Packages

- Packages are containers for classes.
- They are used to keep the class name space compartmentalized. For example, a package allows the user to create a class named **List**, which can be stored inside a package without concern that it will collide with some other class named **List** stored elsewhere.

Defining a package

A package can be defined by including a **package** command as the first statement in a Java source file. The **package** statement defines a name space in which classes are stored. If the **package** statement is omitted, the class names are put into the default package.

This is the general form of the **package** statement:
package *pkg*;
Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage.**
package MyPackage;

A hierarchy of packages can be created. To do so each package name must be separated by a period. (**.**). For Example:
package java.awt.image;

Access Protection

The three access modifiers, private, public, and protected, provide ways to produce varied levels of access.

- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If one wants to allow an element to be seen outside the current package, but only to classes that subclass, then that element is declared protected.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

The above table provides an illustration as to how members with different access specifiers can be accessed.

Access Example

The sequence of codes below shows how the access is different when different modifiers are used.

Test_1 class contains 4 variables ‘a’, ‘b’, ‘c’, ‘d’ of private, public, default and public access respectively.

```
package package_programs_1;


public class Test_1 {

	private int a;
	public int b;
	int c;
	protected int d;

	void print()
	{
		System.out.print(a);
		System.out.print(b);
		System.out.print(c);
		System.out.print(d);
	}


}
```

Test_2 is another class within the same package. (package_programs_1). So the private variable “a” is not accessible.

```
package package_programs_1;

public class Test_2 {
	void print()
```

```
	{
		Test_1 t = new Test_1();
		System.out.print(t.a);// Not accessible
		System.out.print(t.b);
		System.out.print(t.c);
		System.out.print(t.d);
	}

}
```

Sample_1 is a class in different package. (package_programs_2). But this class extends Test_1. So the private variable “a” and default variable “c” are not accessible but the protected variable “d” and public variable “b” are accessible.

```
package package_programs_2;

public class Sample_1 extends package_programs_1.Test_1 {

	void print()
	{

		System.out.print(a); // Not accessible
		System.out.print(b);
		System.out.print(c); // Not accessible
		System.out.print(d);
	}



}
```

Sample_2 is a class in different package. (package_programs_2). Only the public variable “b” is accessible here below.

```
package package_programs_2;

import package_programs_1.Test_1;

public class Sample_2 {

	void print()
	{
		Test_1 t = new Test_1();
		System.out.print(t.a);// Not accessible
		System.out.print(t.b);
		System.out.print(t.c); // Not accessible
```

```
            System.out.print(t.d); // Not accessible
        }

}
```

Importing Packages

import statement is used to bring certain classes, or entire packages, into visibility. **import** statements occur immediately following the **package** statement and before any class definitions. The general form of import statement is:

import *pkg1* [.*pkg2*].(*classname* | *);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).Finally, anexplicit *classname* or a star (*) is specified. * indicates that the Java compiler should import the entire package. **java.lang** is a package that is implicitly imported by default by the compiler for all programs.

The **import** statement is optional. Any place if a user wants to use a class name, it can be done with its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

In this version, **Date** is fully-qualified.

Interfaces

Using the keyword **interface**, a class can be fully abstracted. Using **interface**, one can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

Defining an interface

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);

        type final-varname1 = value;
        type final-varname2 = value;
        .
        .
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

Example:

```
interface Test
{
        int speed = 40;
        void run();
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, the **implements** clause is included in a class definition, and then the methods defined by the interface are created. If a class implements more than one interface, the interfaces are separated with a comma. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

```
class Example implements Test{
        public void run()
        {
                System.out.println("Run fast");
        }
   }
```

The implementations can be accessed with interface references.

```
class Run
{
public static void main(String args[])
{
 Test t = new Example();
```

```
t.run();
}
}
```

Output:

Run fast

Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**. For example:

```
interface Test
{
        void play();
        void run();
}


abstract class Example implements Test{
        public void run()
        {
                System.out.println("Run fast");
        }
  }
```

The class Example above must be declared abstract as it is not implementing both the methods provided by the **Test** interface.

Practical example of an Interface

There are many ways to implement a stack. For example, a stack can be of a fixed size or it can be "growable." No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack, independent of the details of the implementation.

In the example below two stack classes, FixedStack and DynamicStack are created have a common Stack interface.

