Interface

Creating and Using Interfaces

- Interfaces
  - It is a 100% abstraction
  - Contains only constants and `abstract` methods
    - All fields are implicitly `public`, `static` and `final`
    - All methods are implicitly `public abstract` methods
  - Classes can `implement` interfaces
    - The class must declare each method in the interface using the same signature or the class must be declared `abstract`
  - Typically used when disparate classes need to share common methods and constants
  - Normally declared in their own files with the same names as the interfaces and with the `.java` file-name extension

Good Programming Practice

- It is proper style to declare an interface’s methods without keywords `public` and `abstract` because they are redundant in interface method declarations. Similarly, constants should be declared without keywords `public`, `static` and `final` because they, too, are redundant.

- When declaring a method in an interface, choose a method name that describes the method’s purpose in a general manner, because the method may be implemented by a broad range of unrelated classes.

Common Programming Error

- Failing to implement any method of an interface in a concrete class that `implements` the interface results in a syntax error indicating that the class must be declared `abstract`.

What is an Interface?

- An interface is just like Java Class, but it only has static constants and abstract method. Java uses Interface to implement multiple inheritance. A Java class can implement **multiple Java Interfaces**. All methods in an interface are implicitly public and abstract.
- **Syntax for Declaring Interface**

```
interface {
//methods
}
```

- To use an interface in your class, append the keyword "**implements**" after your class name followed by the interface name.

  Example for Implementing Interface

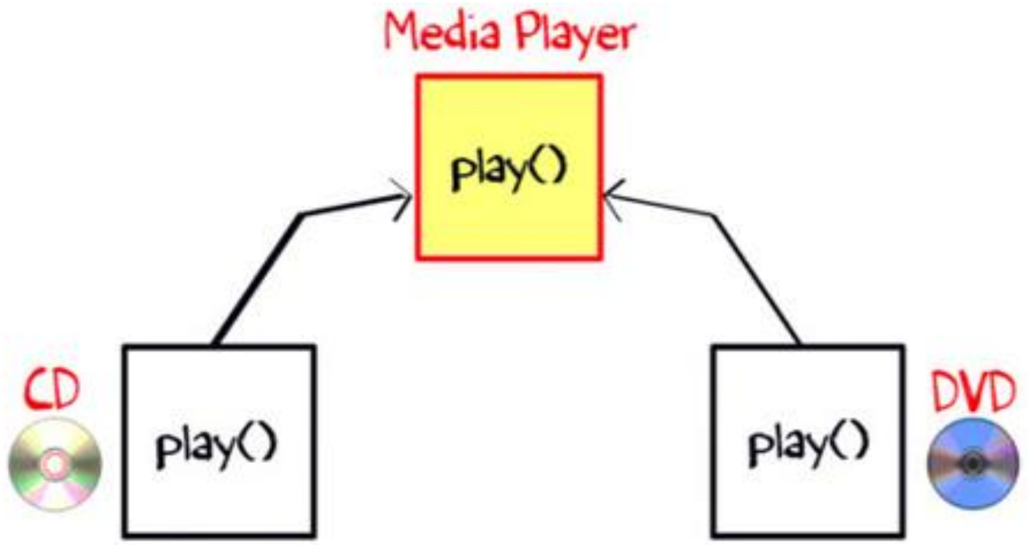
```
class Dog implements Pet
```

- As we've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

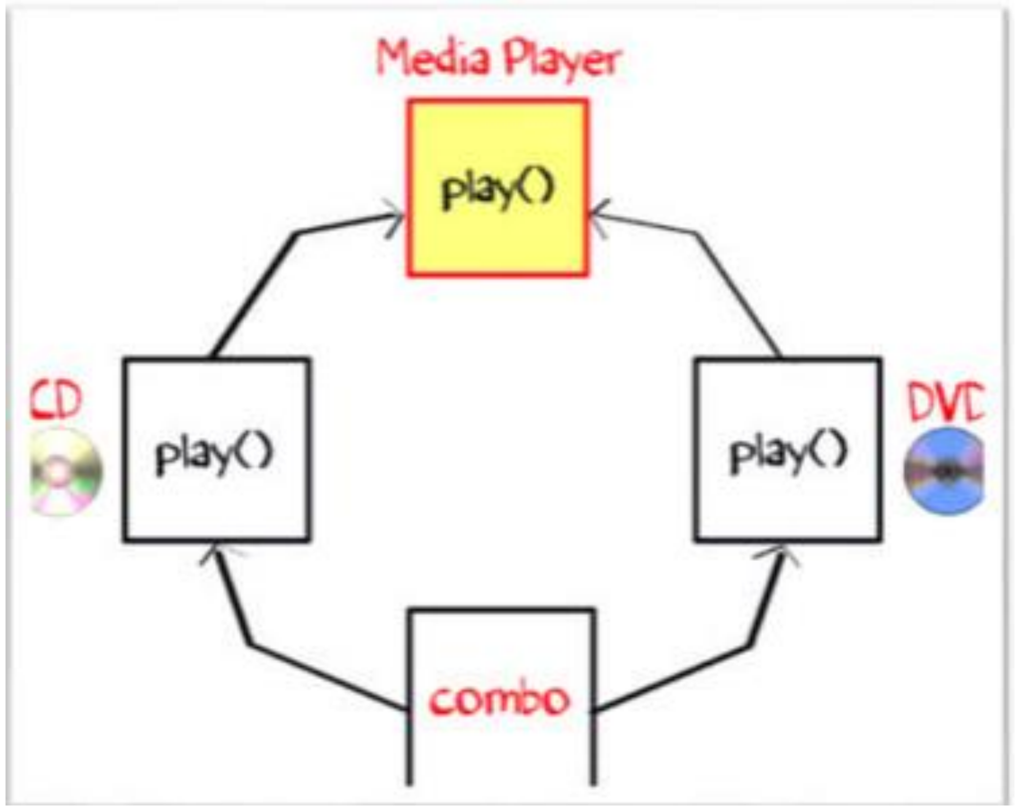
- In its most common form, an interface is a group of related methods with empty bodies

Why is an Interface required?

