Java

Package & Interface

Package

Package

- Java package provides a mechanism for partitioning the class name space into more manageable chunks
  - Both **naming** and **visibility** control mechanism
- ★ Define classes inside a package that are not accessible by code outside that package.
- Define class members that are exposed only to other members of the same package
- This allows classes to have intimate knowledge of each other
  - Not expose that knowledge to the rest of the world

Declaring Package

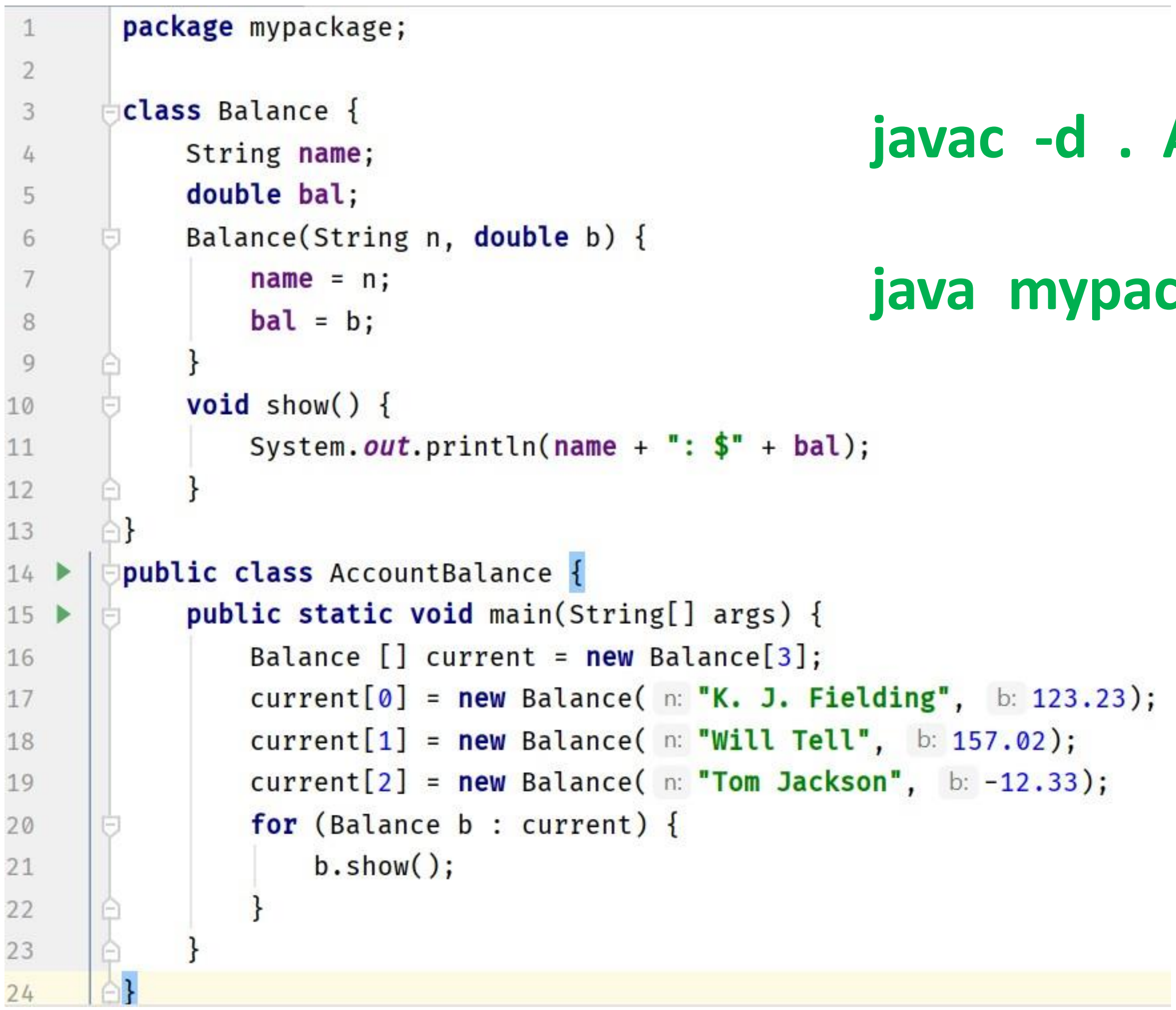
- ***package pkg;***
  - Here, pkg is the name of the package
- ***package mypackage;***
  - creates a package called mypackage
- The package statement defines a name space in which classes are stored
- If you omit the package statement, the class names are put into the **default package**, which has no name

Declaring Package

- Java uses file system directories to store packages
  - the .class files for any classes that are part of mypackage must be stored in a directory called mypackage
- *More than one file can include the same package statement*
- The package statement simply specifies to which package the classes defined in a file belong
- To create hierarchy of packages, separate each package name from the one above it by use of a (.)

package pkg14[.pkg2[.pkg3]];

