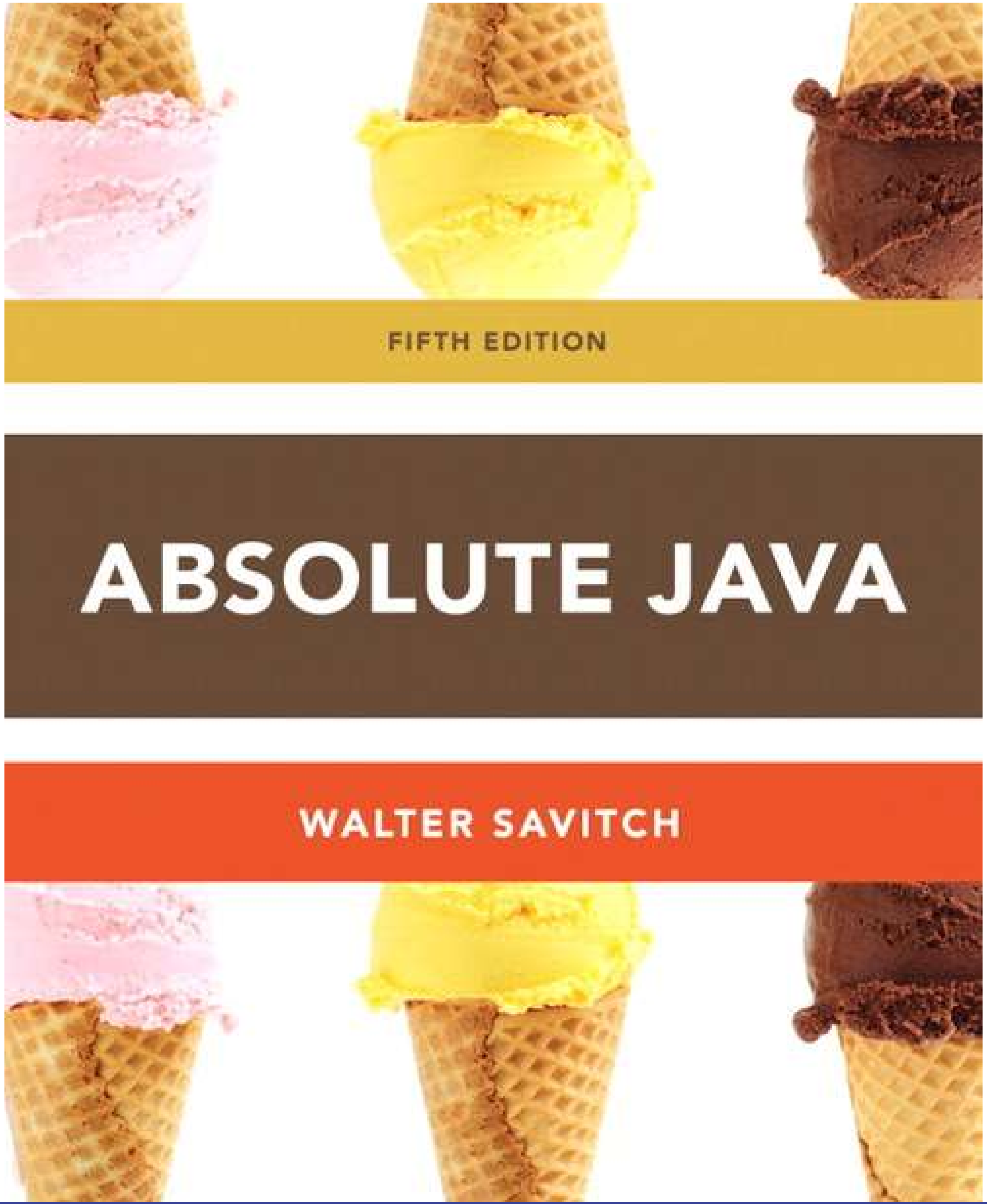