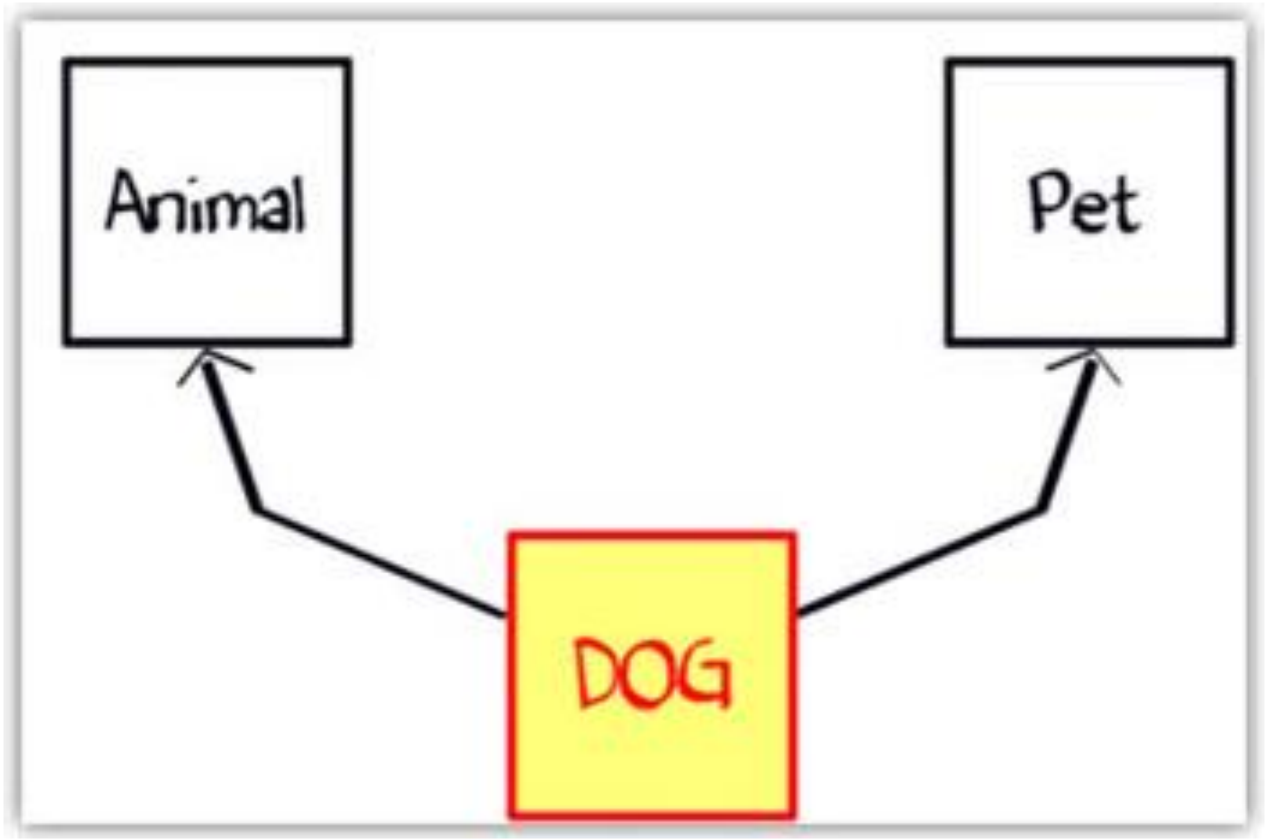
- To understand the concept of Java Interface better, let see an example. The class "Media Player" has two subclasses: CD player and DVD player. Each having its unique implementation method to play music.



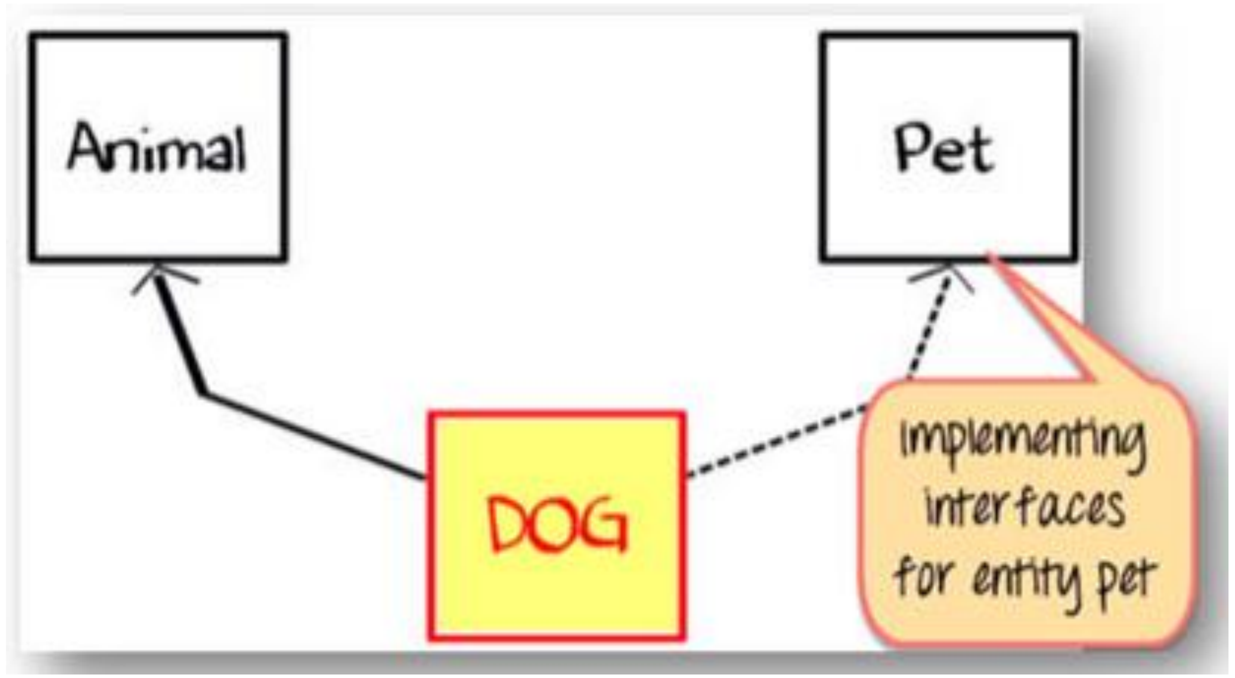
- Another class "Combo drive" is inheriting both CD and DVD (see image below). Which play method should it inherit? This may cause serious design issues. Hence, **Java does not allow multiple inheritance**.



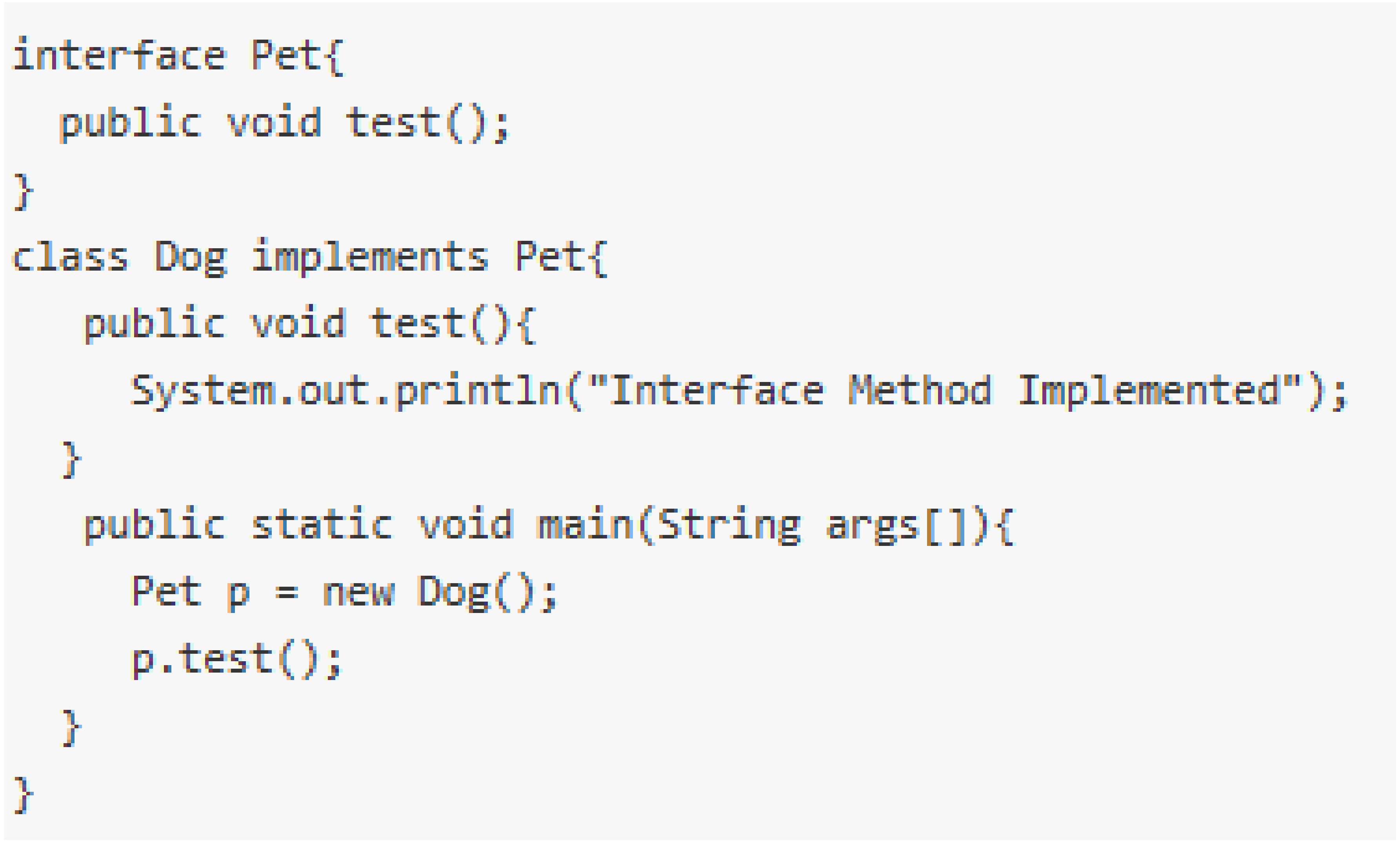
- Suppose you have a requirement where class "dog" inheriting class "animal" and "Pet" (see image below). But you cannot extend two classes in Java. So what would you do? The solution is Interface.