Package Example



```
package mypackage;

class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        System.out.println(name + ": $" + bal);
    }
}
public class AccountBalance {
    public static void main(String[] args) {
        Balance [] current = new Balance[3];
        current[0] = new Balance( n: "K. J. Fielding",  b: 123.23);
        current[1] = new Balance( n: "Will Tell",  b: 157.02);
        current[2] = new Balance( n: "Tom Jackson",  b: -12.33);
        for (Balance b : current) {
            b.show();
        }
    }
}
```

```
javac  -d  .  AccountBalance.java
```

```
java  mypackage.AccountBalance
```

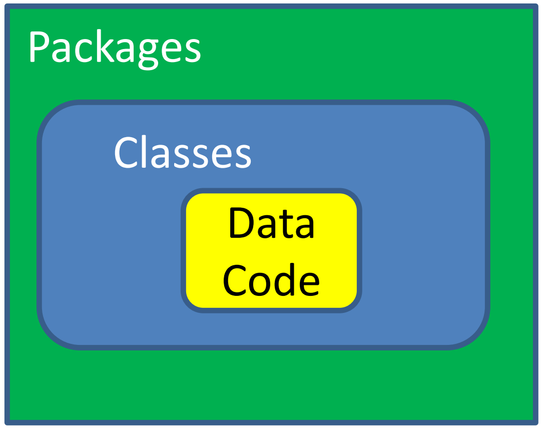
Package Syntax

- The general form of a multilevel package statement
  - ***package pkg1[.pkg2[.pkg3]]***
  - ***package java.util.concurrent***
- Import statements occur immediately following the package statement and before any class definitions
- The general form of the import statement
  - ***import pkg1 [.pkg2].(classname | *)***
  - ***import java.util.Scanner***
  - import statement is optional, class can be used with name that includes full package hierarchy

At times than importing a whole package we can just import class.

Access Protection

- Packages act as containers for classes and other subordinate packages
- Classes act as containers for data and code
- The class is Java's smallest unit of abstraction
- Four categories of visibility for class members
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different package
  - Classes that are neither in the same package nor subclasses



Access Protection

- The three access modifiers provide a variety of ways to produce the many levels of access required
  - private, public, and protected
- The following applies only to members of classes

	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

For detail example, please refer to codes in package **p1** and **p2**

Access Protection

- 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 (***default access***)
- If you want to allow an element to be seen outside your current package, but only to classes that subclass the class directly, declare that ***protected***

Access Protection

- A non-nested class has only two possible access levels
  - **default** and **public** (others are **abstract** and **final**)
- When a class is declared as public, it is accessible by any other code
- If a class has default access, then it can only be accessed by other code within its same package
- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class

