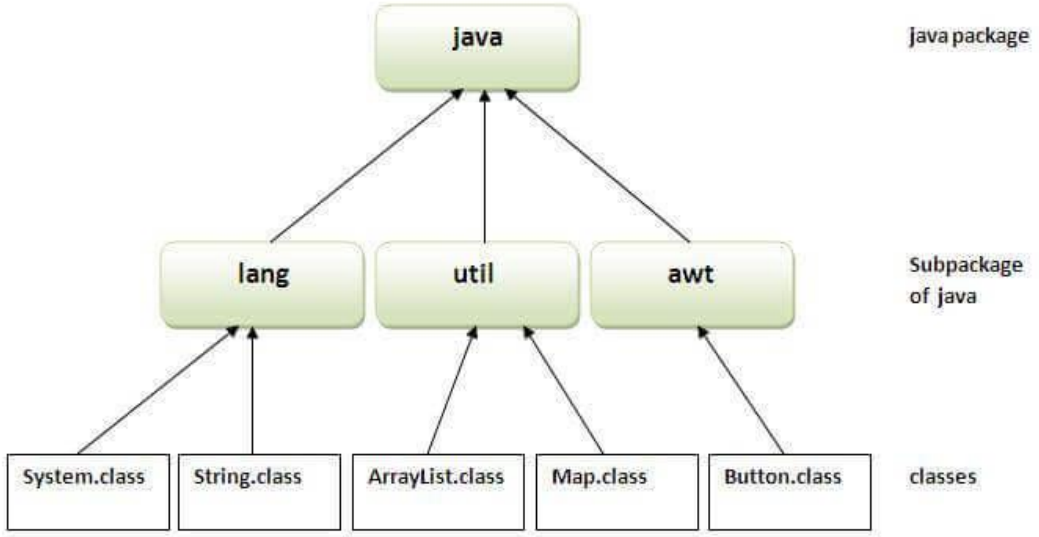
Lecture – 6

Encapsulation

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
  - **Built-in Packages** (java, lang, awt, javax, swing, net, io, util)
  - **User defined Packages**
- The **package keyword** is used to create a package in java.



Example

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

How to compile java package

```
javac -d directory javafilename
```

Example

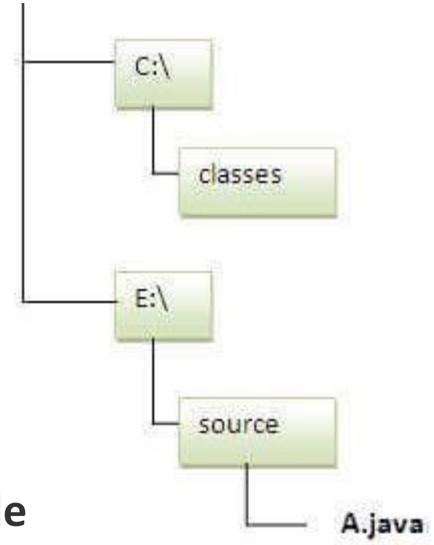
```
javac -d . Simple.java
```

- The -d switch specifies the destination where to put the generated class file.
- You can use any directory name like d:/abc (windows).
- If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

```
Java  mypack.Simple
```

How to send the class file to another directory or drive?



To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

Set the class path and then run

```
e:\sources> set classpath=c:\classes;
e:\sources> java mypack.Simple
```

OR

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Information Hiding

- **Information hiding** means that you separate the description of how to use a class from the implementation details,
  - Such as how the class methods are defined
  - Another term for information hiding is **abstraction**
  - To drive a car, do you need to know how the engine works? Why?
    - `println` method
      - need to know ***what*** the method does
      - but not ***how*** `println` does it
- Provide a more abstract view and hide the details

Defining Encapsulation

- **Encapsulation** means that the data and the actions are combined into a single item (in our case, a class object) and that the details of the implementation are hidden.
- ***Information hiding*** and ***Encapsulation*** are two sides of the same coin.
- If a class is well designed, a programmer who uses a class need not know all the details of the implementation of the class but need only know a much simpler description of how to use the class.