- The rulebook for interface says,
- An interface is 100% abstract class and has only abstract methods.
- Class can implement any number of interfaces.
- Class Dog can extend to class "Animal" and implement interface as "Pet".



Example:



```
interface Pet{
  public void test();
}
class Dog implements Pet{
   public void test(){
     System.out.println("Interface Method Implemented");
  }
   public static void main(String args[]){
     Pet p = new Dog();
     p.test();
  }
}
```

- A class that implements an interface has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

```
interface MyInterface {
  public void method1();
  public void method2();
}

class Demo implements MyInterface {
/* This class must have to implement both the abstract methods * else you will get compilation error */
   public void method1() {
        System.out.println("implementation of method1");
   }
  public void method2() {
       System.out.println("implementation of method2");
   }
   public static void main(String [] arg) {
   MyInterface obj = new Demo();
   obj.method1();
   }
}
```

Interface and Inheritance

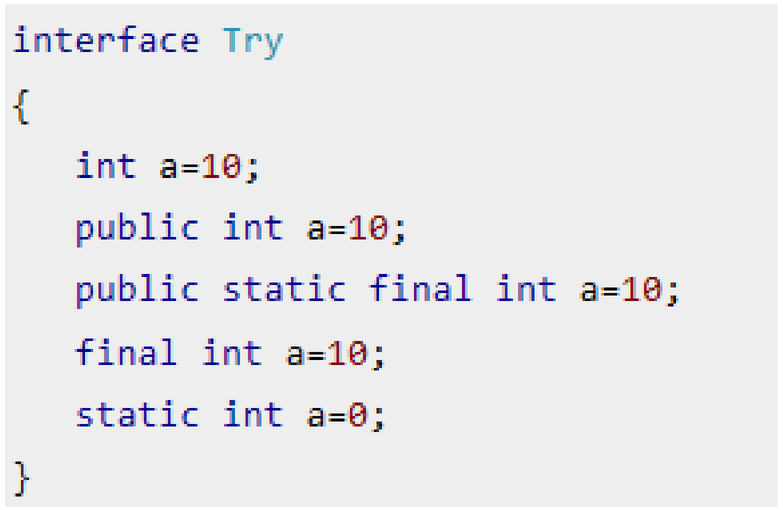
- An interface cannot implement another interface. It has to extend the other interface. If we have two interfaces Inf1 and Inf2 where Inf2 extends Inf1, a class that implements Inf2 has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

```
interface Inf1{
public void method1();
 }
interface Inf2 extends Inf1 {
public void method2();
}
public class Demo implements Inf2{
public void method1(){
System.out.println("method1");
 }
public void method2(){
System.out.println("method2");
 }
```

```
public static void main(String args[]){
Inf2 obj = new Demo();
obj.method2();
}
}
```

Key points about interfaces:

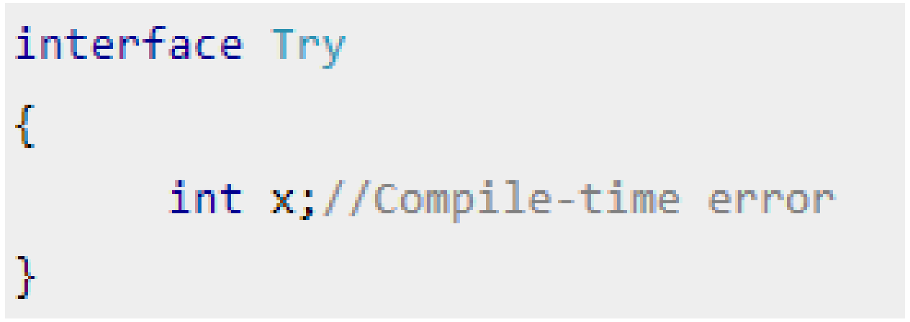
1) We can’t instantiate an interface in java. That means we cannot create the object of an interface
2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.
3) “**implements**” keyword is used by classes to implement an interface.
4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
6) Interface cannot be declared as private, protected or transient.
7) All the interface methods are by default **abstract and public**.
8) Variables declared in interface are **public, static and final** by default.



```java
interface Try
{
   int a=10;
   public int a=10;
   public static final int a=10;
   final int a=10;
   static int a=0;
}
```

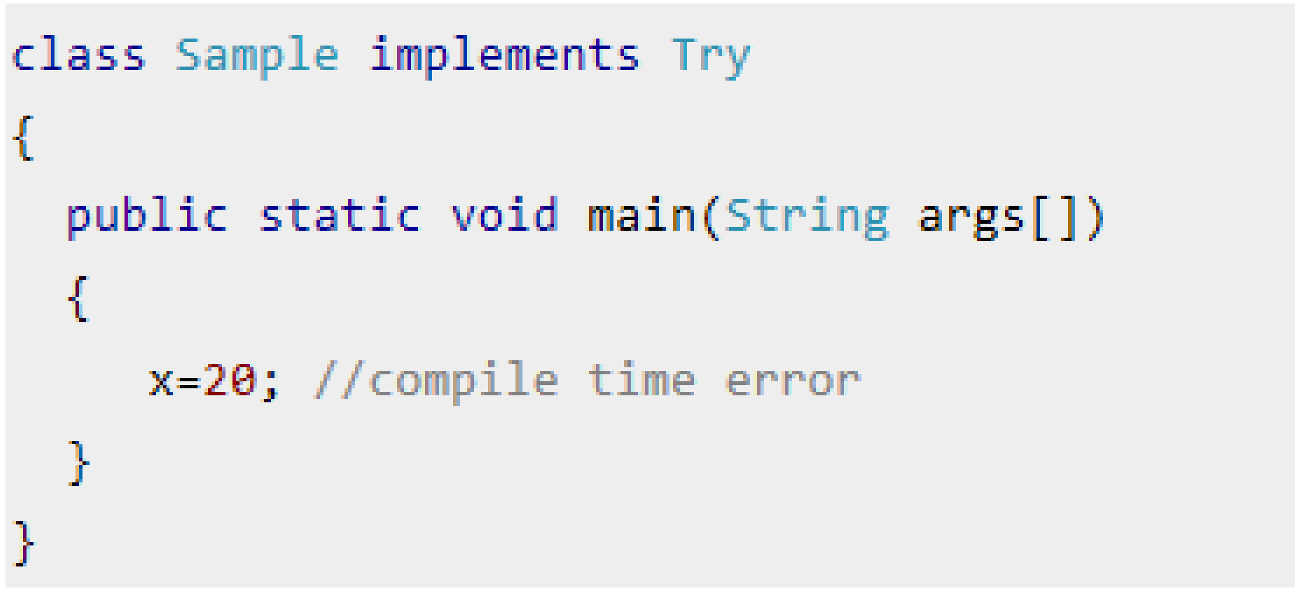
- All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.



```java
interface Try
{
       int x;//Compile-time error
}
```

