   {
      annu h = new annu();
      h.show(); //deprecated
   }
}
```

OutPUT:

Well come to Annotation

Specifying a Retention Policy

- A retention policy determines at what point an annotation is discarded. 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.
    - 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 shown here:

    @Retention(*retention-policy*)

    - we have to pass the retention policy type.The default retention policy type is CLASS.

Example:

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

Obtaining Annotations at Run Time by Use of Reflection:

- Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package.
- The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang**. There are various ways to obtain a **Class** object. One of the easiest is to call **getClass( )**, which is a method defined by **Object**. Its general form is shown here:

final Class<?> getClass( )

- It returns the Class object that represents the invoking object.
- After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations **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**.

Method getMethod(String *methName*, Class<?> ... *paramTypes*)

- From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation( )**.

<A extends Annotation> getAnnotation(Class<A> *annoType*)

Program:

```
import java.lang.annotation.*;

import java.lang.reflect.*;
  @Retention(RetentionPolicy.RUNTIME) @interface MyAnno {
String str(); int val();

}
class annu {
```

```
@MyAnno(str = "This is Retention method", val = 100)

public static void show() {
        annu ob = new annu();
    try {
Class<?> c = ob.getClass();
Method m = c.getMethod("show");

    MyAnno anno = m.getAnnotation(MyAnno.class);
    System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {

System.out.println("Method Not Found.");
}
}
public static void main(String args[]) { show();
}
}
```

OUTPUT:

This is Retention method 100

Program 2: Passing an argument to the method.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)

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

 @MyAnno(str = "Two Parameters", val = 19)

public static void myMeth(String str, int i)
{
mymethod ob = new mymethod();

try {

Class<?>    c = ob.getClass();

Method m   = c.getMethod("myMeth",       String.class, int.class);
```

```
MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val());
}
catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");

}
}
public static void main(String args[]) { myMeth("test", 10);
}
```

OUTPUT:

Two Parameters 19

The Annotated Element Interface:

- The methods **getAnnotation( )** and **getAnnotations( )** are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**.
- This interface supports reflection for annotations and is implemented by the classes **Method**,**Field**, **Constructor**, **Class**, and **Package**, among others.
- In addition to **getAnnotation( )** and **getAnnotations( )**, **AnnotatedElement** defines several other methods.
    - **getDeclaredAnnotations( )**
    - **isAnnotationPresent( )**
    - **getDeclaredAnnotation( )**,
    - **getAnnotationsByType( )**,
    - **getDeclaredAnnotationsByType( )**.

getDeclaredAnnotations( ):

- Method returns all annotations that are directly present on this element.
- It returns all non-inherited annotations present in the invoking object.
- The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers.

Syntax: public Annotation[] getDeclaredAnnotations()

Program:

```
import java.lang.annotation.*;

import java.lang.reflect.*;
  @Retention(RetentionPolicy.RUNTIME) @interface MyAnno {
String str();
int val();
}
class annu {

@MyAnno(str = "This is Retention method and  value  is ", val = 100)

public static void show() {
        annu ob = new annu();
    try {
Class<?> c = annu.class;
Method m = c.getMethod("show");

Annotation[] annotation = m.getDeclaredAnnotations();

// print the annotation
for (int i = 0; i < annotation.length; i++) {
   System.out.println(annotation[i]);
}

}
 catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) { show();
}

}
```

output:

@MyAnno(str=This is Retention method and value is , val=100)

isAnnotationPresent( ):

- Method returns true if an annotation for the specified type is present on this element, else false. This method is designed primarily for convenient access to marker annotations.

  Syntax: public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)

Program:

```
import java.lang.annotation.*;

import java.lang.reflect.*;

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

}
class annu {

@MyAnno(str = "This is Retention method and  value  is ", val = 100)

public static void main(String args[]) {

Package[] pack = Package.getPackages();


// check if annotation hello exists
for (int i = 0; i < pack.length; i++) {
  System.out.println("" + pack[0].isAnnotationPresent(MyAnno.class));
}

}
```

OUTPUT: False….

getDeclaredAnnotation( ):

Method returns this element's annotation for the specified type if such an annotation is present, else null.This methods throws an exception

- **NullPointerException** − if the given annotation class is null
- **IllegalMonitorStateException** − if the current thread is not the owner of the object's monitor.

