# Module 3

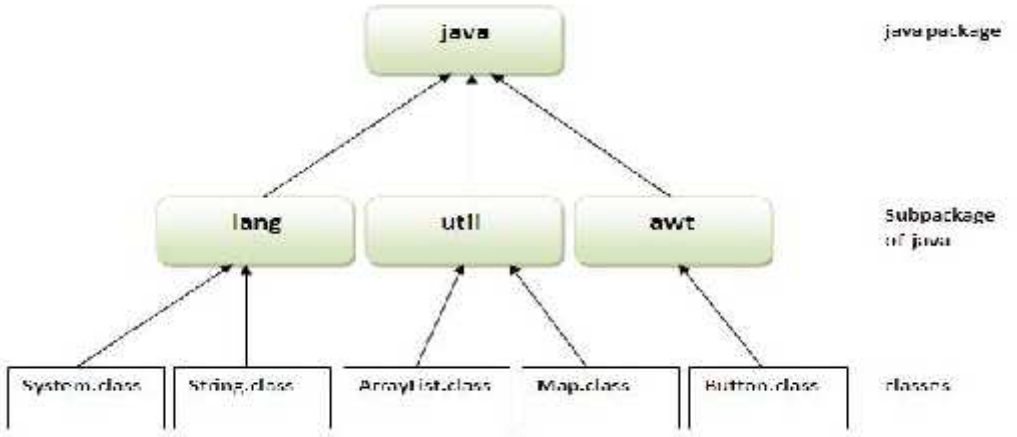
Creating Packages, Interfaces, JAR files and Annotations. The core java API package, New java.Lang Sub package, Built-in Annotations with examples. Working with Java Beans. Introspection, Customizers, creating java bean, manifest file, Bean Jar file, new bean, adding controls, Bean properties, Simple properties, Design Pattern events, creating bound properties, Bean Methods, Bean an Icon, Bean info class, Persistence, Java Beans API.

## 3.1 Creating Packages

- **Packages**
  - A java package is a collection of classes, interfaces, sub-packages, exceptions, enums, etc.
  - Package in java can be categorized in two form, built-in package and user-defined package.
- **Advantages**
  - Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  - Java package provides access protection.
  - Java package removes naming collision.

### 3.1.1 Types of Packages in java

- There are two types of packages in java
  - Built-in package
  - User-defined package
- **Built-in package**
  - Existing Java package for example `java.lang, java.util` etc



- User- defined package
  - Java package created by user to categorize their project's classes and Interface.
  - Below are the steps to create user-defined package
    - Creating a package
    - Compiling a package
    - Import a package

### 3.1.2 Creating a package

- Include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
public class employee
{
    statement;
}
```

- The above statement will create a package with name **mypack** in the project directory.
- Java uses file system directories to store packages. For example, the .java file for any class you define to be part of **mypack** package must be stored in a **directory** called **mypack**.
- A package is always defined as a separate folder having the same name as the package name
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line
- All classes of the package must be compiled before use (So that they are error free)

- Example

```
//save as FirstProgram.java
package learnjava;
public class FirstProgram{
    public static void main(String args[]) {
        System.out.println("Welcome to package");
    }
}
```

- **Compiling java programs inside packages**
  - This is just like compiling a normal java program

```
Syntax: javac -d directory javafilename

Example: javac -d . FirstProgram.java
```

  - The `-d` switch specifies the destination where to put the generated class file.
  - If you want to keep the package within the same directory, you can use **.** (dot).

- **Importing a package**
  - **import** keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.
  - The `import` keyword is used to make the classes and interface of another package accessible to the current package.
  - If you import `packagename.classname` then only the class with name `classname` in the package with name `packagename` will be available for use.

```
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
import pack.A;
class B {
    public static void main(String args[]) {
        A obj = new A();
        obj.msg();
    }
}
```

  - If you use `packagename.*,` then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use.