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.



```
class Sample implements Try
{
  public static void main(String args[])
  {
     x=20; //compile time error
  }
}
```

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

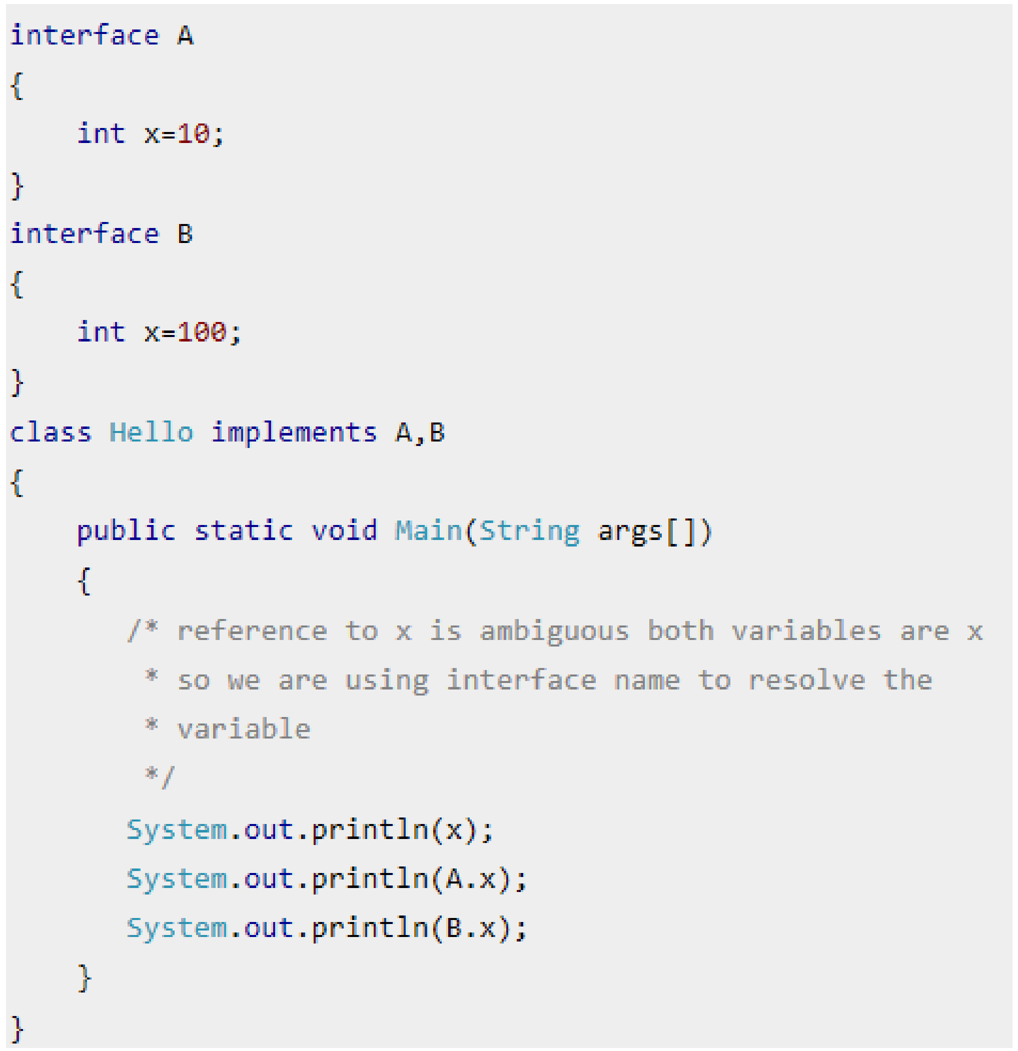
12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

14) A class cannot implement two interfaces that have methods with same name but different return type.

15) Variable names conflicts can be resolved by interface name

Variable names conflict:



```
interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
       /* reference to x is ambiguous both variables are x
        * so we are using interface name to resolve the
        * variable
        */
       System.out.println(x);
       System.out.println(A.x);
       System.out.println(B.x);
    }
}
```

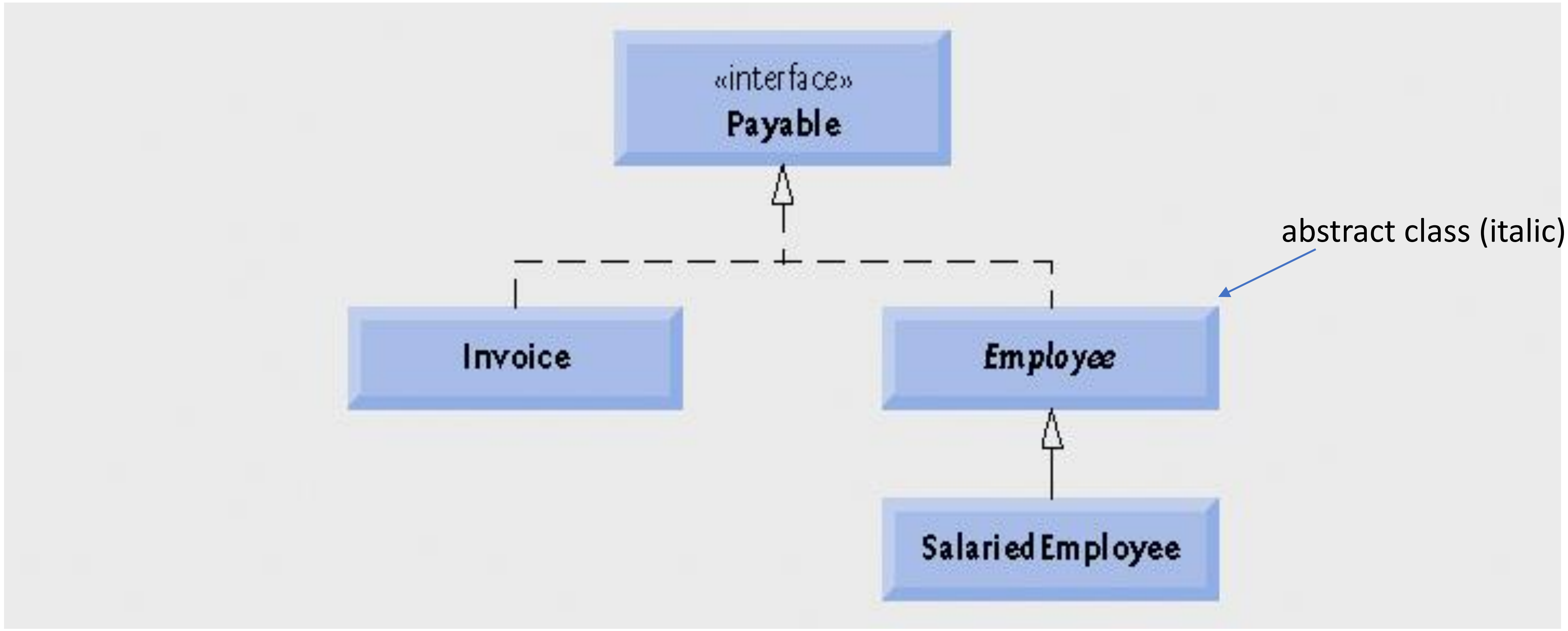
Difference between Class and Interface

Class	Interface
In class, you can instantiate variable and create an object.	In an interface, you can't instantiate variable and create an object.
Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
The access specifiers used with classes are private, protected and public.	In Interface only one specifier is used- Public.