Interface

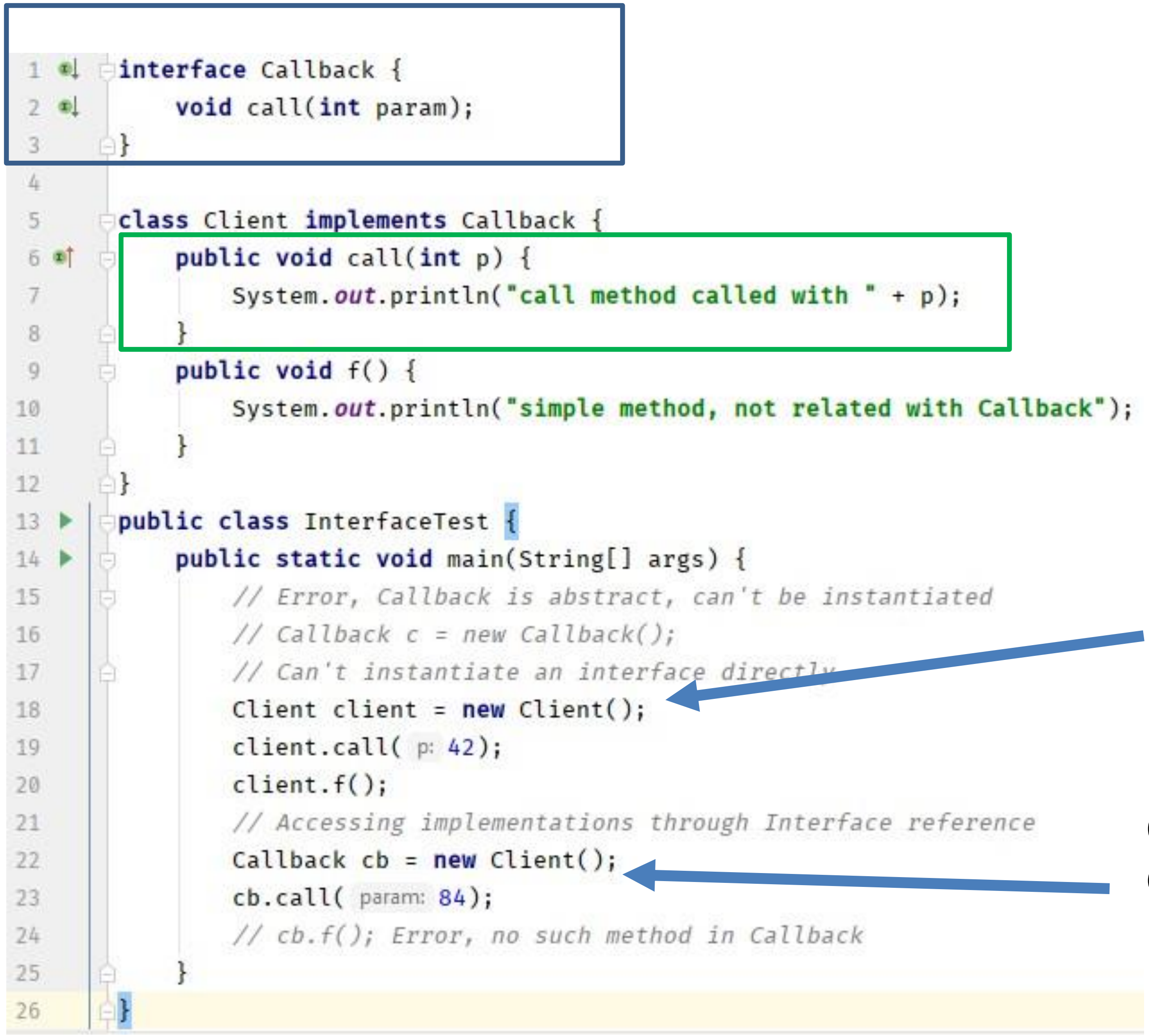
Interface

- We can call it a pure abstract class having no concrete methods.
- You can tell what a class must do, not how it does it.
  - All methods declared in an interface are implicitly **public** and **abstract**
  - All variables declared in an interface are implicitly **public, static** and **final**
- An interface can't have instance variables, so can't maintain state information unlike class
- A class can only extend from a **single class**, but a class can implement **multiple interfaces**

Implementing Interface

- When you implement an interface method, it must be declared as public
- By implementing an interface, a class signs a contract with the compiler that it will definitely provide implementation of all the methods
  - If it fails to do so, the class will be considered as abstract
  - Then it must be declared as abstract and no object of that class can be created
- An abstract class specifies **what an object is** and an interface specifies **what the object can do**

Simple Interface



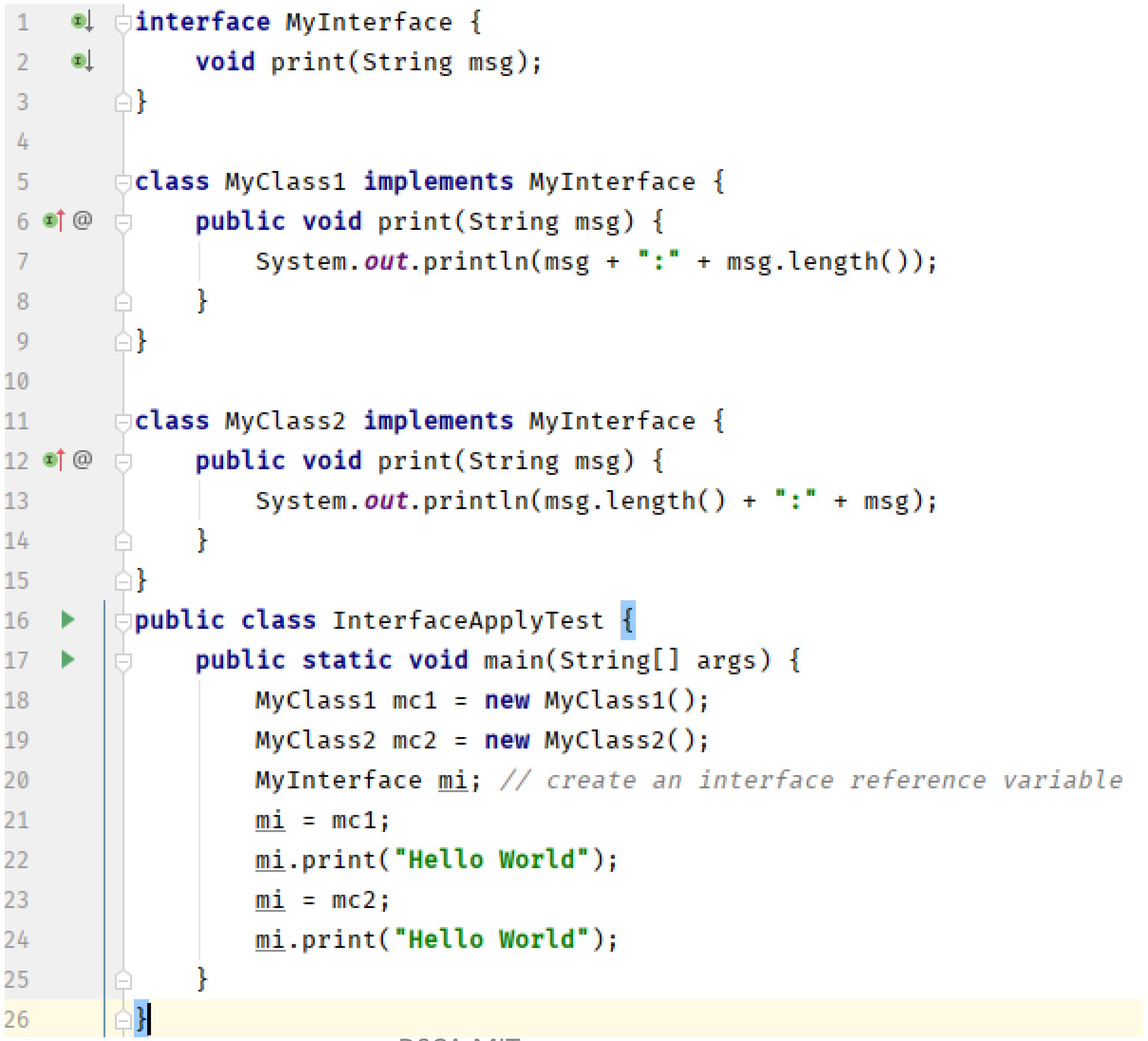
```
interface Callback {
    void call(int param);
}

class Client implements Callback {
    public void call(int p) {
        System.out.println("call method called with " + p);
    }
    public void f() {
        System.out.println("simple method, not related with Callback");
    }
}
public class InterfaceTest {
    public static void main(String[] args) {
        // Error, Callback is abstract, can't be instantiated
        // Callback c = new Callback();
        // Can't instantiate an interface directly
        Client client = new Client();
        client.call( p: 42);
        client.f();
        // Accessing implementations through Interface reference
        Callback cb = new Client();
        cb.call( param: 84);
        // cb.f(); Error, no such method in Callback
    }
}
```