Controlling access to class members

- **Access Modifier**
  - Determines access rights for the class and its members
  - Defines where the class and its members can be used

Why use these

- It is important in many applications to hide data from the programmer
  - E.g., a password program must be able to read in a password and compare it to the current one or allow it to be changed
  - But the password should **never be accessed directly!**

```
public class Password {
        public String my_password;
                …
}
```

```
Password ProtectMe;
        …
ProtectMe.my_password = “backdoor”; // this is
bad
```

Access Modifiers

- Member modifiers change the way class members can be used
- *Access modifiers* describe how a member can be accessed

Modifier	Description
`(no modifier) / default`	The access level is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
`public`	The access level is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
`Protected`	The access level is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
`Private`	The access level is only within the class. It cannot be accessed from outside the class.

Access Modifiers

Access Modifier	Within class	Within Package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Encapsulating a Class

- Members of a class must always be declared with the minimum level of visibility.
- Provide *setters and getters* (also known as accessors/mutators) to allow *controlled* access to private data.
- Provide other public methods (known as *interfaces*) that other objects must adhere to in order to interact with the object.

Accessors and Mutators

- **Accessor methods**: Public methods that allow attributes (instance variables) to be read
  - Get methods are also commonly called accessor methods
  - Much better than making instance variables public
- **Mutator methods**: Public methods that allow attributes (instance variables) to be modified
  - Set methods are also commonly called mutator methods, because they typically change an object’s state

Setters and Getters

- Setters and Getters allow controlled access to class data
- *Setters* are methods that (only) alter the state of an object
  - Use setters to validate data before changing the object state
- *Getters* are methods that (only) return information about the state of an object
  - Use getters to format data before returning the object's state

Example

```
public class Person {
  private String name; // private = restricted access

  // Getter
  public String getName() {
    return name;
  }

  // Setter
  public void setName(String newName) {
    this.name = newName;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";  // error
    System.out.println(myObj.name); // error
  }
}
```

Correct Solution

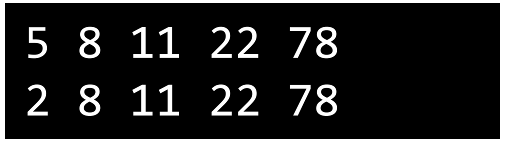
```
public class Main {
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.setName("John"); // Set the value of the name
variable to "John"
    System.out.println(myObj.getName());
  }
}
```

Using Reference of an Object Directly in a Setter

Inside the **'Array'** class we have a private array which is an **'element'**.
We have written a setElement() setter function in which values of one array are copied into the array declared inside the class.
So the **'element'** array values are set which is the same as the **'Arr'** array which is declared inside the main method.

```
class Array {
    private int[] element;

    //setter method to copy the value of a in element array
    void setElement(int[] a) {
        this.element = a;
    }

    //to print the value of the element array
    void display() {
        //using .length function
        //to find length of an array
        int len = (this.element).length;

        for(int i = 0; i < len; i++) {
            System.out.print(this.element[i] + " ");
        }
    }
}
```

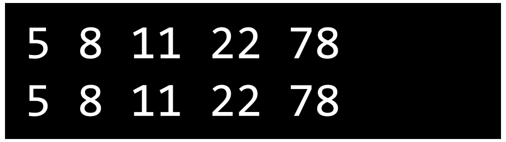
```
public class Main {
    public static void main(String[] args) {
        Array a1 = new Array();

        int Arr[] = {5, 8, 11, 22, 33};

        // calling the setter function
        a1.setElement(Arr);

        // calling the display function
        a1.display();

        // new value is set at the 0th index
        Arr[0] = 2;
        System.out.println();

        // calling the display function one more time
        a1.display();
    }
}
```



```
5  8  11  22  78
2  8  11  22  78
```

```
class Array {
    private int[] element;
     void setElement(int[] a) {
        int len2 = a.length;
        // dynamically allocating the memory to element[]
        // according to the a[] array length
        element = new int[len2];
        for(int i = 0; i < len2; i++) {
           // copying the value one by one
           // into the element array
            this.element[i] = a[i];
        }
    }
    // to print the value of the element array
    void display() {                    //using .length function
        //to find length of an array
        int len = (this.element).length;

        for(int i = 0; i < len; i++) {
            System.out.print(this.element[i] + " ");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Array a1 = new Array();

        int Arr[] = {5, 8, 11, 22, 78};

        // calling the setter function
        a1.setElement(Arr);

        // calling the display function
        a1.display();

        // new value is set at the 0th index
        Arr[0] = 2;
        System.out.println();

        // calling the display function one
more time
        a1.display();
    }
}
```



```
5 8 11 22 78
5 8 11 22 78
```