Example: Developing a `Payable` Hierarchy

- `Payable` interface
  - Contains method `getPaymentAmount`
  - Is implemented by the `Invoice` and `Employee` classes
- UML representation of interfaces
  - Interfaces are distinguished from classes by placing the word “interface” in guillemets (« and ») above the interface name
  - The relationship between a class and an interface is known as realization
    - A class “realizes” the methods of an interface

`Payable` interface hierarchy UML class diagram.



- Hollow arrowhead pointing from the implementing class to the interface
- Concrete class SalariedEmployee extends Employee and inherits it's superclass realization relationship with interface Payable

Example: Interface declaration

```java
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Declare interface **`Payable`**

Declare **`getPaymentAmount`** method which is implicitly **`public`** and **`abstract`**

```java
// Fig. 10.12: Invoice.java
// Invoice class implements Payable.

public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part;
   } // end method setPartNumber

```

Class **`Invoice`** implements interface **`Payable`**

```java
   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description;
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
   } // end method setQuantity

   // get quantity
   public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

   // set price per item
   public void setPricePerItem( double price )
   {
      pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
   } // end method setPricePerItem

   // get price per item
   public double getPricePerItem()
   {
      return pricePerItem;
   } // end method getPricePerItem

   // return String representation of Invoice object
   public String toString()
   {
      return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
         "invoice", "part number", getPartNumber(), getPartDescription(),
         "quantity", getQuantity(), "price per item", getPricePerItem() );
   } // end method toString

   // method required to carry out contract with interface Payable
   public double getPaymentAmount()
   {
      return getQuantity() * getPricePerItem(); // calculate total c
   } // end method getPaymentAmount
} // end class Invoice
```

Declare **`getPaymentAmount`** to fulfill contract with interface **`Payable`**

- A class can implement as many interfaces as it needs
    - Use a comma-separated list of interface names after keyword implements
        - Example: `public class` *ClassName* `extends` *SuperclassName* `implements` *FirstInterface*`,` *SecondInterface*`,` …

Example: Creating abstract class Employee that implements interface Payable

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

Class **`Employee`** implements interface **`Payable`**

```java
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

`getPaymentAmount` method is not implemented here

Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy

- Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface
  - A reference to a subclass object can be assigned to an interface variable if the superclass implements that interface

Software Engineering Observation

- Inheritance and interfaces are similar in their implementation of the “is-a” relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any subclasses of a class that implements an interface also can be thought of as an object of the interface type.

```java
// Fig. 10.14: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      weeklySalary = salary < 0.0 ? 0.0 : salary;
   } // end method setWeeklySalary

```

Class **`SalariedEmployee`** extends class **`Employee`** (which implements interface **`Payable`**)

```
    // return salary
    public double getWeeklySalary()
    {
       return weeklySalary;
    } // end method getWeeklySalary

    // calculate earnings; implement interface Payable method that was
    // abstract in superclass Employee
    public double getPaymentAmount()
    {
       return getWeeklySalary();
    } // end method getPaymentAmount

    // return String representation of SalariedEmployee object
    public String toString()
    {
       return String.format( "salaried employee: %s\n%s: $%,.2f",
          super.toString(), "weekly salary", getWeeklySalary() );
    } // end method toString
 } // end class SalariedEmployee
```

Declare **`getPaymentAmount`** method instead of **`earnings`** method

Software Engineering Observation

- The “is-a” relationship that exists between superclasses and subclasses, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives a variable of a superclass or interface type, the method processes the object received as an argument polymorphically.

Software Engineering Observation

- Using a superclass reference, we can polymorphically invoke any method specified in the superclass declaration (and in class `Object`). Using an interface reference, we can polymorphically invoke any method specified in the interface declaration (and in class `Object`).

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

public class PayableInterfaceTest
{
   public static void main( String args[] )
   {
      // create four-element Payable array
      Payable payableObjects[] = new Payable[ 4 ];

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

```

Declare array of **Payable** variables

Assigning references to **Invoice** objects to **Payable** variables

Assigning references to **SalariedEmployee** objects to **Payable** variables

```
      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

Call **`toString`** and **`getPaymentAmount`** methods polymorphically

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Interface cannot implement interface

Why?

- **Implements** denotes defining an **implementation** for the methods of an interface. However **interfaces** have no implementation. Therefore it is not possible.

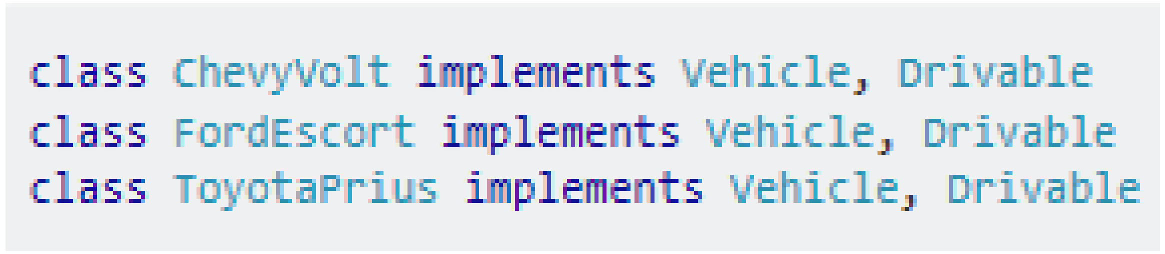
Interface can extends more than one interfaces

- It is possible for interface to extend more than one interfaces to inherit all the methods from the parent interfaces.
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.
- e.g.

```
interface Maininterface extends inter1, inter2, inter3 {
        //methods
}
```

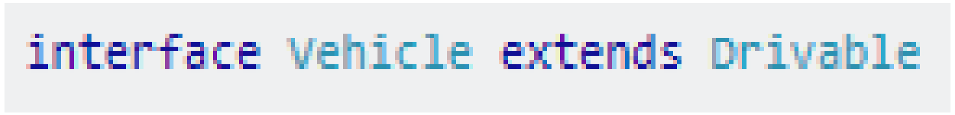
What circumstances do we extend an interface from an interface

- If you have a interface Vehicle and an interface Drivable it stands to reason that all vehicles are drivable. Without interface inheritance every different kind of car class is going to require:

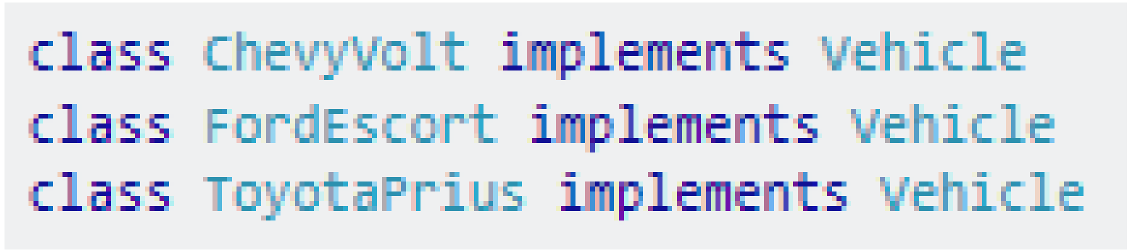


```
class ChevyVolt implements Vehicle, Drivable
class FordEscort implements Vehicle, Drivable
class ToyotaPrius implements Vehicle, Drivable
```

- Since all vehicles are drivable so it's easier to just have:



```
interface Vehicle extends Drivable
```



```
class ChevyVolt implements Vehicle
class FordEscort implements Vehicle
class ToyotaPrius implements Vehicle
```

If you don't want B to be implemented without implementing A,
you can make B extends A.