Object to class Client is possible: Pass

Object to interface Callback is not possible: Error

InterfaceTest.java

Applying Interfaces



```java
interface MyInterface {
    void print(String msg);
}

class MyClass1 implements MyInterface {
    public void print(String msg) {
        System.out.println(msg + ":" + msg.length());
    }
}

class MyClass2 implements MyInterface {
    public void print(String msg) {
        System.out.println(msg.length() + ":" + msg);
    }
}
public class InterfaceApplyTest {
    public static void main(String[] args) {
        MyClass1 mc1 = new MyClass1();
        MyClass2 mc2 = new MyClass2();
        MyInterface mi; // create an interface reference variable
        mi = mc1;
        mi.print("Hello World");
        mi = mc2;
        mi.print("Hello World");
    }
}
```

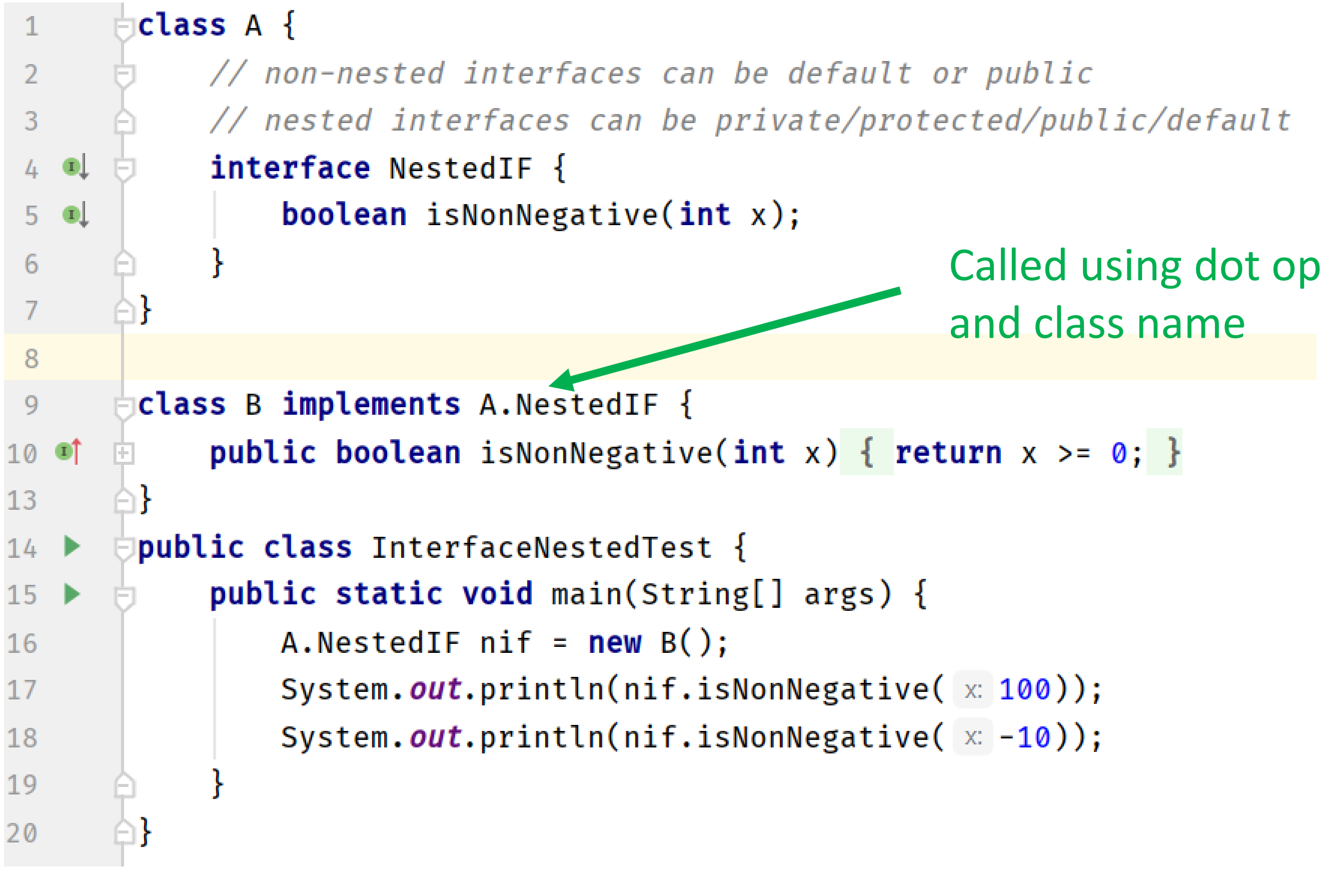
InterfaceApplyTest.java

Nested or Member Interfaces

- An interface can be declared as a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Called using dot operator and class name

Nested or Member Interfaces



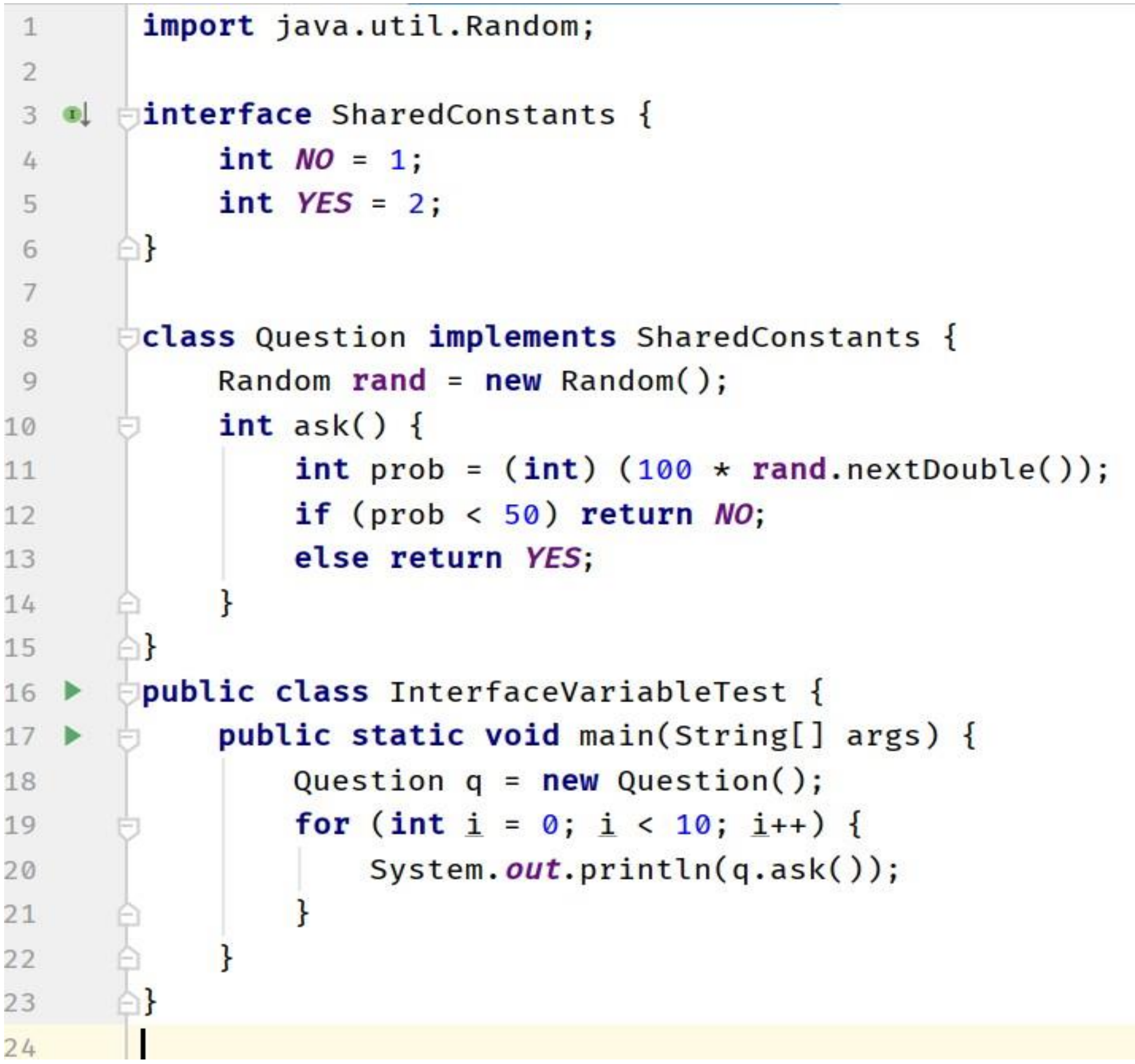
```java
class A {
    // non-nested interfaces can be default or public
    // nested interfaces can be private/protected/public/default
    interface NestedIF {
        boolean isNonNegative(int x);
    }
}

class B implements A.NestedIF {
    public boolean isNonNegative(int x) { return x >= 0; }
}
public class InterfaceNestedTest {
    public static void main(String[] args) {
        A.NestedIF nif = new B();
        System.out.println(nif.isNonNegative( x: 100));
        System.out.println(nif.isNonNegative( x: -10));
    }
}
```