```
interface IntStack {
        void push(int item); // store an item
        int pop(); // retrieve an item
}


class FixedStack implements IntStack {
                int stck[];
                int tos;

                FixedStack(int size) {
                        stck = new int[size];
                        tos = -1;
                }

                public void push(int item) {
                        if(tos==stck.length-1)
                        System.out.println("Stack is full.");
                        else
                        stck[++tos] = item;
                }

                public int pop() {
```

```
                        if(tos < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                        }

                        else
                        return stck[tos--];
                        }
                }
 class DynStack implements IntStack {
                int stck[];
                int tos;

                DynStack(int size) {
                        stck = new int[size];
                        tos = -1;
                }
public void push(int item) {

                        if(tos==stck.length-1) {
                        int temp[] = new int[stck.length * 2]; // double size
                        for(int i=0; i<stck.length; i++) temp[i] = stck[i];
                        stck = temp;
                        stck[++tos] = item;
                        }

                        else
                        stck[++tos] = item;
                }

                public int pop() {
                        if(tos < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                        }
                        else
                        return stck[tos--];
                }
 }


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

        FixedStack fs = new FixedStack(5)
        DynStack ds = new DynStack(5);

        for(int i=0; i<12; i++)
        fs.push(i);

        System.out.println("Fixed Stack elements:");
        for(int i=0; i<12; i++)
        System.out.println(fs.pop());

        for(int i=0; i<12; i++)
        ds.push(i);

        System.out.println("Dynamic Stack elements:");
        for(int i=0; i<12; i++)
        System.out.println(ds.pop());



        }
 }
```

The size of the fixed stack is 5 and hence there will be “stack is full” message once the stack is full but in the dynamic stack the size doubles each time the stack is full and hence no such messages pop up.

Default methods in an interface:

Like regular interface methods, default methods are implicitly public; there's no need to specify the *public* modifier. Unlike regular interface methods, they are declared with the *default* keyword at the beginning of the method signature, and they provide an implementation.

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down. Default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Therefore, there will be no need to modify the already implemented classes.

Eg:

/*interface*/

```
public interface Vehicle {

    String getBrand();

    String speedUp();

    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }

}
```

/*Class implementing the interface*/

```
class Car implements Vehicle {

    private String brand;

    Car(String brand){
        this.brand=brand;
    }

    @Override
    public String getBrand() {
        return brand;
    }

    @Override
    public String speedUp() {
        return "The car is speeding up.";
    }

    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

/*Main Class*/

```
class Main_Run{
    public static void main(String[] args) {
        Vehicle car = new Car("BMW");
        System.out.println(car.getBrand());
        System.out.println(car.speedUp());
        System.out.println(car.slowDown());
        System.out.println(car.turnAlarmOn());
        System.out.println(car.turnAlarmOff());
    }
}
```

In the above example, the default methods, turnAlarmOn() and turnAlarmOff(), from Vehicle interface are automatically available in the Car class. Furthermore, if at some point we decide to add more default methods to the Vehicle interface, the application will still continue working, and we won't have to force the class to provide implementations for the new methods. The most common use of interface default methods is to incrementally provide additional functionality to a given type without breaking down the implementing classes.

Static Interface Methods

In addition to declaring default methods in interfaces, Java 8 also allows us to define and implement static methods in interfaces. Since static methods don't belong to a particular object, they're not part of the API of the classes implementing the interface; therefore, they have to be called by using the interface name preceding the method name.

Ex:

```
public interface Vehicle {
   // regular / default interface methods
   static int getSq(int s) {
      return (s*s);
   }
}
```

Defining a static method within an interface is identical to defining one in a class. Moreover, a static method can be invoked within other static and default methods.