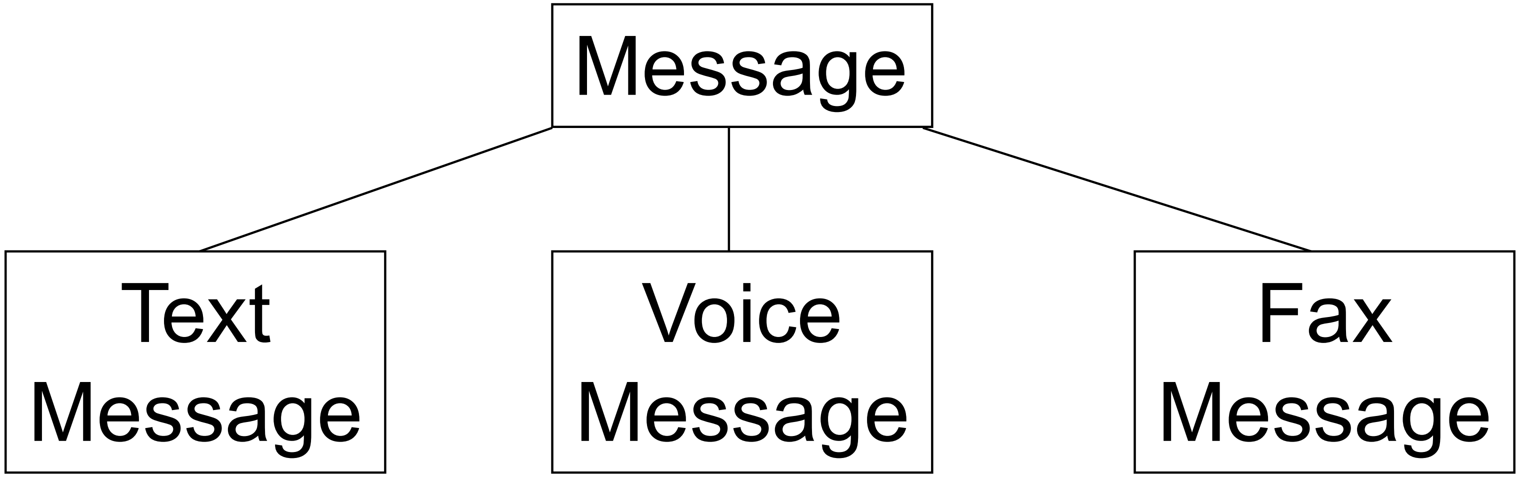


