Object Oriented Programming with Java

Zheng Chen

Package, access control and interface

1. Package and import

2. Access control

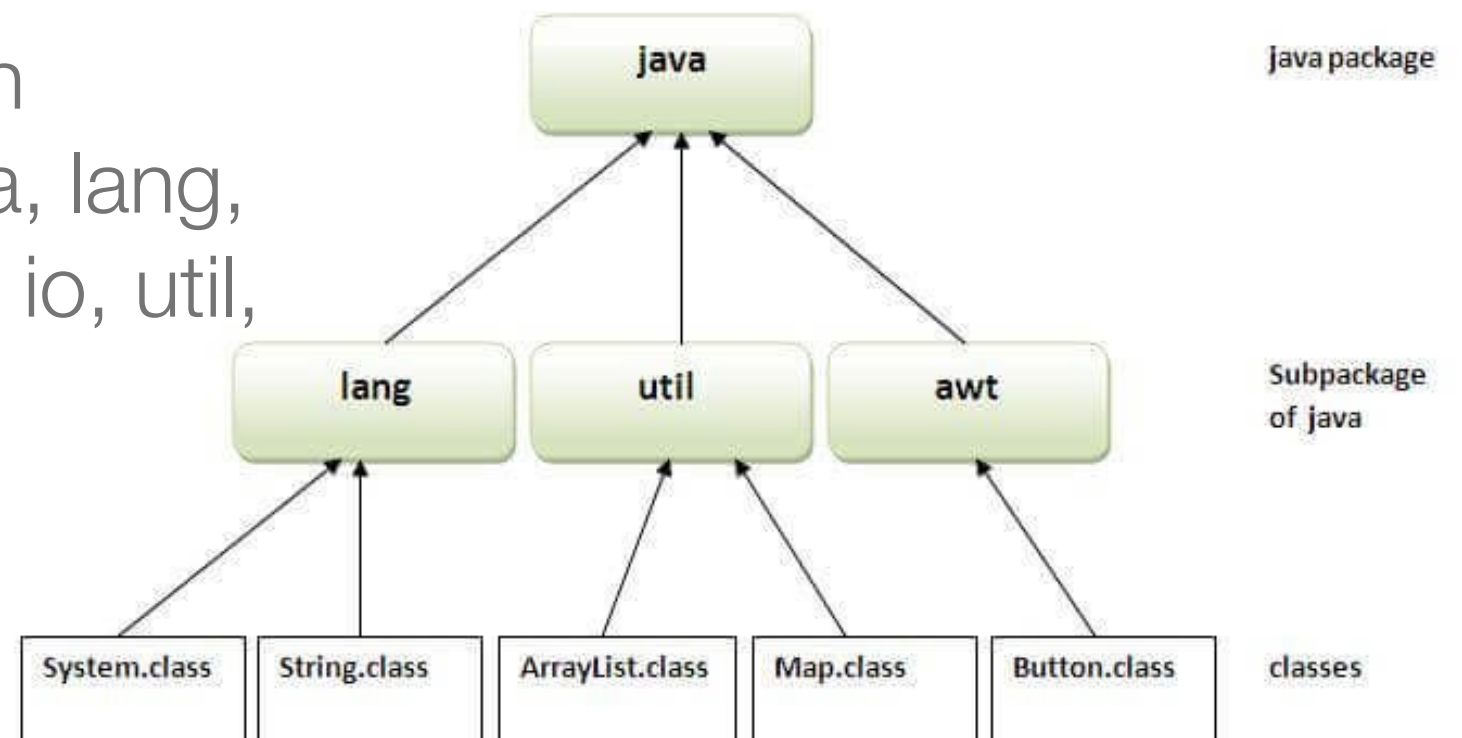3. Interface

Package

- A java package is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.

  -- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Why use package

- Preventing naming conflicts.

- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier

- Providing controlled access: protected and default have package level access control.

- Packages can be considered as data encapsulation (or data-hiding).