```
//save by First.java
package learnjava;
public class First{
    public void msg() {
        System.out.println("Hello");
    }
}

//save by Second.java
import learnjava.*;
class Second {
    public static void main(String args[]) {
        First obj = new First();
        obj.msg();
    }
}
```

### 3.1.3 Creating Packages that have sub package

- A subpackage is just a package in a subdirectory of another package's directory. Java uses dot as the package name separator
- Suppose we have a file called "HelloWorld.java". and want to store it in a subpackage "**subpackage**", which stays inside package "**package**".
- The "HelloWorld" class should look something like this:

```
package package.subpackage;

public class HelloWorld {
  public void show(){
      System.out.println("Hello World”);
  }
```

## 3.2 Interfaces

- Interface is a pure abstract class.They are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body.
- Interface is used to achieve complete **abstraction** in Java.
- When you create an interface it defines what a class can do without saying anything about how the class will do it.

- **Syntax:**

```
interface interface_name { }
```

- **Example**

```
interface Moveable
{
     int AVERAGE-SPEED=40;
     void move();
}
```

- **Note**
  - Complier automatically converts methods of interface as public and abstract, and the data member as public, static and final by default.
  - Methods inside Interface must not be static, final.
  - All variables declared inside interface are implicitly public static final variables(constants).
  - All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
  - Interface can extend one or more other interface.
  - Interface cannot implement a class.
  - Interface can be nested inside another interface.

- **Example of Interface implementation**

```
interface Moveable
{
 int AVG-SPEED = 40;
 void move();
}

class Vehicle implements Moveable
{
 public void move()
 {
  System .out. print in ("Average speed is"+AVG-SPEED");
 }
 public static void main (String[] arg)
 {
  Vehicle vc = new Vehicle();
  vc.move();
 }
}

Output:
Average speed is 40
```

- **Interfaces supports Multiple inheritance**
  - Though classes in java doesn't support multiple inheritance, but a class can implement more than one interface.

```
interface Moveable
{
 boolean isMoveable();
}
interface Rollable
{
 boolean isRollable
}
class Tyre implements Moveable, Rollable
{
 int width;
 boolean isMoveable()
 {
  return true;
 }
```

```
 boolean isRollable()
 {
  return true;
 }
 public static void main(String args[])
 {
  Tyre tr=new Tyre();
  System.out.println(tr.isMoveable());
  System.out.println(tr.isRollable());
 }
}

Output:
true
true
```

- **Interface extends other Interface**
  - Classes implements interfaces, but an interface extends other interface.

```
interface NewsPaper
{
 news();
}

interface Magazine extends NewsPaper
{
 colorful();
}
```

- **Difference between Abstract Class and Interface**

| **Abstract Class** | **Interface** |
|---|---|
| Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes. | Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods |
| Abstract class is a Class prefix with an abstract keyword followed by Class definition. | Interface is a pure abstract class which starts with interface keyword. |
| Abstract class can also contain concrete methods. | Whereas, Interface contains all abstract methods and final variable declarations. |
| Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes. | Interfaces are useful in a situation that all properties should be implemented |

## 3.3 JAR files

- The Java Archive (JAR) file format enables you to *bundle multiple* files into a single archive file.
- Typically a JAR file **contains**
  - the class files
  - auxiliary resources (applets, images, audio)
- An **archive file** is a file that is **composed** of one or more computer files along with metadata.
- Archive files are used to collect multiple data files together into a single file for **easier portability** and **storage**.
- It can be digitally sign the jar file to prove their origin.

### 3.3.1 Creating a jar file

- Basic command for creating a JAR file:

```
jar cf jar-file input-file(s)
```

- Options

  - ***c***: create a JAR file
  - ***f***: output jar file
  - ***v***: produce verbose output
  - ***jar-file***: the name of the resulting JAR file
    - can use any filename for a JAR file
    - by convention, JAR filenames are given a .jar extension
  - ***input-file(s)***: a space-separated list of one or more files to be placed in the JAR file.
    - can contain the wildcard * symbol
    - can be directory names; directories are added recursively

### 3.3.2 Viewing a JAR file

- Basic command for viewing the contents of a JAR file**:**

```
jar tf jar-file
```

- **Options**

Options

- ***t*** : show table of contents
- ***f*** : jar file
- ***v*** : produce additional information like file sizes and dates

This command will display the table of contents of the JAR file to stdout

### 3.3.3 Extracting Files from JAR File

- To extract the files from a .jar file

```
Syntax: Jar xf jarfile.jar
```

- Unpacking all the files in a JAR file will create directory structure

**Example:**

```
C:\jar xf pack.jar
```

This will create the following directories in C:\

META-INF

pack // in this directory , we can see class1.class and class2.class.

### 3.3.4 Updating a JAR File

- The Jar tool provides a 'u' option which you can use to update the contents of an existing JAR file by modifying its manifest or by adding files. The basic command for adding files has this format:

```
jar uf jar-file input-file(s)
```

**More on JAR**

**Options for Creating JAR**

| Options | Descriptions |
|---|---|
| -c | creates new archive file |
| -v | generates verbose output. It displays the included or extracted resource on the standard output |
| -m | includes manifest information from the given mf file. |
| -f | specifies the archive file name |
| -x | extracts files from the archive file |
| -t | To view the contents of a JAR file |

To perform basic tasks with JAR files, you use the Java Archive Tool provided as part of the Java Development Kit (JDK).

| Operation | Command |
|---|---|
| To create a JAR file | jar cf jar-file input-file(s) |
| To view the contents of a JAR file | jar tf jar-file |
| To extract the contents of a JAR file | jar xf jar-file |
| To extract specific files from a JAR file | jar xf jar-file archived-file(s) |
| To run an application packaged as a JAR file (requires the Main-class manifest header) | java -jar app.jar |
| To invoke an applet packaged as a JAR file | <applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height></applet> |

## 3.4 Annotations

- Java annotations is a tag that represents metadata.
- i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Java annotations were added to Java from Java 5.
- Annotations start with '@'

### 3.4.1 Categories of Annotation

- There are three categories of annotations
- **Marker Annotation**
  - Marker Annotation are annotation without any methods.
  - The purpose is to mark declaration
  - **@Override** is an example of Marker Annotation.

- **Single valued Annotation**
  - These annotations contain only one member specifying the value of the member.
  - Only value has to specified, not required to specify name of the member
  - Example: - @SuppressWarnings(value);

- **Multivalued Annotation**
  - These annotations consist of multiple data members/ name, value, pairs.
  - Example: - @WebServlet(attribute1 =value1, atrribute2=value2, attribute3=value3)

## 3.4.2 Predefined/ Standard Annotations/Built-in Annotations

- imported from java.lang.annotation
  - **@Retention**
  - **@Documented**
  - **@Target**
  - **@Inherited**.
- included in java.lang
  - **@Deprecated**
  - **@Override**
  - **@SuppressWarnings**

- **@Deprecated Annotation**
  - It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
  - The Javadoc @deprecated tag should be used when an element has been deprecated
  - @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.
  - **Example:**

```
public class Test
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecated test display()");
    }

    public static void main(String args[])
    {
        Test d = new Test();
        d.Display();
    }
}
```

Output:

Deprecatedtest display()

- **@Override**
  - It is a marker annotation that can be used only on methods.
  - A method annotated with **@Override** must override a method from a superclass.
  - It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

```
class Animal{
void eatSomething(){
          System.out.println("eating something");
     }
}

class Dog extends Animal{
@Override
void eatSomething(){
     System.out.println("eating foods");
     }
}

class TestAnnotation1 {
     public static void main(String args[]){
          Animal a=new Dog();
          a.eatSomething();
     }
}
Output:
eating foods
```

- **@SuppressWarnings**
    - It is used to inform the compiler to suppress specified compiler warnings
    - The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.
    - Java groups warnings under two categories. They are **deprecation** and **unchecked.**. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

```
import java.util.*;
class TestAnnotation2{

@SuppressWarnings("unchecked")
public static void main(String args[]){
     ArrayList list=new ArrayList();

     list.add("anil");
     list.add("vimal");
     list.add("ratan");

for(Object obj:list)
```

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

**Output:**

anil
vimal
ratan

- **@Retention Annotation**
  - It determines where and how long the annotation is retent.
  - The 3 values that the @Retention annotation can have:
    - **SOURCE:** Annotations will be retained at the source level and ignored by the compiler.
    - **CLASS:** Annotations will be retained at compile time and ignored by the JVM.
    - **RUNTIME:** These will be retained at runtime.
- **@Documented**
  - It is a marker interface that tells a tool that an annotation is to be documented.
  - It is designed to be used only as an annotation to an annotation declaration
- **@Target**
  - It is designed to be used only as an annotation to another annotation.
  - **@Target** takes one argument, which must be constant from the **ElementType** enumeration.
  - This argument specifies the type of declarations to which the annotation can be applied.
  - The constants are shown below along with the type of declaration to which they correspond.

| **Target Constant** | **Annotations Can be applied for** |
|---|---|
| ANNOTATION_TYPE | Another annotation |
| CONSTRUCTOR | Constructor |
| FIELD | Field |
| LOCAL_VARIABLE | Local Variable |
| METHOD | Method |
| PACKAGE | Package |
| PARAMETER | Parameter |
| TYPE | Class, Interface, or enumeration |

- **@Inherited**
  - @Inherited is a marker annotation that can be used only on annotation declaration.
  - It affects only annotations that will be used on class declarations.
  - **@Inherited** causes the annotation for a superclass to be inherited by a subclass.
  - Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that

annotation is present in the superclass, and if it is annotated with **@Inherited,** then that annotation will be returned.

## User defined/ Custom Annotation

- User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc. These annotations can be applied just before declaration of an element (constructor, method, classes, etc).
- **Syntax of Declaration:**