Chapter 13

Interfaces

Abstract Class

- A class that cannot instantiate objects.



Message

Text Message

Voice Message

Fax Message

```
Public abstract class Message {
}
```

Abstract Class

- An abstract class used as supertype
- An object cannot be created from an abstract class
- An array of the abstract type is used to contain objects of the concrete subclasses

From abstract class

To Interface

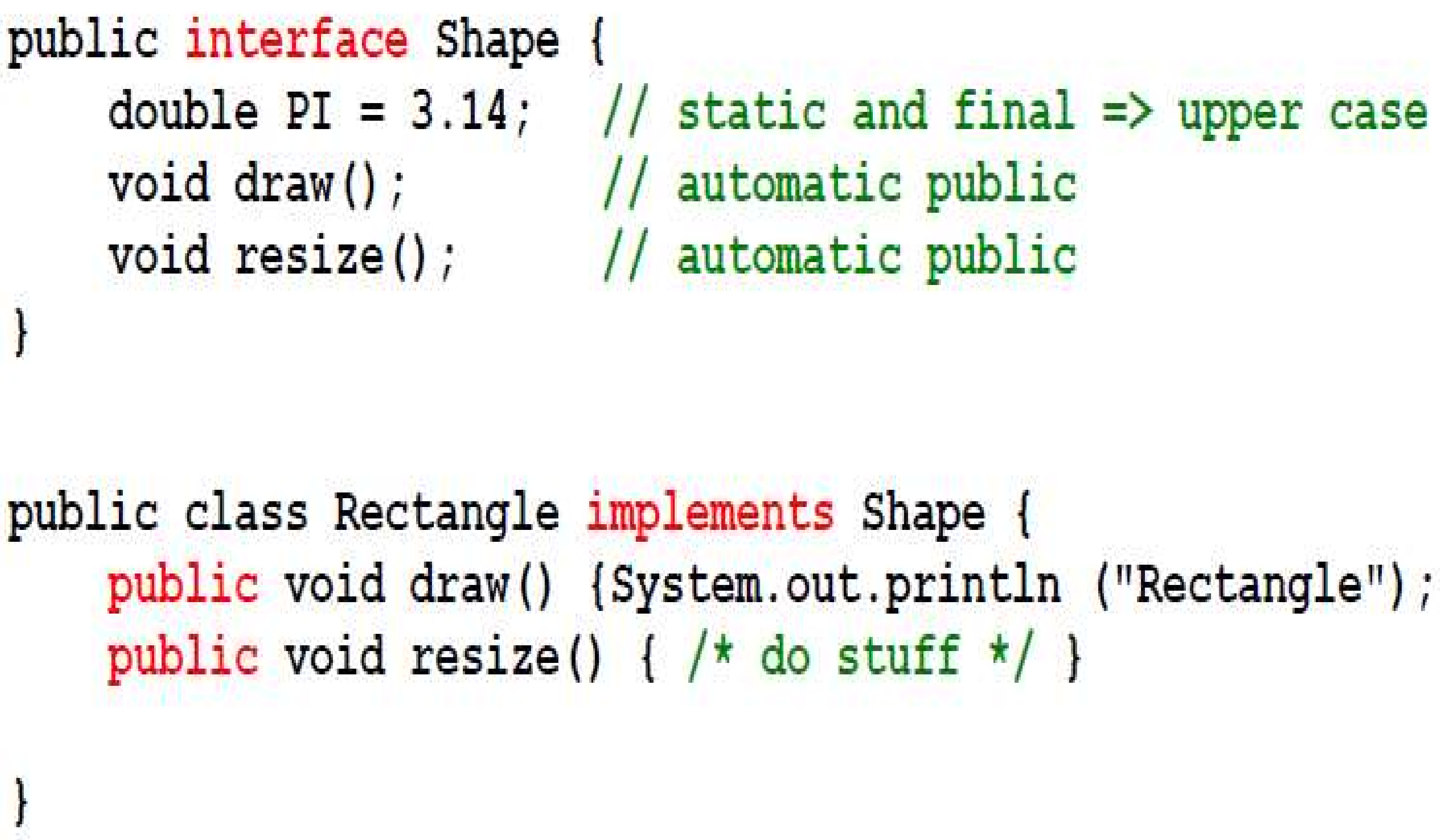
What is an Interface?