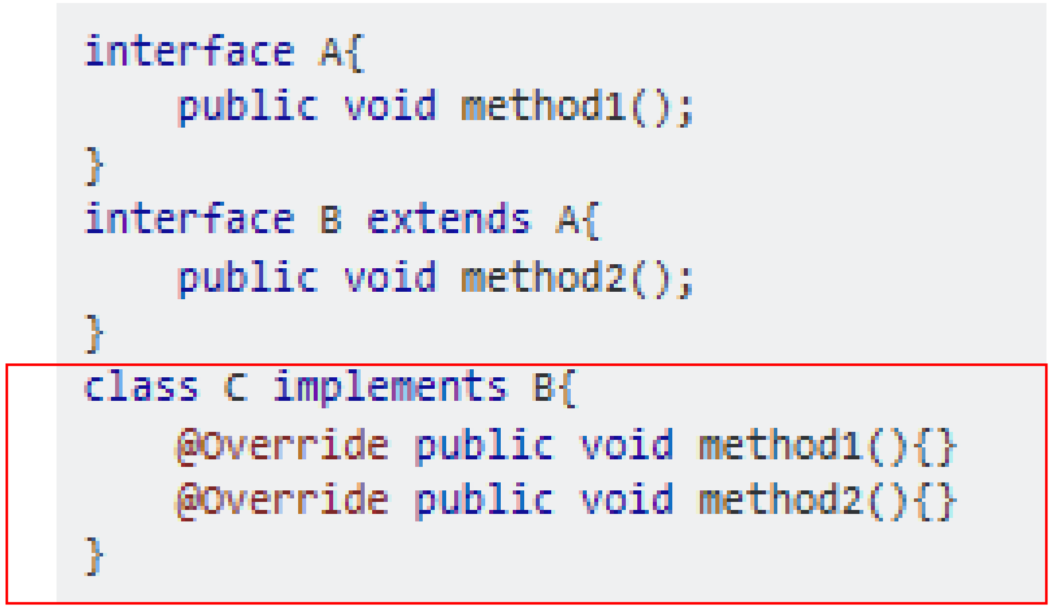
Example:

```
interface LivingThing{
 public void eat();
}
interface Dog extends LivingThing{
 public void Bark();
}
```

It is possible to be a LivingThing without being a dog but it is
not possible to be a dog without being a LivingThing. So if it can bark it
can also eat but the opposite is not always true.

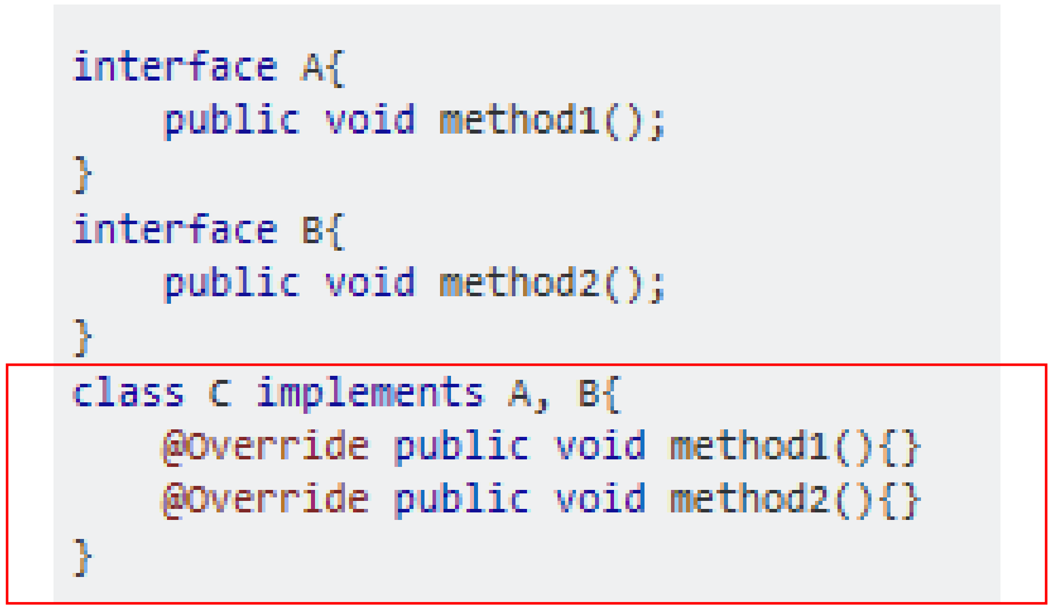
Discussions of two set of codes: similarity and differences

Figure 1



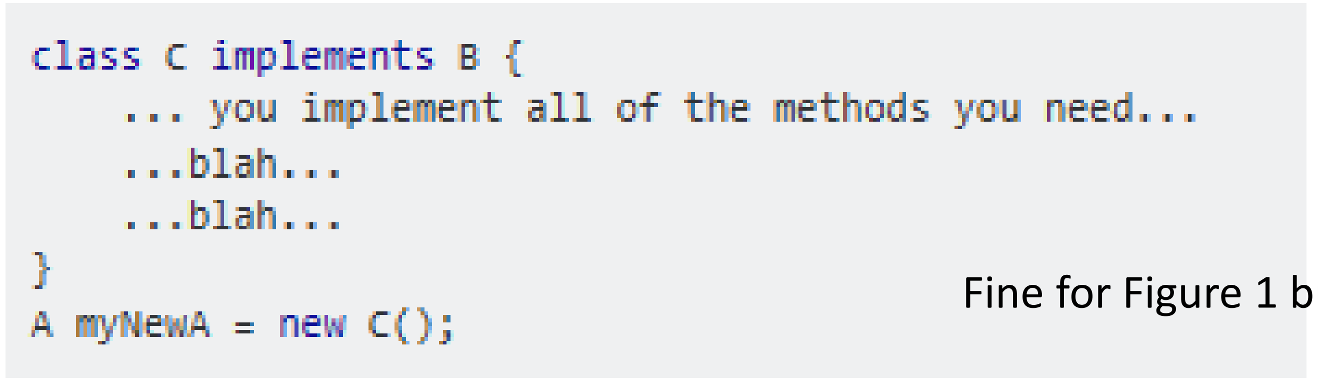
```
interface A{
    public void method1();
}
interface B extends A{
    public void method2();
}
class C implements B{
    @Override public void method1(){}
    @Override public void method2(){}
}
```

Figure 2



```
interface A{
    public void method1();
}
interface B{
    public void method2();
}
class C implements A, B{
    @Override public void method1(){}
    @Override public void method2(){}
}
```

If you change the highlighted part to the following:



```
class C implements B {
    ... you implement all of the methods you need...
    ...blah...
    ...blah...
}
A myNewA = new C();
```

Fine for Figure 1 but error for Figure 2, why?