```
[Access Specifier]@interface<AnnotationName>
{
   DataType <Method Name>() [default value];
}
```

- **AnnotationName** is an identifier.
- Parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
- Parameters will not have a null value but can have a default value.
- ***default value*** is optional.
- Return type of method should be either primitive, enum, string, class name or array of primitive, enum, string or class name type.

## 3.5 The core java API package

- Java **APl(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality.

| Package | Description |
|---|---|
| java.lang | Language support classes. They include classes for primitive types, string, math functions, thread and exceptions. |
| java.util | Language utility classes such as vectors, hash tables, random numbers, data, etc. |
| java.io | Input/output support classes. They provide facilities for the input and output of data. |
| java.applet | Classes for creating and implementing applets |
| java.net | Classes for networking. They include classes for communicating with local computers as well as with internet servers. |
| java.awt | Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on |

### 3.6 Working with Java Beans

- **Java Bean**
  - Beans are simply *Java classes* that are written in a *standard format*( javaBeans API).
  - Java bean component are known as beans.

- **Characteristics**
  - It provides a default, no-argument **constructor.**
  - It should be **serializable** and implement the java.io.**Serializable** interface.
  - It may have a number of **"getter"** and **"setter"** methods for the properties.
  - Class must *not define* any **public** instance variables.

**Creating Java Bean**

```
class MyBean implements Serializable {
      private String name;
      MyBean() //default constructor
      {
      }
      public void setname(String name){
            this.name=name;
      }
      public void getname(){
            return name;
      }
}
```

- **Advantages**
  - A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
  - The properties, events, and methods of a Bean that are exposed to another application can be controlled.
  - The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
  - A Bean may register to receive events from other objects and can generate events that are sent to other objects

- **Key features of bean**
  - **Introspection** -discover's a bean
  - **Properties** - appearance and behavior characteristics of a bean
  - **Events**- as a simple communication metaphor than can be used to connect up beans
  - **Persistence** - enables beans to save and restore their state
  - **Customization** - to allow the customization of the appearance and behavior of a bean.

## Introspection

- Introspection can be defined as the technique of obtaining information about bean properties, events and methods.
- Basically introspection means analysis of bean capabilities.
- Beans support introspection in two ways:
    - By adhering to specific rules, known as design patterns, when naming bean features.
    - By explicitly providing property, method, and event information with a related bean information class.

## Properties

- A property is a subset of a Bean's state.
- A bean property is a named attribute of a bean that can affect its behavior or appearance.
- Examples of bean properties include color, label, font, font size, and display size.
- Types of JavaBeans Properties
    - Simple properties
    - Indexed properties
    - Bound properties
    - Constraints properties

**Simple properties**

- A bean property refers to private variables with a single value whose changes are independent of changes in any other property.
- Simple properties are retrieved and specified using get and set methods respectively.
- A read/write property has both of these methods to access its values.
- The **get** method used to read the value of the property. The **set** method that sets the value of the property.
- The ***setXXX***() and ***getXXX()*** methods are the heart of the java beans properties mechanism. This is also called getters and setters.

**Syntax:**

```
public return_type get()<PropertyName>()
public void set<PropertyName>(data_type value)
```

**Example:**

```
public double getDepth() {
     return depth;
     }

public void setDepth(double d) {
     Depth=d;
}
```

**Indexed Properties**

- A bean property that supports a range of values instead of a single value.
- Indexed Properties are consists of multiple values.
- If a simple property can hold an array of value they are no longer called simple but instead indexed properties.

**Syntax:**