- An interface is a collection of constants and method declarations
- An interface describes a set of methods that can be called on an object
- The method declarations do not include an implementation
  - there is no method body

What is an Interface?

- A child class that *extends* a parent class can also *implement* an interface to gain some additional behavior
- Implementing an interface is a “promise” to include the specified method(s)
- A method in an interface cannot be made *private*

Example



```
public interface Shape {
    double PI = 3.14;   // static and final => upper case
    void draw();        // automatic public
    void resize();      // automatic public
}


public class Rectangle implements Shape {
    public void draw() {System.out.println ("Rectangle");
    public void resize() { /* do stuff */ }


}
```

When A Class Definition *Implements* An Interface:

- It must implement each method in the interface
- Each method must be *public* (even though the interface might not say so)
- Constants from the interface can be used as if they had been defined in the class (They should not be re-defined in the class)

Declaring Constants with Interfaces

- Interfaces can be used to declare constants used in many class declarations
  - These constants are implicitly `public`, `static` and `final`

Creating and Using Interfaces

- Declaration begins with **`interface`** keyword
- Classes **`implement`** an interface (and its methods)
- Contains `public` `abstract` methods
  - Classes (that `implement` the interface) must implement these methods

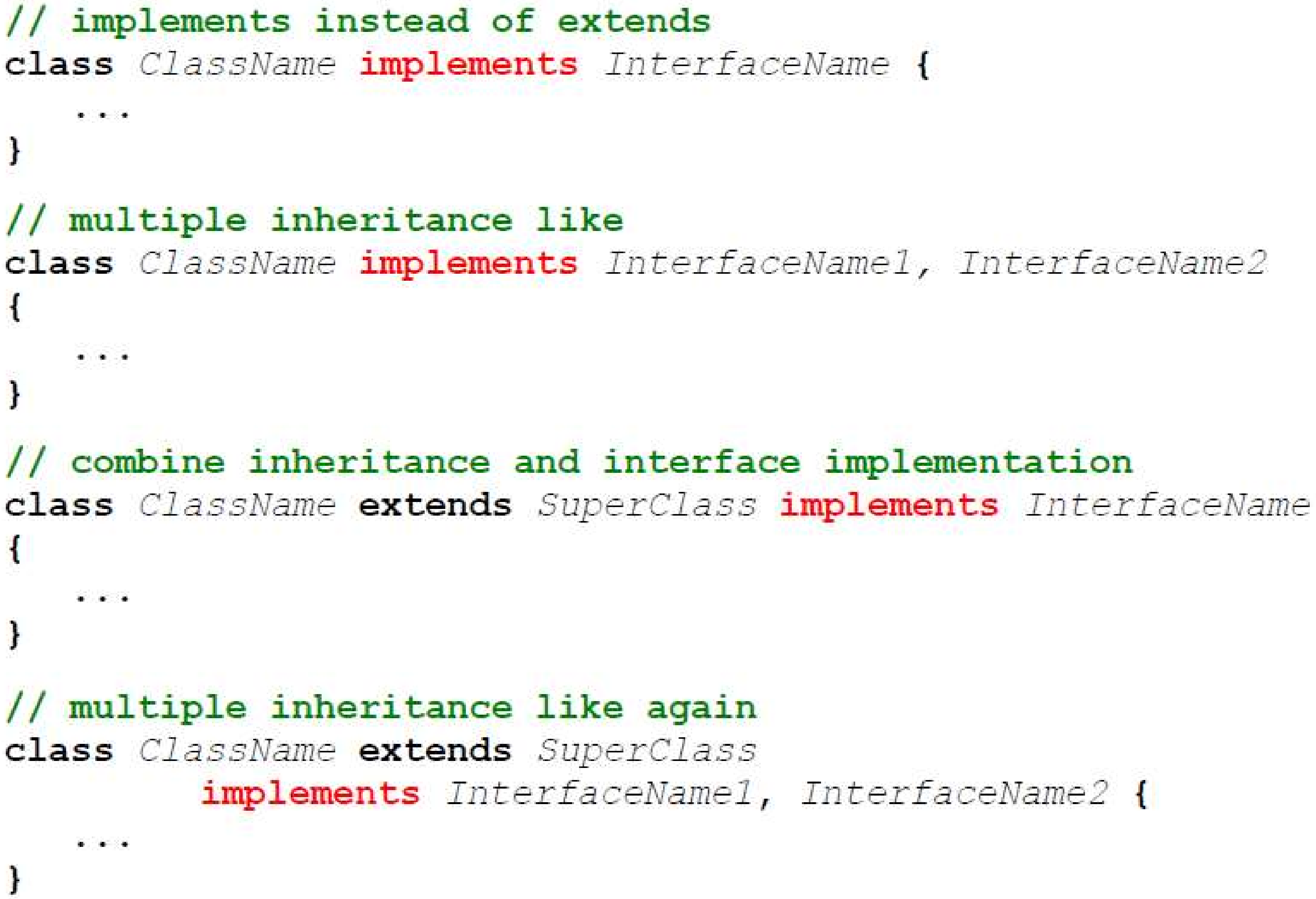
Interface Body

- The interface body contains method declarations for **ALL** the methods included in the interface.
- A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it.
- All methods declared in an interface are implicitly <u>public</u> and <u>abstract</u>.

Implement an Interface

- An interface defines a protocol of behavior.
- A class that implements an interface adheres to the protocol defined by that interface.
- To declare a class that implements an interface, include an implements clause in the class declaration.

Rules



```
// implements instead of extends
class ClassName implements InterfaceName {
    ...
}

// multiple inheritance like
class ClassName implements InterfaceName1, InterfaceName2
{
    ...
}

// combine inheritance and interface implementation
class ClassName extends SuperClass implements InterfaceName
{
    ...
}

// multiple inheritance like again
class ClassName extends SuperClass
          implements InterfaceName1, InterfaceName2 {
    ...
}
```

