Annotations

- Introduction
- Comment Vs Annotations
- XML Vs Annotations
- Types of Annotations
- Standard Annotations
- Custom Annotations

Annotation:

Annotation is a Java Feature provided by JDK 5.0 version, it can be used to represent metadata in Java applications.

Q) In Java Applications, to describe Metadata we have already Comments then what is the Requirement to go for Annotations?

- **In Java applications, if we provide metadata by using comments then "Lexical Analyzer" will remove comments metadata from Java program as part of Compilation.**
- **As per the application requirement, if we want to bring metadata up to .java file, up to .class file and up to RUNTIME of our application then we have to use "Annotations".**

Q) In Java Applications, to provide Metadata at Runtime of Java Applications we are able to use XML Documents then what is the Requirement to go for "Annotations"?

In Java applications, if we provide metadata by using XML documents then we are able to get the following problems.

- **Every time we have to check whether XML documents are located properly or not.**
- **Every time we have to check whether XML documents are formatted properly or not.**
- **Every time we have to check whether we are using right parsing mechanism or not to access the data from XML document.**
- **First Developers must aware XML tech.**

To overcome all the above problems, we have to use Java alternative that is "Annotation".

EX: Servlet Configuration with XML Document

web.xml

```
<web-app>

    <servlet>
        <servlet-name>ls</servlet-name>
        <servlet-class>LoginServlet</servlet-name>
    </servlet>
```

```
<servlet-mapping>
        <servlet-name>ls</servlet-name>
        <url-pattern>/login</url-pattern>
    </servlet-mapping>

</web-app>
```

Servlet Configuration with Annotation

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
}
```

XML-Based Tech	Annotation Based Tech
JDK 1.4	JDK 5.0
JDBC 3.0	JDBC 4.0
Servlets 2.5	Servlets 3.0
Struts 1.x	Struts 2.x
JSF 1.x	JSF 2.x
Hibernate 3.2.4	Hibernate 3.5
EJBs 2.x	EJBs 3.x
Spring 2.x	Spring 3.x

NOTE: Annotations are not the complete alternative for XML tech, with in a Java application we can use annotations as an alternative for XML documents but when we go for distributed applications where applications are designed on different tech standards there annotations are not possible, only we have to use XML tech.

To process the annotations, Java has provided a predefined tool in the form of "Annotation Processing tool"[APT}, it was managed by Java up to JAVA 7 version, it was removed from Java in JAVA8 version.

Syntax to declare Annotation:

```
@interface Annotation_Name {
        data_Type member_Name() [default] value;
}
```

Syntax to use Annotation:

@Annotation_Name(member_name1=value1,member_Name2=value2.........)
In Java, all the annotations are interfaces by default.
In Java, the common and default super type for all the annotations is "java.lang.annotation.Annotation".
As per the number of members inside annotations, there are three types of annotations.

- **Marker Annotations:**

  It is an annotation without members.

  EX:
  ```
  @interface Override {
  }
  ```

- **Single-Valued Annotation:**

  It is an Annotation with exactly one single member

  EX:
  ```
  @interface SuppressWarnings {
          String value();
  }
  ```

- **Multi-Valued Annotation:**

  It is an annotation with more than one member.

  EX:
  ```
  @interface WebServlet {
          int loadOnStartup();
          String[] urlPatterns();
          -----
  }
  ```

In Java Annotations are divided into the following two types as per their nature.

- Standard Annotations
- Custom Annotations

- **Standard Annotations:**

  These are predefined Annotations provided by Java along with Java software.
  There are two types of Standard Annotations

  - General Purpose Annotations
  - Meta Annotations

  - **General Purpose Annotations:**

    These Annotations are commonly used Annotations in Java applications
    These Annotations are provided by Java as part of java.lang package.

    EX:
    ```
    @Override
    @Deprecated
    @SuppressWarnings
    @FunctionalInterface [JAVA 8]
    ```

Meta Annotations:

These Annotations can be used to define another Annotations.
These Annotations are provided by Java as part of java.lang.annotation package.
EX: @Inherited
@Documented
@Target
@Retention

@Override:

- In Java applications if we want to perform method overriding then we have to provide a method in subclass with the same prototype of super class method.
- While performing method overriding, if we are not providing the same super class method at sub class method then compiler will not rise any error and JVM will not rise any exception, JVM will provide super class method output.
- In the above context, if we want to get an error about to describe failure case of method overriding from compiler then we have to use @Override

EX:

```
class JdbcApp {
        public void getDriver() {
                System.out.println("Type-1 Driver");
        }
}
class New_JdbcApp extends JdbcApp {
        @Override
        public void getdriver() {
                        System.out.println("Type-4 Driver");
        }
}
class Test {
      public static void main(String args[]) {
            JdbcApp app = new New_JdbcApp();
            app.getDriver();
      }
}
```

If we compile the above program then compiler will rise an error like "method does not override or implement a method from a SuperType".

NOTE: If we compile the above programme, compiler will recognize @Override annotation, compiler will take sub class method name which is marked with @Override annotation, compiler will compare sub class method name with all the super class method names. If any method name is matched then compiler will not rise any error.

If no super class method name is matched with sub class method name then compiler will rise an error.

• @Deprecated:

The main intention of this annotation is to make a method as deprecated method and to provide deprecation message when we access that deprecated method.
In general, Java has provided Deprecation support for only predefined methods, if we want to get deprecation support for the user defined methods we have to use @Deprecated annotation.

NOTE: Deprecated method is an outdated method introduced in the initial versions of Java and having alternative methods in the later versions of Java.

```
class Employee {
      @Deprecated
       public void gen_Salary(int basic, float hq) {
           System.out.println("Salary is calculated on the basis of basic amount,hq");
       }
       public void gen_Salary(int basic, float hq, int ta, float pf) {
   System.out.println("Salary is calculated on the basis of basic amount,hq,ta,pf");
       }
}
class Test {
        public static void main(String args[]) {
              Empoloyee emp = new Employee();
              emp.gen_Salary(25000, 20.0f);
        }
}
```

Note: If we compile the above code then compiler will provide the following deprecation message.

Note: Java uses or overrides a deprecated API"

• @SuppressWarnings(--):

In Java applications, when we perform unchecked or unsafe operations then compiler will rise some warning messages. In this context,to remove the compiler generated warning messages we have to use @SuppressWarnings("unchecked") annotation.

EX:

```
import java.util.*;
class Bank {
   @SuppressWarnings("unchecked")
   public ArrayList listCustomers() {
   ArrayList al = new ArrayList();
   al.add("chaitu");
   al.add("Mahesh");
   al.add("Jr NTR");
   al.add("Pavan");
   return al;
   }
}
class Test {
    public static void main(String args[]) {
              Bank b = new Bank();
              List l = b.listCustomers();
              System.out.println(l);
    }
}
```

@FunctionalInterface:

- **In Java, if we provide any interface without abstract methods then that interface is called as "Marker Interface".**
- **In Java, if we provide any interface with exactly one abstract method then that interface is called as "Functional Interface".**
- **To make any interface as Functional Interface and to allow exactly one abstract method in any interface then we have to use "@FunctionalInterface" annotation.**
- **NOTE: This annotation is a new annotation provided by JAVA8 version.**

```
@FunctionalInterface
interface Loan {
     void getLoan();
}
class GoldLoan implements Loan {
       public void getLoan() {
              System.out.println("GoldLoan");
       }
}
```

```
class Test {
    public static void main(String args[]) {
        Loan l = new GoldLoan();
        l.getLoan();
    }
}
```

@Inherited:

In general all the annotations are not inheritable by default.
If we want to make/prepare any annotation as Inheritable annotation then we have to declare that annotation as Inheritable annotation then we have to declare that annotation with @Inherited annotation.

EX:

```
@interface Persistable
{
}
@Persistable
class Employee
{
}
class Manager extends Employee{
}
```

In the above example, only Employee class objects are Persistable i.e eligible to store in database.

```
@Inherited
@interface Persistable
{
}
@Persistable
class Employee {
}
class Manager extends Employee {
}
```

NOTE: In the above example, both Employee class objects and manager class objects are Persistable i.e eligible to store in database.

• @Documented:

- In Java, by default all the annotations are not documentable.
- In Java applications, if we want to make any annotation as documentable annotation then we have to prepare the respective annotation with @Documented annotation.

EX:

```
@interface Persistable
{
}
@Persistable
class Employee
{
}
javadoc Employee.java
```

If we prepare html documentation for Employee class by using "javadoc" tool then @Persistable annotation is not listed in the html documentation.