Package statement

- The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

<div align="center">

package packagename;

</div>

- If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

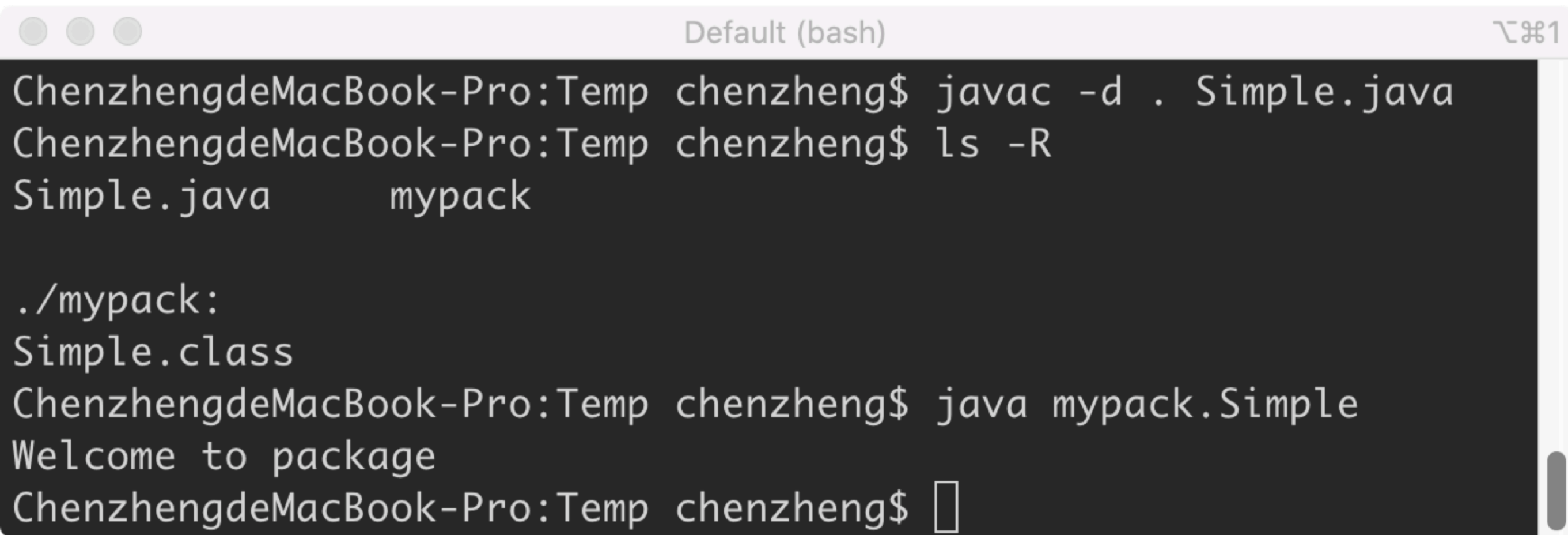- To compile the Java programs with package statements, you have to use -d option.

<div align="center">

javac -d Destination_folder file_name.java

</div>

How packages work?

- Package names and directory structure are closely related. If a package name is edu.uestc.ss, then there are three directories, edu, uestc and ss such that ss is present in uestc and uestc is present edu.

- The directory college is accessible through **CLASSPATH** variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