Why Use Interfaces

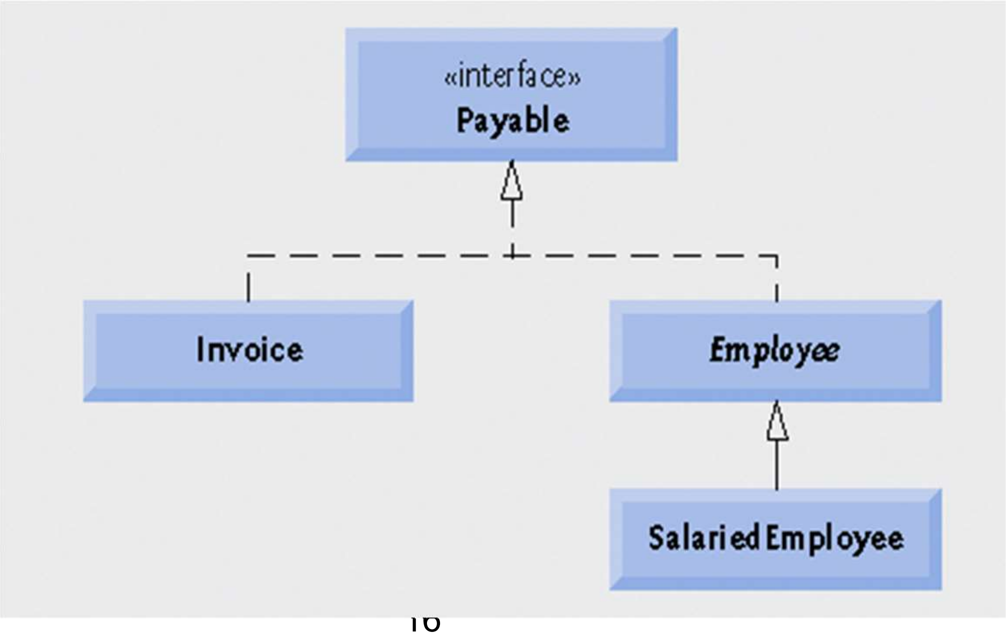
- Java has single inheritance, only
- This means that a child class inherits from only one parent class
- Sometimes multiple inheritance would be convenient
- *Interfaces* give Java some of the advantages of multiple inheritance without incurring the disadvantages