Called using dot operator and class name

InterfaceNestedTest.java

Variables in Interfaces

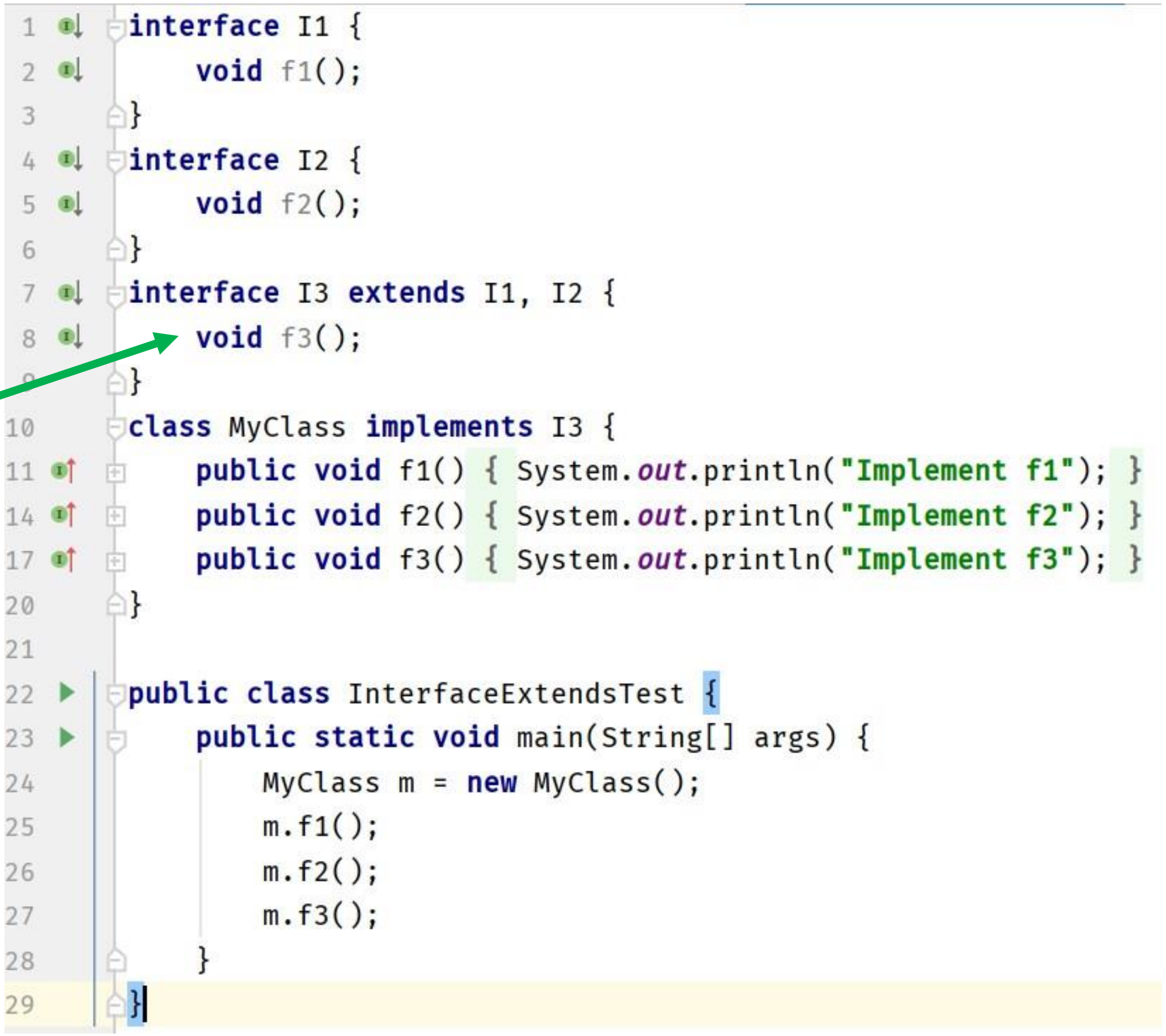
You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.



```java
import java.util.Random;

interface SharedConstants {
    int NO = 1;
    int YES = 2;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 50) return NO;
        else return YES;
    }
}
public class InterfaceVariableTest {
    public static void main(String[] args) {
        Question q = new Question();
        for (int i = 0; i < 10; i++) {
            System.out.println(q.ask());
        }
    }
}
```