Simple example of java package

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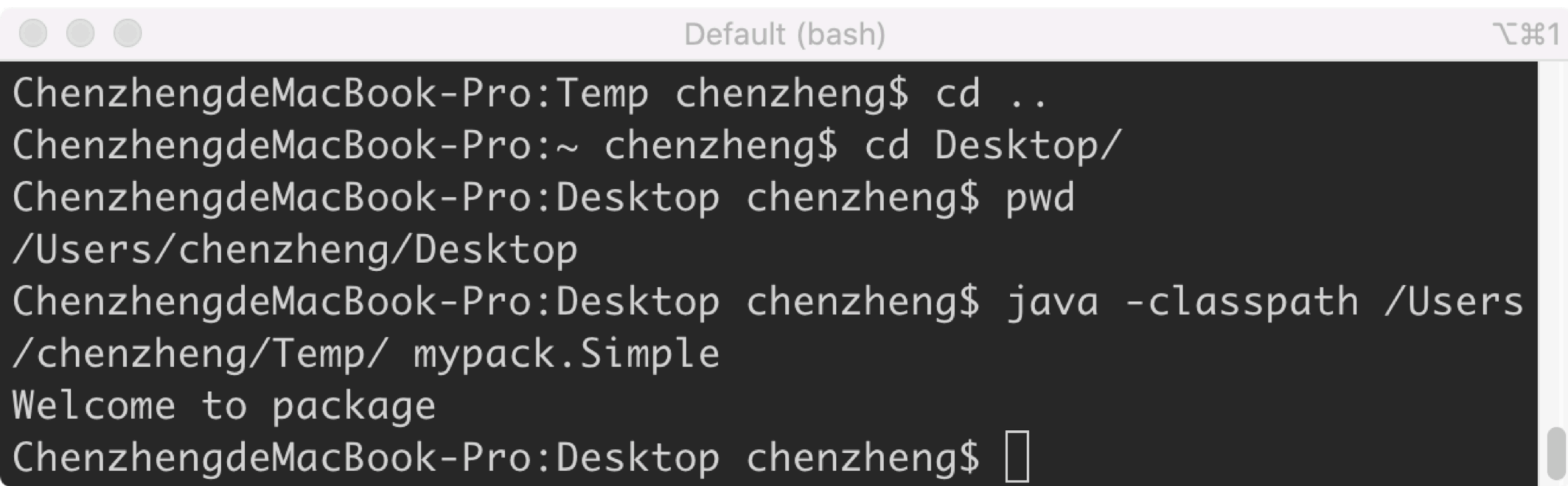


```
ChenzhengdeMacBook-Pro:Temp chenzheng$ javac -d . Simple.java
ChenzhengdeMacBook-Pro:Temp chenzheng$ ls -R
Simple.java      mypack

./mypack:
Simple.class
ChenzhengdeMacBook-Pro:Temp chenzheng$ java mypack.Simple
Welcome to package
ChenzhengdeMacBook-Pro:Temp chenzheng$ ▯
```

run program by -classpath

- The -classpath switch can be used with javac. You can use -classpath switch of java that tells where to look for class file.

```
Default (bash)                                                          ⌥⌘1
ChenzhengdeMacBook-Pro:Temp chenzheng$ cd ..
ChenzhengdeMacBook-Pro:~ chenzheng$ cd Desktop/
ChenzhengdeMacBook-Pro:Desktop chenzheng$ pwd
/Users/chenzheng/Desktop
ChenzhengdeMacBook-Pro:Desktop chenzheng$ java -classpath /Users
/chenzheng/Temp/ mypack.Simple
Welcome to package
ChenzhengdeMacBook-Pro:Desktop chenzheng$ ▯
```

Access package

- There are three ways to access the package from outside the package.

  - import package.*;

  - import package.classname;

  - fully qualified name.

Using fully qualified name

- If you use fully qualified name then there is no need to import.

- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

```
package anotherpack;
public class AnotherSimple{
 public static void main(String args[]){
   mypack.Simple s=new mypack.Simple();
   System.out.println("I can access " + s);
  }
}
```

import packagename.*

- The **import** keyword is used to make the classes and interface of another package accessible to the current package.

- If you import package.* then all the classes and interfaces of this package will be accessible **but not subpackages**.

```
package anotherpack;
import mypack.*;
public class AnotherSimple{
 public static void main(String args[]){
   Simple s=new Simple();
   System.out.println("I can access " + s);
  }
}
```

import packagename.classname

- If you import package.classname then only declared class of this package will be accessible.

```
package anotherpack;
import mypack.Simple
public class AnotherSimple{
 public static void main(String args[]){
    Simple s=new Simple();
    System.out.println("I can access " + s);
 }
}
```