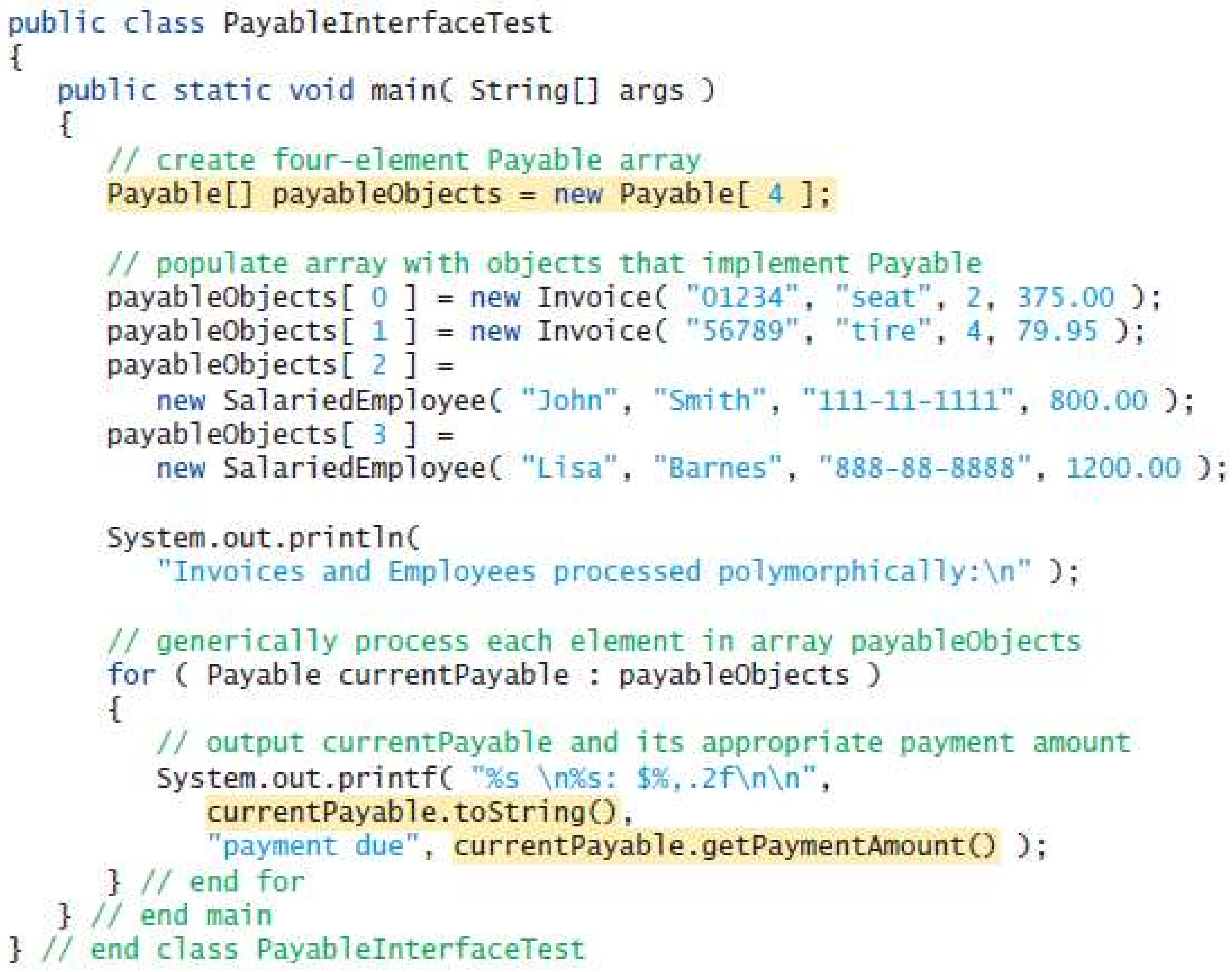
Why Use Interfaces

- Provide capability for unrelated classes to implement a set of common methods
- Define and standardize ways people and systems can interact
- Interface specifies what operations must be permitted
- Does not specify how performed

Case Study: A **Payable** Hierarchy

- Payable interface
  - Contains method getPaymentAmount
  - Is implemented by the Invoice and Employee classes





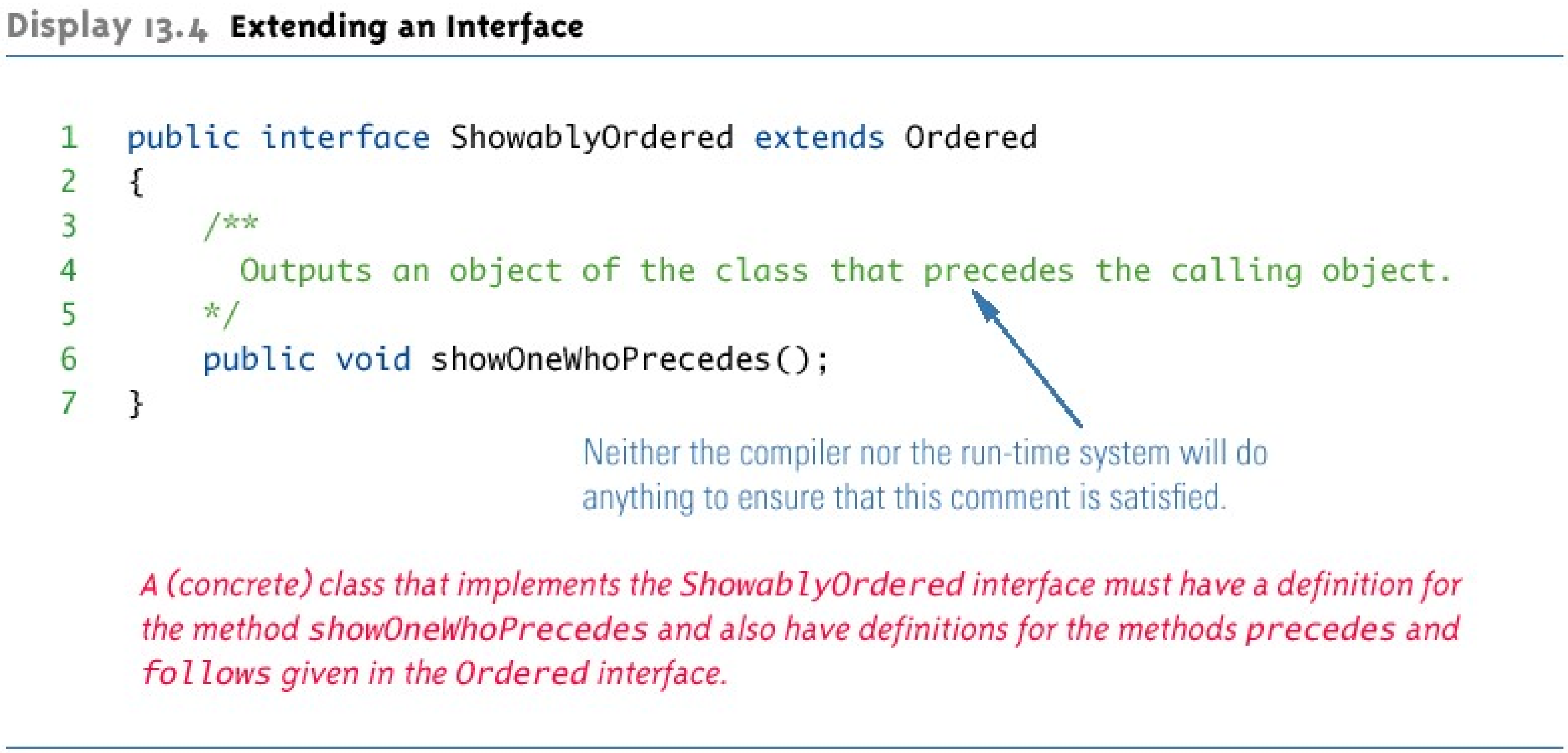
```java
public class PayableInterfaceTest