InterfaceVariableTest.java

Extending Interfaces



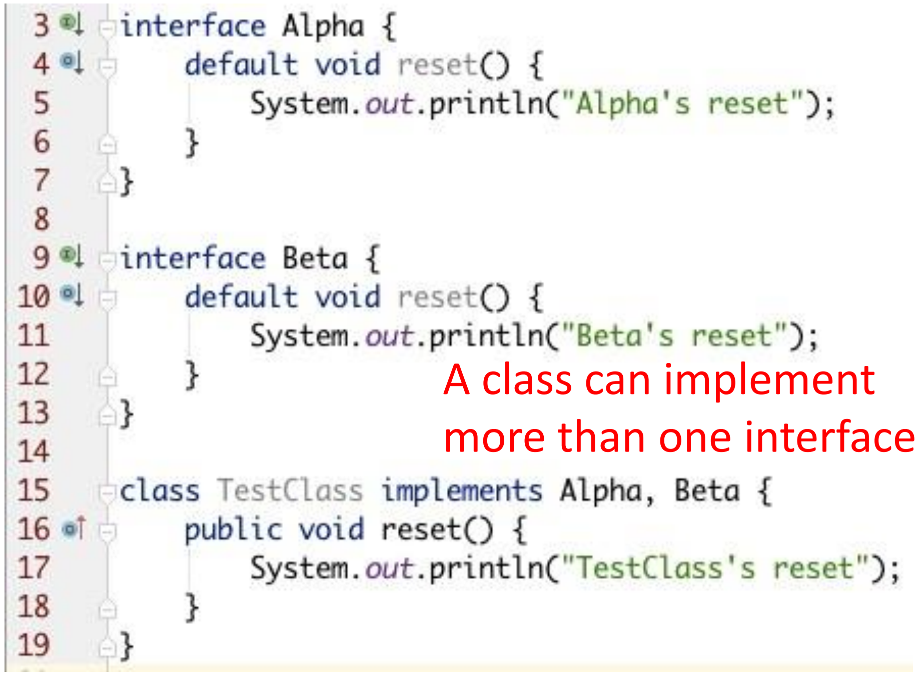
```
interface I1 {
    void f1();
}
interface I2 {
    void f2();
}
interface I3 extends I1, I2 {
    void f3();
}
class MyClass implements I3 {
    public void f1() { System.out.println("Implement f1"); }
    public void f2() { System.out.println("Implement f2"); }
    public void f3() { System.out.println("Implement f3"); }
}

public class InterfaceExtendsTest {
    public static void main(String[] args) {
        MyClass m = new MyClass();
        m.f1();
        m.f2();
        m.f3();
    }
}
```

Interfaces can inherit other interfaces.

InterfaceExtendsTest.java

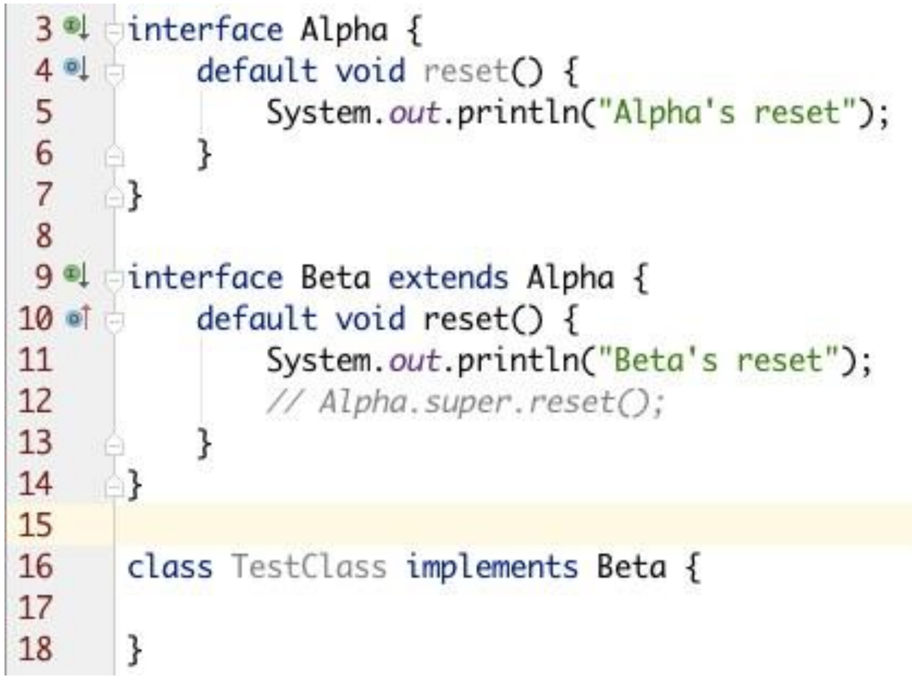
Multiple Inheritance Issues

Multiple Vs Multi Level Interface



```java
interface Alpha {
    default void reset() {
        System.out.println("Alpha's reset");
    }
}

interface Beta {
    default void reset() {
        System.out.println("Beta's reset");
    }
}

class TestClass implements Alpha, Beta {
    public void reset() {
        System.out.println("TestClass's reset");
    }
}
```