Static Import

- A static import declaration imports static members (variables/methods) of a type into a compilation unit.

```java
import static  java.lang.System.out;
public class  StaticImportTest  {
    public static  void  main(String[]  args)  {
        out.println("Hello static  import!");
    }
}
```
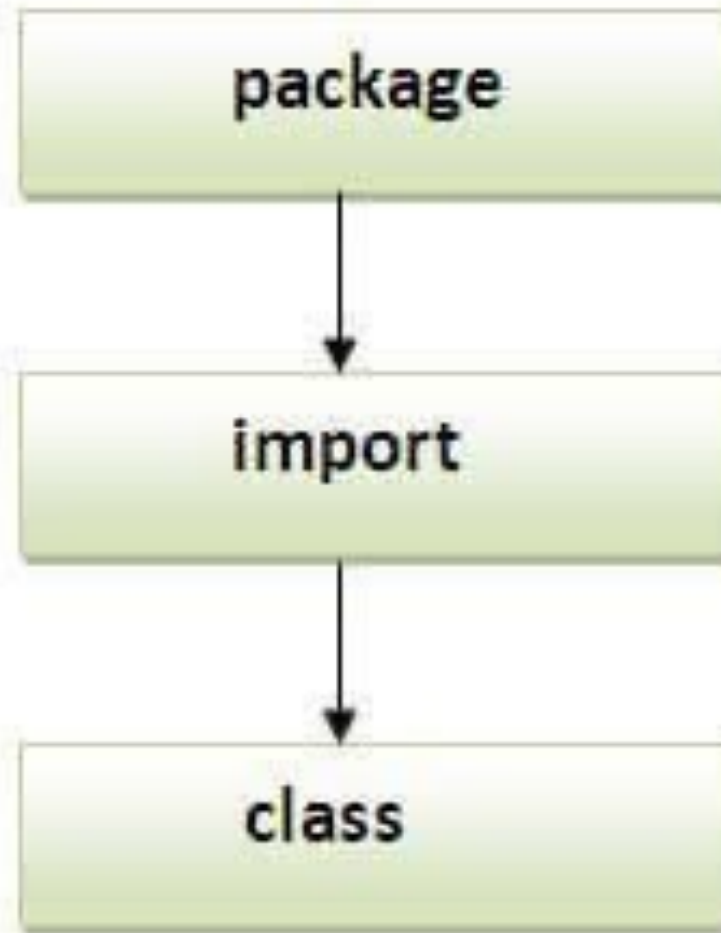
Sequence of the program must be package then import then class.

Package class

- The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle() etc.

Package example

```
class PackageInfo{

    public static void main(String args[]){

        Package p=Package.getPackage("java.lang");

        System.out.println("package name: "+p.getName());

        System.out.println("Specification Title: "+p.getSpecificationTitle());
        System.out.println("Specification Vendor: "+p.getSpecificationVendor());
        System.out.println("Specification Version: "+p.getSpecificationVersion());

        System.out.println("Implementaion Title: "+p.getImplementationTitle());
        System.out.println("Implementation Vendor: "+p.getImplementationVendor());
        System.out.println("Implementation Version: "+p.getImplementationVersion());
        System.out.println("Is sealed: "+p.isSealed());
    }
}
```

Package, access control and interface

KEEP
CALM
AND
CODE
JAVA

Access Modifiers

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.

  - **private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

  - **default**: The access level of a default modifier 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.

  - **protected**: The access level of a protected modifier 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.

  - **public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

The **private** is accessible only within the class

```java
class A {
    private A() {
    }//private constructor
    void msg() {
        System.out.println("Hello java");
    }
}

public class AnotherClass {
    public static void main(String args[]) {
        A obj = new A();//Compile Time Error
    }
}
```

The **default** is accessible only within package

```
package pack;

class A {
    void msg() {
        System.out
            .println("Hello");
    }
}
```

```
package anotherpack;
import pack.*;

class B {
    public static void main(String args[]) {
        A obj = new A();
        //Compile Time Error
        obj.msg();
        //Compile Time Error
    }
}
```

The **protected** is accessible within package and outside the package but through inheritance only

```
package pack;

public class A {
    protected void msg() {
        System.out
            .println("Hello");
    }
}
```

```
package anotherpack;
import pack.*;

class B extends A {
    public static void main(String args[]) {
        B obj = new B();
        obj.msg(); //legal
    }
}
```

The **public** is accessible everywhere

```java
package pack;

public class A {
    public void msg() {
        System.out
            .println("Hello");
    }
}
```

```java
package anotherpack;
import pack.*;

class B {
    public static void main(String args[]) {
        B obj = new B(); //legal
        obj.msg(); //legal
    }
}
```

Access Modifiers with Method Overriding

- If you are overriding any method, overridden method (i.e. declared in subclass) **must not** be more restrictive.

```
class A {
    protected void msg() {
        System.out.println("Hello java");
    }
}

public class Simple extends A {
    void msg() {
        System.out.println("Hello java");
    }//C.T.Error
}
```

Package, access control and interface

Interface

- An interface in java is a **blueprint** of a class. It has static constants and abstract methods.

- The interface in Java is a mechanism to achieve **abstraction**. There can be only abstract methods in the Java interface.

- It is used to achieve **multiple inheritance** in Java.

- Java Interface also represents the **IS-A** relationship.