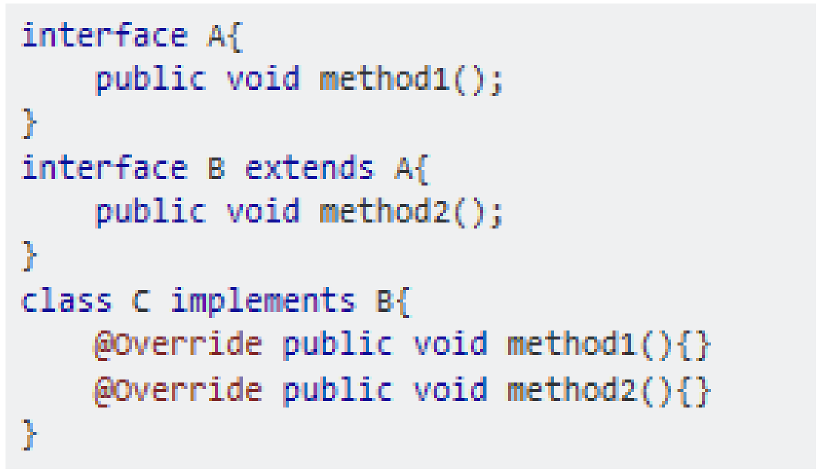
In Figure 1

- In Figure 1, your new interface B is defined to extend from A, so going forward, *any* class that implements B *automatically* implements A. Basically, you're telling the compiler "here's what it means to be an A, here's what it means to be a B, and oh, **by the way, all B's are also A's**!"

In Figure 2

But, for Figure 2, you don't declare B to extend A. When you try
to assign an instance of (the second kind of) C into a reference for A,
 it would complain, because you *haven't* told the complier that
"all B's are really A's." (i.e. there is no relation between A and C

- One more thing to take note:



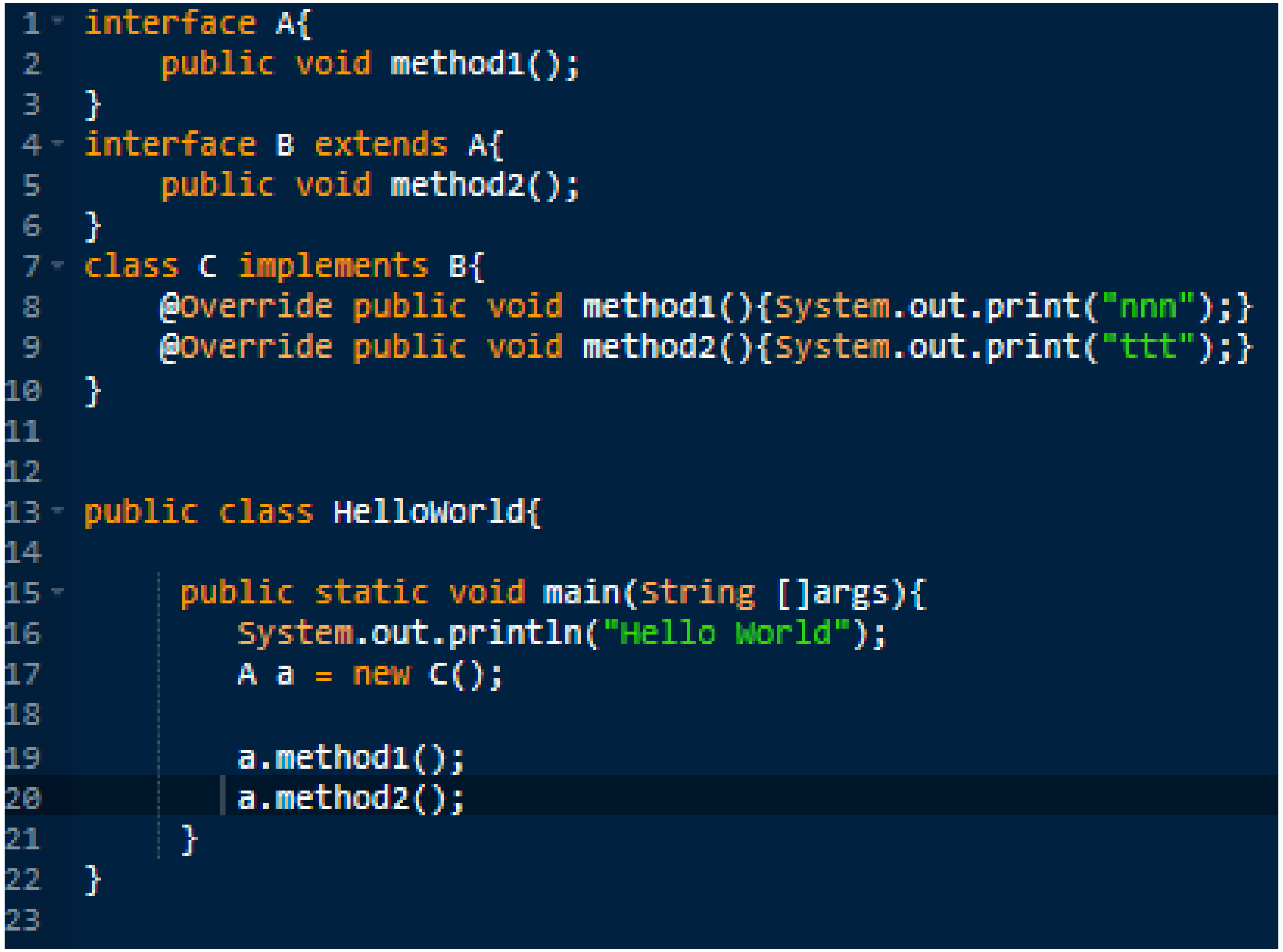
```
interface A{
    public void method1();
}
interface B extends A{
    public void method2();
}
class C implements B{
    @Override public void method1(){}
    @Override public void method2(){}
}
```

if you declare

A a = new C();

- you CANNOT call method2(); because interface A knows nothing about elements in interface B even you implements methods in class C.

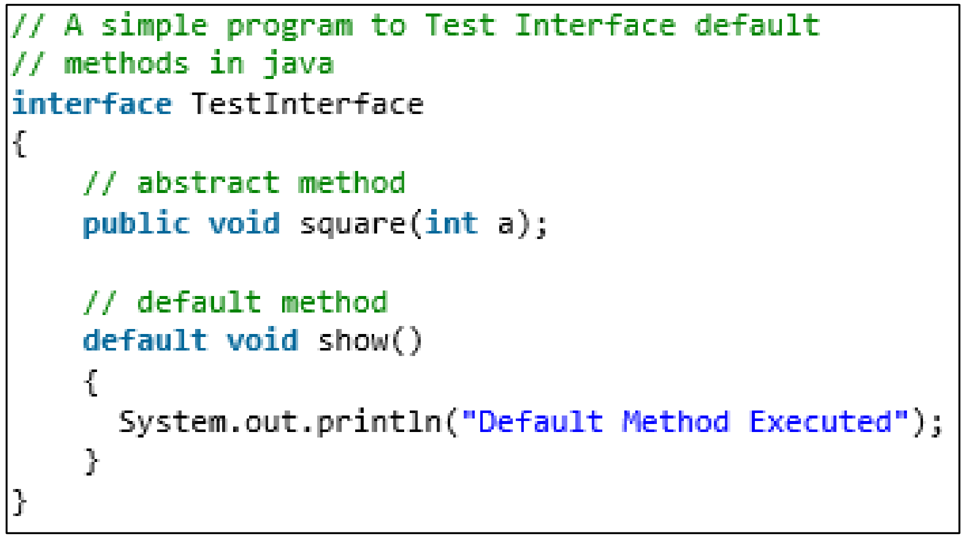
Try to run the following and observe the output:



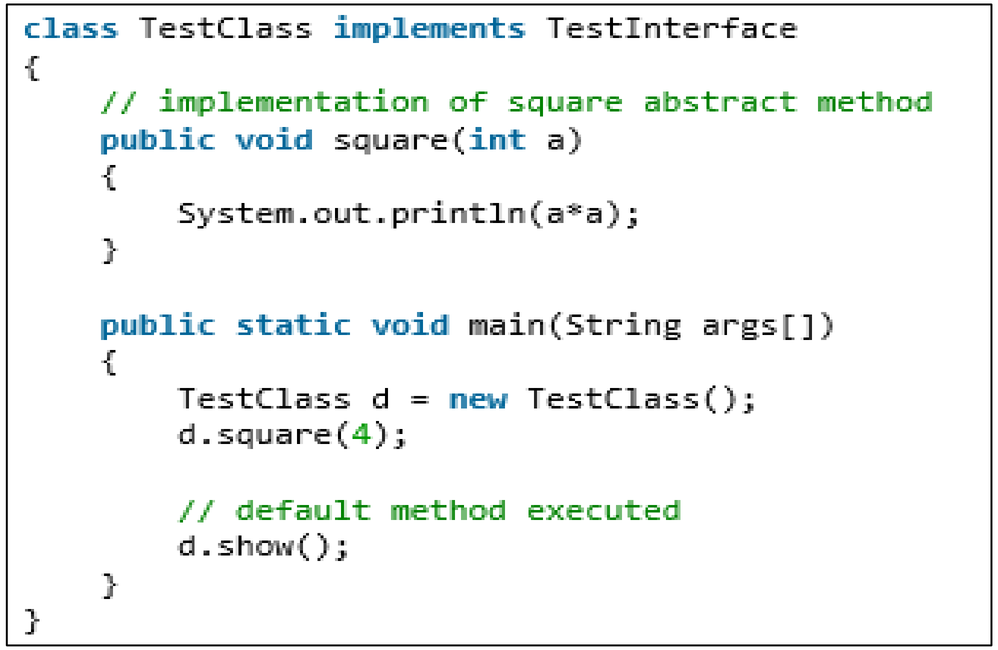
```
interface A{
    public void method1();
}
interface B extends A{
    public void method2();
}
class C implements B{
    @Override public void method1(){System.out.print("nnn");}
    @Override public void method2(){System.out.print("ttt");}
}


public class HelloWorld{

     public static void main(String []args){
        System.out.println("Hello World");
        A a = new C();

        a.method1();
        a.method2();
     }
}
```

Default Methods In Java 8

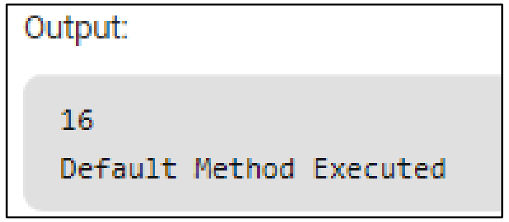
- Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class.
- So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface.
- To overcome this 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.



```
// A simple program to Test Interface default
// methods in java
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}
```

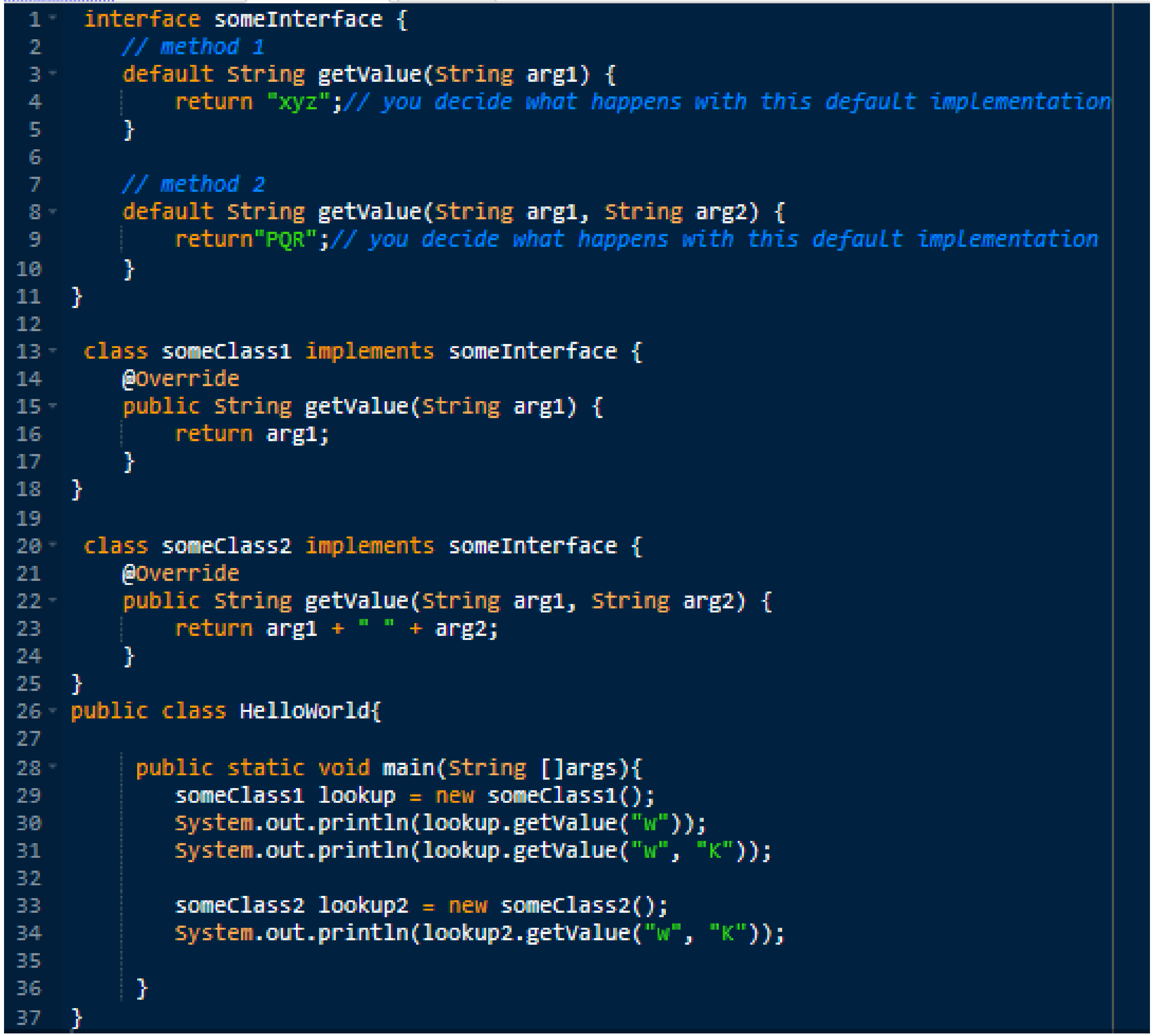


```
class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // default method executed
        d.show();
    }
}
```



Output:

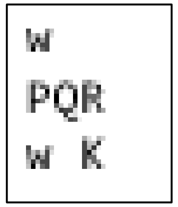
```
16
Default Method Executed
```



```java
interface someInterface {
    // method 1
    default String getValue(String arg1) {
        return "xyz";// you decide what happens with this default implementation
    }

    // method 2
    default String getValue(String arg1, String arg2) {
        return"PQR";// you decide what happens with this default implementation
    }
}

class someClass1 implements someInterface {
    @Override
    public String getValue(String arg1) {
        return arg1;
    }
}

class someClass2 implements someInterface {
    @Override
    public String getValue(String arg1, String arg2) {
        return arg1 + " " + arg2;
    }
}
public class HelloWorld{

     public static void main(String []args){
        someClass1 lookup = new someClass1();
        System.out.println(lookup.getValue("W"));
        System.out.println(lookup.getValue("W", "K"));

        someClass2 lookup2 = new someClass2();
        System.out.println(lookup2.getValue("W", "K"));

     }
}
```

Using default for method overloading in interface

Output:



```
W
PQR
W  K
```