```
@Documented
@interface Persistable
{
}
@Persistable
class Employee
{
}
```

If we prepare html documentation for Employee class by using "javadoc" tool then @Persistable annotation will be listed in the html documentation.

• @Target:

The main intention of this annotation is to define a list of target elements to which we are applying the respective annotation.

Synatx: @Target(--list of constants from ElementType enum---)
Where ElementType enum contains FIELD, CONSTRUCTOR, METHOD,
TYPE [Classes, abstract classes, interfaces]

EX:

```
@Target(ElementType.TYPE, ElementType.FIELD, ElementType.METHOD)
@interface persistable
{
}
@Persistable
```

```
class Employee
{
    @Persistable
    Account acc;

    @Persistable
    public Address getAddress(){}
}
```

@Retention:

The main intention of the annotation is to define life time of the respective annotation in Java application.

Syntax: @Retention(---a constant from RestentionPolicy enum---)
Where RestentionPolicy enum contains the constants like SOURCE, CLASS, RUNTIME

```
@Retention(RetentionPolicy.RUNTIME)
@interface Persistable
{
}
@Persistable
class Employee
{
}
```

Custom Annotations:

These are the annotations defined by the developers as per their application requirements.
To use custom annotations in java applications, we have to use the following steps.

- **Declare User defined Annotation:**

Bank.java

```
import java.lang.annotation.*;
@Inherited
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Bank {
     String name() default "ICICI Bank";
     String branch() default "S R Nagar";
     String phone() default "040-123456";
}
```

11) Utilize User defined Annotations in Java Applications:

Account.java

```
@Bank(name="Axis Bank",phone="040-987654")
public class Account {
        String accNo;
        String accName;
        String accType;
        public Account(String accNo, String accName, String accType) {
              this.accNo = accNo;
              this.accName = accName;
              this.accType = accType;
        }
        public void getAccountDetails() {
                  System.out.println("Account Details");
                  System.out.println("----------------");
                  System.out.println("Account Number:"+accNo);
                  System.out.println("Account Name :"+accName);
                  System.out.println("Account Type :"+accType);
        }
}
```

Access the Data from User-Defined Annotation in Main Application:

- **If the Annotation is Class Level Annotation then use the following Steps**
  - Get java.lang.Class object of the respective class
  - Get Annotation object by using getAnnoataion(Class c) method

- **If the annotation is Field level annotation then use the following steps:**
  - Get java.lang.Class object of the respective class
  - Get java.lang.reflect.Field class object of the respective variable from java.lang.Class by using getField(String field_name) method
  - Get Annotation object by using getAnnotation(Class c) method from java.lang.reflect.Field class.

- **If the Annotation is Method Level Annotation then use the following Steps**
  - Get java.lang.Class object of the respective class
  - Get java.lang.reflect.Method class object of the respective Method from java.lang.Class by using getMethod(String method_name) method
  - Get Annotation object by using getAnnotation(Class c) method from java.lang.reflect.Method class.

MainApp.java

```java
import java.lang.annotation.*;
import java.lang.reflect.*;
public class MainApp {
      public static void main(String args[]) throws Exception {
              Account acc = new Account("abc123","Durga","Hyd");
              acc.getAccountDetails();
              System.out.println();
              Class c = acc.getClass();
              Annotation ann = c.getAnnotation(Bank.class);
              Bank b = (Bank)ann;
              System.out.println("Bank Details");
              System.out.println("------------");
              System.out.println("Bank Name  :"+b.name());
              System.out.println("Branch Name :"+b.branch());
              System.out.println("Phone   "+b.phone());
      }
}
```

NOTE: class Level Annotation

Method Level Annotation

Course.java

```java
import java.lang.annotation.*;
@Inherited
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface Course {
        String cid() default "C-111";
        String cname() default "Java";
        int ccost() default 5000;
}
```

Student.java

```java
public class Student {
        String sid;
        String sname;
        String saddr;
        public Student(String sid, String sname, String saddr) {
                this.sid = sid;
                this.sname = sname;
                this.saddr = saddr;
        }
        @Course(cid = "C-222", cname = ".NET")
        public void getStudentDetails() {
                System.out.println("Student Details");
                System.out.println("---------------");
                System.out.println("Student Id  :"+sid);
                System.out.println("Sudent Name  :"+sname);
                System.out.println("Student Address:"+saddr);
        }
}
```

ClientApp.java

```java
import java.lang.annotation.*;
import java.lang.reflect.*;
public class ClientApp {
        public static void main(String args[]) throws Exception {
                Student std = new Student("S-111", "Durga", "Hyd");
                std.getStudentDetails();
                System.out.println();
                Class cl = std.getClass();
                Method m = cl.getMethod("getStudentDetails");
                Annotation ann = m.getAnnotation(Course.clas);
                Course c = (Course)ann;
                System.out.println("Course Details");
                System.out.println("---------------");
                System.out.println("Course Id  :"+c.cid());
                System.out.println("Course Name :"+c.cname());
                System.out.println("Course Cost  :"+c.ccost());
        }
}
```