{
   public static void main( String[] args )
   {
      // create four-element Payable array
      Payable[] payableObjects = new Payable[ 4 ];

      // 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" );

      // 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
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase

    `extends` ***`BaseInterfaceName`***
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Extending an Interface



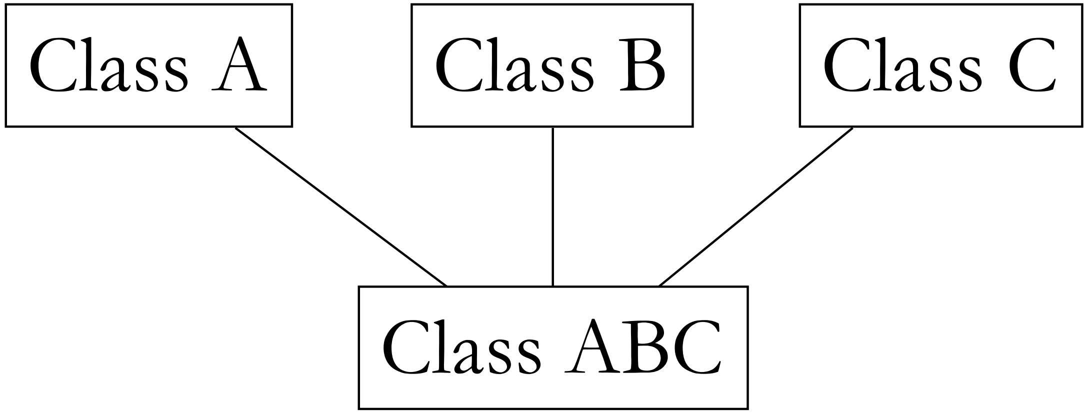
Display 13.4 Extending an Interface