Syntax: public <A extends Annotation> A getAnnotation(Class<A> annotationClass)

Program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
  @Retention(RetentionPolicy.RUNTIME) @interface MyAnno {
String str(); int val();
}
class annu {
@MyAnno(str = "This is Retention method and  value  is ", val = 100)

public static void show() {
        annu ob = new annu();

    try {
Class<?> c = annu.class;
Method m = c.getMethod("show");
   MyAnno anno = m.getAnnotation(MyAnno.class);
    System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {

System.out.println("Method Not Found.");
}
}
public static void main(String args[]) { show();
}

}
```

output:

This is Retention method and value is 100

getAnnotationsByType( ):

Returns annotations that are *associated* with this element. If there are no annotations *associated* with this element, the return value is an array of length 0.

program:

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

@Retention(RetentionPolicy.RUNTIME)
@interface LogHistory {
  Log[] value();
}
@Repeatable(LogHistory.class)
@interface Log {
  String date();
  String comments();
}

@Log(date = "02/01/2014", comments = "A")
@Log(date = "01/22/2014", comments = "B")

public class HelloWorld {
  public static void main(String[] args) {
    Class<HelloWorld> mainClass = HelloWorld.class;

    Log[] annList = mainClass.getAnnotationsByType(Log.class);
    for (Log log : annList) {
      System.out.println("Date=" + log.date() + ", Comments=" + log.comments());
    }
}
}
```

output:

Date=02/01/2014, Comments=A

Date=01/22/2014, Comments=B

getDeclaredAnnotationsByType( ):

Returns this element's annotation(s) for the specified type if such annotations are either *directly present* or *indirectly present*. This method ignores inherited annotations. If there are no specified annotations directly or indirectly present on this element, the return value is an array of length 0.

Program:

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

@Retention(RetentionPolicy.RUNTIME)
@interface LogHistory {
  Log[] value();
}
@Repeatable(LogHistory.class)
@interface Log {
  String date();
  String comments();
}

@Log(date = "02/01/2014", comments = "A")
@Log(date = "01/22/2014", comments = "B")
public class HelloWorld {
  public static void main(String[] args) {
    Class<HelloWorld> mainClass = HelloWorld.class;

    Log[] annList = mainClass.getDeclaredAnnotationsByType(Log.class);
    for (Log log : annList) {
      System.out.println("Date=" + log.date() + ", Comments=" + log.comments());
    }
}
}
```

output:

Date=02/01/2014, Comments=A

Date=01/22/2014, Comments=B

Using Default Values:

we can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

type member( ) default *value* ;

Program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyINF {
String str() default "WELCOME";
int val() default 5;

}
class annu {
@MyINF()
public static void myMeth() {
annu ob = new annu();
try {
Class<?> c = ob.getClass();
Method m = c.getMethod("myMeth");
MyINF anno = m.getAnnotation(MyINF.class);
System.out.println(anno.str() + " " + anno.val());
}
catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth();
}

}
```

OUTPUT:

WELCOME 5

Marker Annotations:

A *marker* annotation is a special kind of annotation that contains no members. Its sole purpose is to mark an item. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method **isAnnotationPresent( )**, which is defined by the **AnnotatedElement** interface.

Syntax: public @interface Example{

}

Program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME) @interface MyMarker { }
class annu {
        @MyMarker
public static void myMeth() {
                annu ob = new annu();
try {

Method m = ob.getClass().getMethod("myMeth");
if(m.isAnnotationPresent(MyMarker.class))
System.out.println("MyMarker is present.");
}
catch (NoSuchMethodException exc)
{ System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
        myMeth();
        }

        }
```

output:

MyMarker is present

Single-Member Annotations:

- A *single-member* annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However,
- In order to use this shorthand, the name of the member must be **value**. Here is an example that creates and uses a single-member annotation:

Syntax:

```
public @interface Example{
  String showSomething();
   }
```

Program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyINF {
//String str() ;
        int value();
}
class annu {
@MyINF(100)
public static void myMeth() {
annu ob = new annu();
try {
Class<?> c = ob.getClass();
Method m = c.getMethod("myMeth");
MyINF anno = m.getAnnotation(MyINF.class);
System.out.println(anno.value());
}
catch (NoSuchMethodException exc) {
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth();
```

}

}

output:

100

The Built-In Annotations:

- Java defines seven built-in annotations out of which three (@Override, @Deprecated, and @SuppressWarnings) are applied to Java code and they are included in java.lang library. These three annotations are called *regular Java annotations*.
- Rest four (@Retention, @Documented, @Target, and @Inherited) are applied to other annotations and they are included in java.lang.annotation library. These annotations are called *meta Java annotations*.