Interface Syntax

```
interface InterfaceName{
    // declare constant fields
    // declare methods that abstract
    // by default.
}

class ClassName implements InterfaceName{
    //implement methods
}
```

- The Java compiler adds **public** and **abstract** keywords before the interface method. Moreover, it adds **public**, **static** and **final** keywords before data members.

Interface

- Interface only has static constants and abstract methods.

- It cannot be instantiated just like the abstract class.

- Since Java 8, we can have default and static methods in an interface.

- Since Java 9, we can have private methods in an interface.

The relationship between classes and interfaces
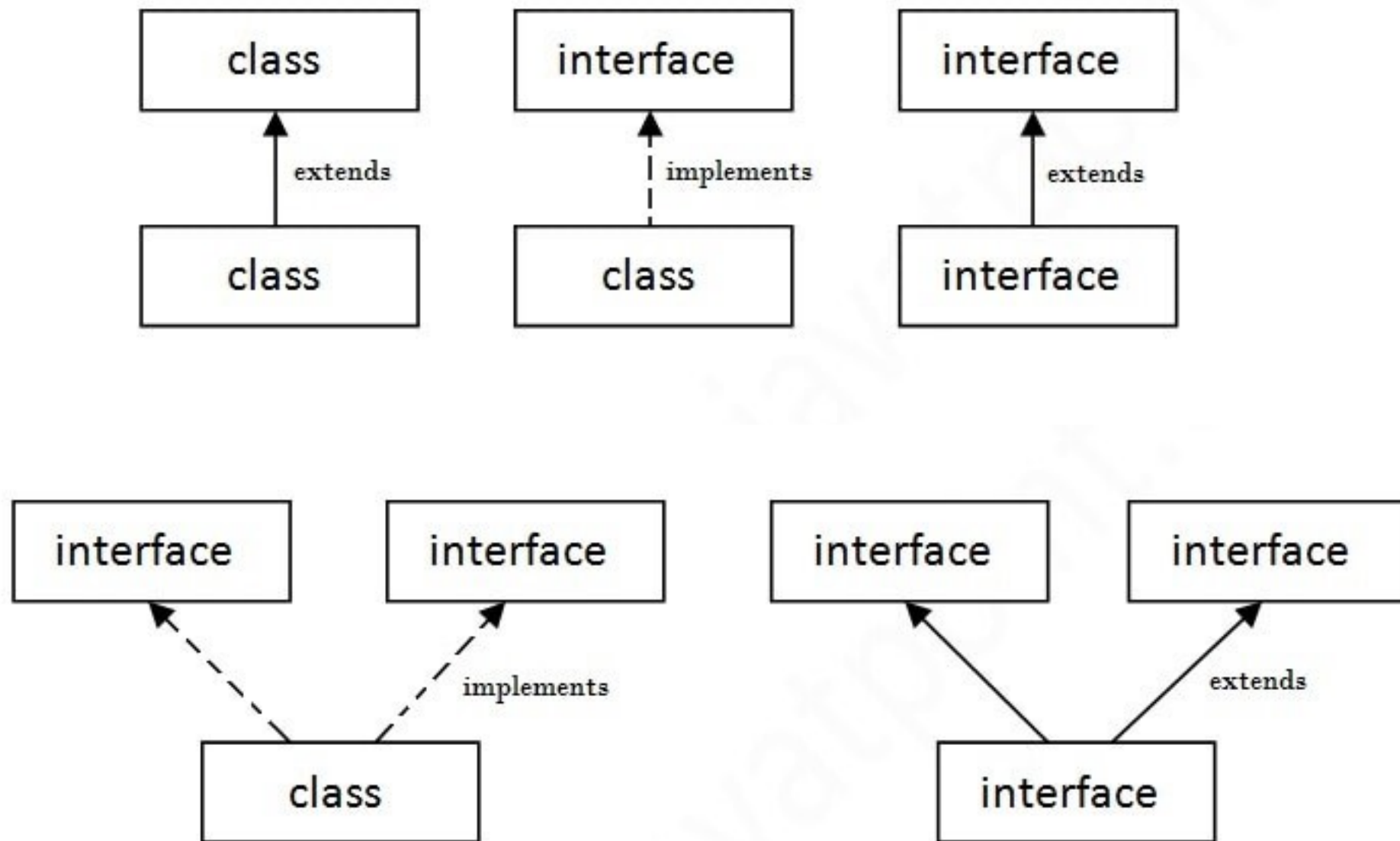
Interface Example

```java
interface printable {
    void print();
}

class Hello implements printable {
    public void print() {
        System.out.println("Hello");
    }
    public static void main(String args[]) {
        Hello obj = new Hello();
        obj.print();
    }
}
```