A class can implement more than one interface



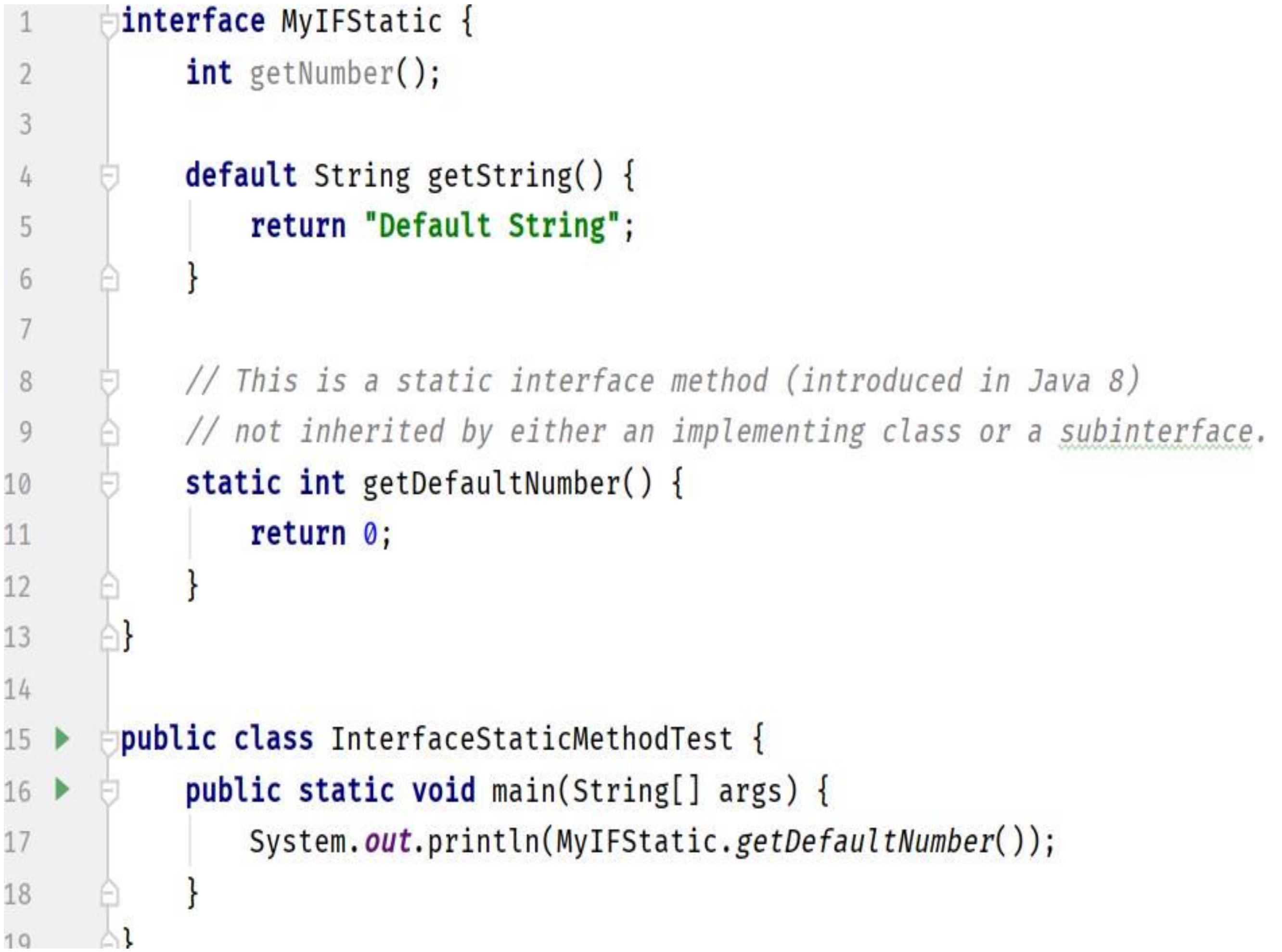
```java
interface Alpha {
    default void reset() {
        System.out.println("Alpha's reset");
    }
}

interface Beta extends Alpha {
    default void reset() {
        System.out.println("Beta's reset");
        // Alpha.super.reset();
    }
}

class TestClass implements Beta {

}
```

Aplha →Beta → TestClass

Static Methods in Interface

- Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object.

- **static** method is called by specifying the interface name, followed by a period, followed by the method name.
- Syntax:



```java
interface MyIFStatic {
    int getNumber();

    default String getString() {
        return "Default String";
    }

    // This is a static interface method (introduced in Java 8)
    // not inherited by either an implementing class or a subinterface.
    static int getDefaultNumber() {
        return 0;
    }
}

public class InterfaceStaticMethodTest {
    public static void main(String[] args) {
        System.out.println(MyIFStatic.getDefaultNumber());
    }
}
```

InterfaceName.staticMethodName

InterfaceStaticMethodTest.java