```
public int[] get()
public property_datatype get(int index)
```

**Example:**

```
private double data[];
public double getData(int index)
{return data[index];}

public void setData(int index,double value) {
Data[index]=value; }
```

**Bound Properties**

- A bean property for which a change to the property results in a notification being sent to some other bean.
- Bound Properties are the properties of a JavaBean that inform its listeners about changes in its values.
- Bound Properties are implemented using the **PropertyChangeSupport** class and its methods.

**Constraint Properties**

- It generates an event when an attempt is made to change it value.
- Constrained Properties are implemented using the **PropertyChangeEvent** class.
- The event is sent to objects that previously registered an interest in receiving an such notification.
- A bean property for which a change to the property results in validation by another bean. The other bean may reject the change if it is not appropriate.

**Design Pattern Events**

- Enables Beans to communicate and connect together.
- Beans generate events and these events can be sent to other objects. Event means any activity that interrupts the current ongoing activity.
- Example: mouse clicks, pressing key
- The event model that is used by the JavaBeans architecture is a delegation model. This model is composed of three main parts: *sources, events, and listeners*
    - *Event Source*: Generates the event and informs all the event listeners that are registered with it.
    - *Event Listener*: Receives the notification, when an event source generates an event.
    - *Event Object*: Represents the various types of events that can be generated by the event sources.

## Persistence

- Persistence enables beans to save and restore their state through object serialization.
- Object serialization means converting an object into a data stream and writing it to storage.
- Persistence means an ability to save properties and events of our beans to non-volatile storage and retrieve later.
- It has the ability to save a bean to storage and retrieve it at a later time configuration settings are saved It is implemented by Java serialization.

## Customizers

- *Customization* provides a means for modifying the appearance and behavior of a bean within an application builder so it meets your specific needs.
- There are several levels of customization available for a bean developer to allow other developers to get maximum benefit from a bean's potential functionality.
- A bean's appearance and behavior can be customized at design time within beans-compliant builder tools. There are two ways to customize a bean:
    - By using a ***property editor***. Each bean property has its own property editor. The NetBeans GUI Builder usually displays a bean's property editors in the Properties window. The property editor that is associated with a particular property type edits that property type.
    - By using ***customizers***. Customizers give you complete GUI control over bean customization. Customizers are used where property editors are not practical or

applicable. Unlike a property editor, which is associated with a property, a customizer is associated with a bean.

## Manifest file

- The manifest file for a JavaBean application contains a list of all the class files that make up a JavaBean
- The entry in the manifest file enables the target application to recognize the JavaBean classes for an application
- For example, the entry for the MyBean JavaBean in the manifest file is as shown:

```
Manifest-Version: 1.0
Name: MyBean.class
Java-Bean: true
```

Note: write that 2 lines code in the notepad and save that file as MyBean.mf

## Bean jar file

- Syntax for creating jar file using manifest file

```
C:\Beans >jar cfm MyBean.jar MyBean.mf MyBean.class
```

- m indicates that the manifest file

## Bean an Icon

- A bean implementor who wishes to provide explicit information about their bean may provide a BeanInfo class that implements this BeanInfo interface and provides explicit information about the methods, properties, events, etc, of their bean.

## Bean info class

- The BeanInfo interface provides the methods that enable you to specify and retrieve the information about a JavaBean.
- A BeanInfo class has to be derived from the SimpleBeanInfo class and its name has to start with the name of the associated JavaBean.

## Java Beans API

- The Java Beans functionality is provided by a set of classes and interfaces in the java.beans package.
- lists the classes and interface in java.beans.

| **Interface Summary** | |
|---|---|
| AppletInitializer | This interface is designed to work in collusion with java.beans.Beans.instantiate. |
| BeanInfo | A bean implementor who wishes to provide explicit information about their bean may provide a BeanInfo class that implements this BeanInfo interface and provides explicit information about the methods, properties, events, etc, of their bean. |
| Customizer | A customizer class provides a complete custom GUI for customizing a target Java Bean. |
| PropertyChangeListener | A "PropertyChange" event gets fired whenever a bean changes a "bound" property |
| PropertyEditor | A PropertyEditor class provides support for GUIs that want to allow users to edit a property value of a given type. |
| **Class summary** | |
| BeanDescriptor | A BeanDescriptor provides global information about a "bean", including its Java class, its displayName, etc. |
| Beans | This class provides some general purpose beans control methods |