Interface Example 2

```
interface Drawable {
    void draw();
}

class Rectangle implements Drawable
{
    public void draw() {
        System.out.println
            ("drawing rectangle");
    }
}

class Circle implements Drawable {
    public void draw() {
        System.out.println
            ("drawing circle");
    }
}
```

```
class TestInterface {

    public static void main(String args[]) {
        Drawable d = new Circle();
        d.draw();

        d = new Rectangle();
        d.draw();

        ……
    }
}
```

Interface and abstract

```java
interface RateAble {
    float rateOfInterest();
}

abstract class Bank
        implements RateAble {
    abstract String getName();
}

class SBI extends Bank {
    String getName() {
        return "SBI";
    }
    public float rateOfInterest() {
        return 9.15f;
    }
}
```

```java
class PNB extends Bank {
    String getName() {
        return "PBI";
    }
    public float rateOfInterest() {
        return 9.7f;
    }
}

class TestInterface {
    public static void main(String[] args) {
        Bank b = new SBI();
        System.out.println(b.getName()
                + "'s ROI: " + b.rateOfInterest());
        b = new PNB();
        System.out.println(b.getName()
                + "'s ROI: " + b.rateOfInterest());
    }
}
```

Interface multi-implementation

```
interface RateAble {
    float rateOfInterest();
}

interface HasName {
    String getName();
}

abstract class Bank implements RateAble, HasName {

}
```

Interface multi-inheritance

```
interface RateAble {
    float rateOfInterest();
}

interface HasName {
  String getName();
}

interface AllInOne extends RateAble, HasName {

}

abstract class Bank implements AllInOne {

}
```

Mixture of implementation and inheritance

```java
interface RateAble {
    float rateOfInterest();
}

interface HasName {
    String getName();
}

class Bank {
    //...
}
```

```java
class SBI
    extends Bank
    implements HasName, RateAble {
    public String getName() {
        return "SBI";
    }
    public float rateOfInterest() {
        return 9.15f;
    }
}
```

marker interface

- An interface which has no member is known as a marker interface. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//Serializable is a marker interface
public interface Serializable{
    //no member…
}
```

Java 8 Default Method in Interface

```
interface Drawable {
    void draw();
    default void msg() {
        System.out.println("default method");
    }
}


class Rectangle implements Drawable {
    public void draw() {
        System.out.println("drawing rectangle");
    }
}


class TestInterfaceDefault {
    public static void main(String args[]) {
        Drawable d = new Rectangle();
        d.draw();
        d.msg();
    }
}
```

- Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

Java 8 Static Method in Interface

- The interfaces can have static methods as well which is similar to static method of classes.

```java
interface Drawable {
    void draw();
    static void showName() {
        System.out.println("Static Method in Drawable");
    }
}

class TestInterfaceStatic {
    public static void main(String args[]) {
        Drawable.showName();
    }
}
```

Java 9 Private methods in Interfaces

- java 9 private interface methods are not inherited by sub-interfaces or implementations.

- If two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it's implementing classes.

```java
public interface DBLogging {
    default void logInfo(String message) {
        log(message, "INFO");
    }
    default void logWarn(String message) {
        log(message, "WARN");
    }
    default void logError(String message) {
        log(message, "ERROR");
    }
    default void logFatal(String message) {
        log(message, "FATAL");
    }

    private void log(String message, String msgPrefix) {
        // Step 1: Connect to DataStore
        // Step 2: Log Message with Prefix and styles etc.
        // Step 3: Close the DataStore connection
    }
}
```

Why use Java interface?

- It is used to achieve abstraction.

- By interface, we can support the functionality of multiple inheritance.

- It can be used to achieve loose coupling.

Abstract class	Interface
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

GeometricObject
<<interface>>

+getPerimeter():double
+getArea():double

Circle

#radius:double = 1.0

+Circle(radius:double)
+getPerimeter():double
+getArea():double

Resizable
<<interface>>

+resize(percent:int)

ResizableCircle

+ResizableCircle(radius:double)
+resize(percent:int)