Annotation Name	Applicable To	Use	Included in
Java Annotations Applied to Java code			
@Override	Member Methods	Checks that this method overrides a method from its superclass	java.lang
@Deprecated	All annotable items	Marks item as deprecated	java.lang
@SuppressWarnings	All annotable items except packages and annotations	Suppress warning of given type	java.lang
Java Annotations Applied to Other Annotations			
@Retention	Annotations	Specifies how long this annotation is retained - whether in code only, compiled into the class, or available at run time through reflection.	java.lang.annotation
@Documented	Annotations	Specifies that this annotation should be included in the documentation of annotated items	java.lang.annotation

@Target	Annotations	Specifies the items to which this annotation can be applied	java.lang.annotation
@Inherited	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses.	java.lang.annotation

@Override:

@**Override** is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded

Example:

```
public class Animal {

   public void makeSound(){

   }
}

class Cat extends Animal{

   @Override
   public void makeSound(){
      System.out.println("myyyyyaaawwwwww");
   }
}
```

@Deprecated

Use this annotation on methods or classes which you ***need to mark as deprecated***. Any class that will try to use this deprecated class or method, will get a compiler "**warning**".

```
@Deprecated public Integer myMethod()

{

   return null;
```

```
}
```

@SuppressWarnings

This annotation ***instructs the compiler to suppress the compile time warnings*** specified in the annotation parameters. e.g. to ignore the warnings of unused class attributes and methods use @SuppressWarnings("unused") either for a given attribute or at class level for all the unused attributes and unused methods.

```
@SuppressWarnings("unused")
public class DemoClass
{
    //@SuppressWarnings("unused")
    private String str = null;
      //@SuppressWarnings("unused")
    private String getString(){
      return this.str;
    }
}
```

@Target Java Annotation

- While defining a custom Java annotation we have to specify which element (class, method, field, constructor etc.) this newly defined annotation would be applicable on. The @Target annotation is used for that purpose to set the target elements on which the custom annotation can be applied
- The possible values of elements for @Target annotation. They belong to the enumerated type ElementType.

Element Type	Annotation Applies To
ANNOTATION_TYPE	Annotation type declarations
PACKAGE	Packages
TYPE	Classes (including enum) and interfaces (including annotation types)
METHOD	Methods

CONSTRUCTOR	Constructors
FIELD	Fields (including enum constants)
PARAMETER	Method or constructor parameters
LOCAL_VARIABLE	Local variables

example:

```
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
  }
}
```

@Retention Java Annotation

- As name suggets, @Retention meta annotation specifies till what level an annotation will be retained. To decide the scope of the custom annotation we have to specify one of the three values (SOURCE, CLASS, or RUNTIME) of RetentionPolicy. The default is RetentionPolicy.CLASS.
    - RetentionPolicy.SOURCE specifies the scope of custom annotation to the compile time. Annotations having retention policy RetentionPolicy.SOURCE are not included in bytecode.
    - Annotations those are carrying RetentionPolicy.CLASS policy of retention are included in .class files, but the virtual machine need not to load them.
    - Annotations having RetentionPolicy.RUNTIME policy are included in class files and loaded by the virtual machine. Java annotations that are given RUNTIME retention policy can be accessed at run time through the reflection API.

example:

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

//@Retention(RetentionPolicy.CLASS)
@Retention(RetentionPolicy.RUNTIME)
//@Retention(RetentionPolicy.SOURCE)
public @interface MyCustomAnnotation
```

```
{
   //some code
}
```

@Documented Java Annotation

The @Documented meta annotation hints Javadoc tool to include this annotation in the documentation wherever it is used. Documented Java annotations should be treated just like other modifiers, such as protected or static, for documentation purposes. The use of other annotations is not included in the documentation.

Example:

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

@Inherited Java Annotation

The @Inherited annotation can be applied only to annotations for classes. When a superclass is annotated with an @Inherited Java annotation then all of its subclasses automatically have the same annotation.

Example:

```
java.lang.annotation.Inherited

@Inherited
public @interface MyCustomAnnotation {

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