```
public interface ShowablyOrdered extends Ordered
{
    /**
      Outputs an object of the class that precedes the calling object.
    */
    public void showOneWhoPrecedes();
}
```

Neither the compiler nor the run-time system will do anything to ensure that this comment is satisfied.

A (concrete) class that implements the `ShowablyOrdered` interface must have a definition for the method `showOneWhoPrecedes` and also have definitions for the methods `precedes` and `follows` given in the `Ordered` interface.

Multiple Inheritance



Class ABC inherits all variables and methods from Class A, Class B, and Class C.

Java does NOT support multiple inheritances.

However, you can use interface to implement the functionality of multiple inheritance.

The `Comparable` Interface

- The **`Comparable`** interface is in the **`java.lang`** package, and so is automatically available to any program
- It has only the following method heading that must be implemented:

```
public int compareTo(Object other);
```

- It is the programmer's responsibility to follow the semantics of the **`Comparable`** interface when implementing it

The **`Comparable`** Interface Semantics

- The method **`compareTo`** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter **`other`** is not of the same type as the class being defined, then a **`ClassCastException`** should be thrown

The **`Comparable`** Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

The `Serializable` Interface

- An extreme but commonly used example of an interface is the `Serializable` interface
  - It has no method headings and no defined constants: It is completely empty
  - It is used merely as a type tag that indicates to the system that it may implement file I/O in a particular way

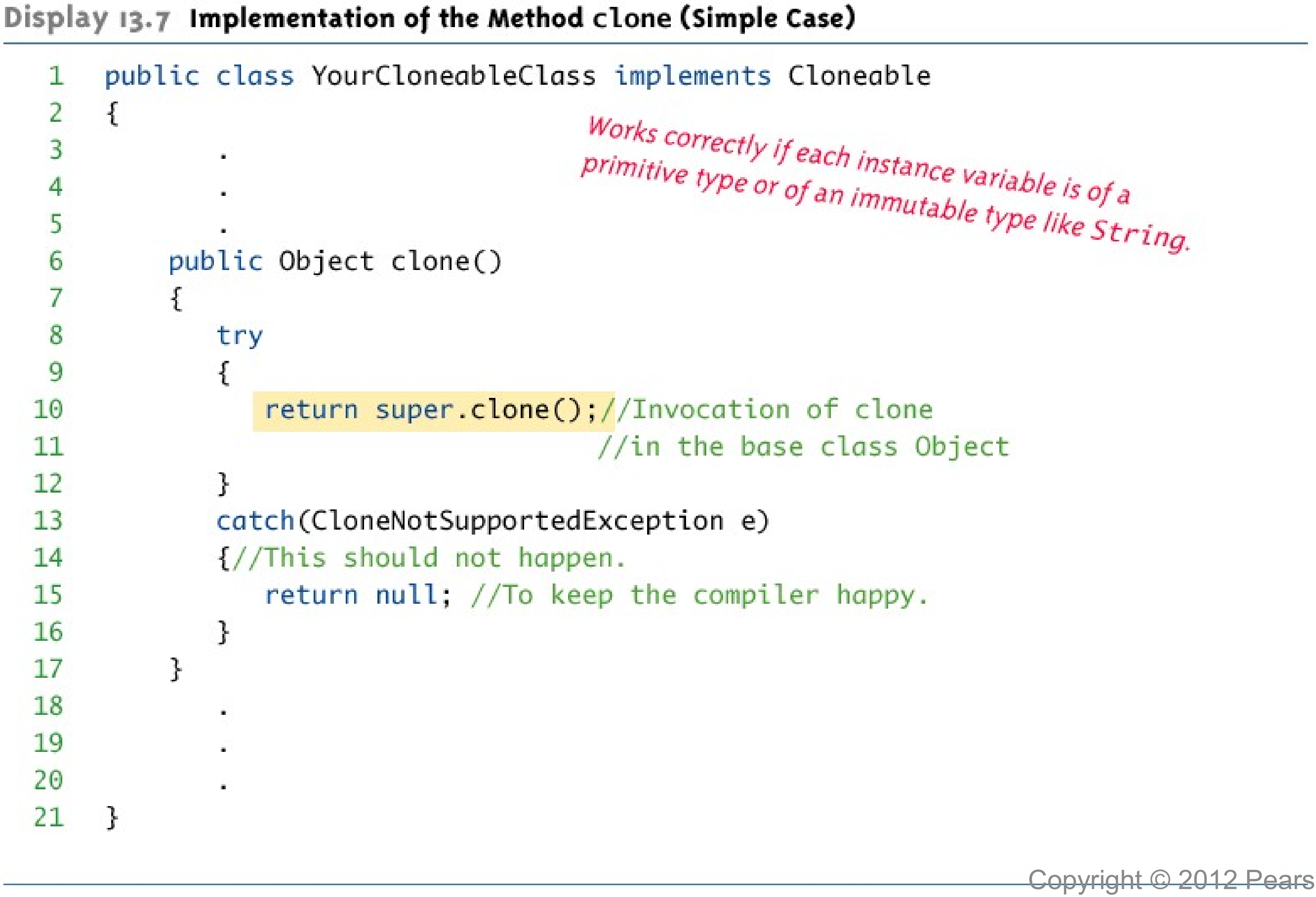
The **`Cloneable`** Interface

- The **`Cloneable`** interface is another unusual example of a Java interface
  - It does not contain method headings or defined constants
  - It is used to indicate how the method **`clone`** (inherited from the **`Object`** class) should be used and redefined

The `Cloneable` Interface

- The method `Object.clone()` does a bit-by-bit copy of the object's data in storage
- If the data is all primitive type data or data of immutable class types (such as `String`), then this is adequate
  - This is the simple case
- The following is an example of a simple class that has no instance variables of a mutable class type, and no specified base class
  - So the base class is `Object`

Implementation of the Method **`clone`**: Simple Case



Display 13.7 Implementation of the Method clone (Simple Case)

```
public class YourCloneableClass implements Cloneable
{
       .
       .
       .
    public Object clone()
    {
       try
       {
          return super.clone();//Invocation of clone
                               //in the base class Object
       }
       catch(CloneNotSupportedException e)
       {//This should not happen.
          return null; //To keep the compiler happy.
       }
    }
       .
       .
       .
}
```

Works correctly if each instance variable is of a primitive type or of an immutable type like `String`.

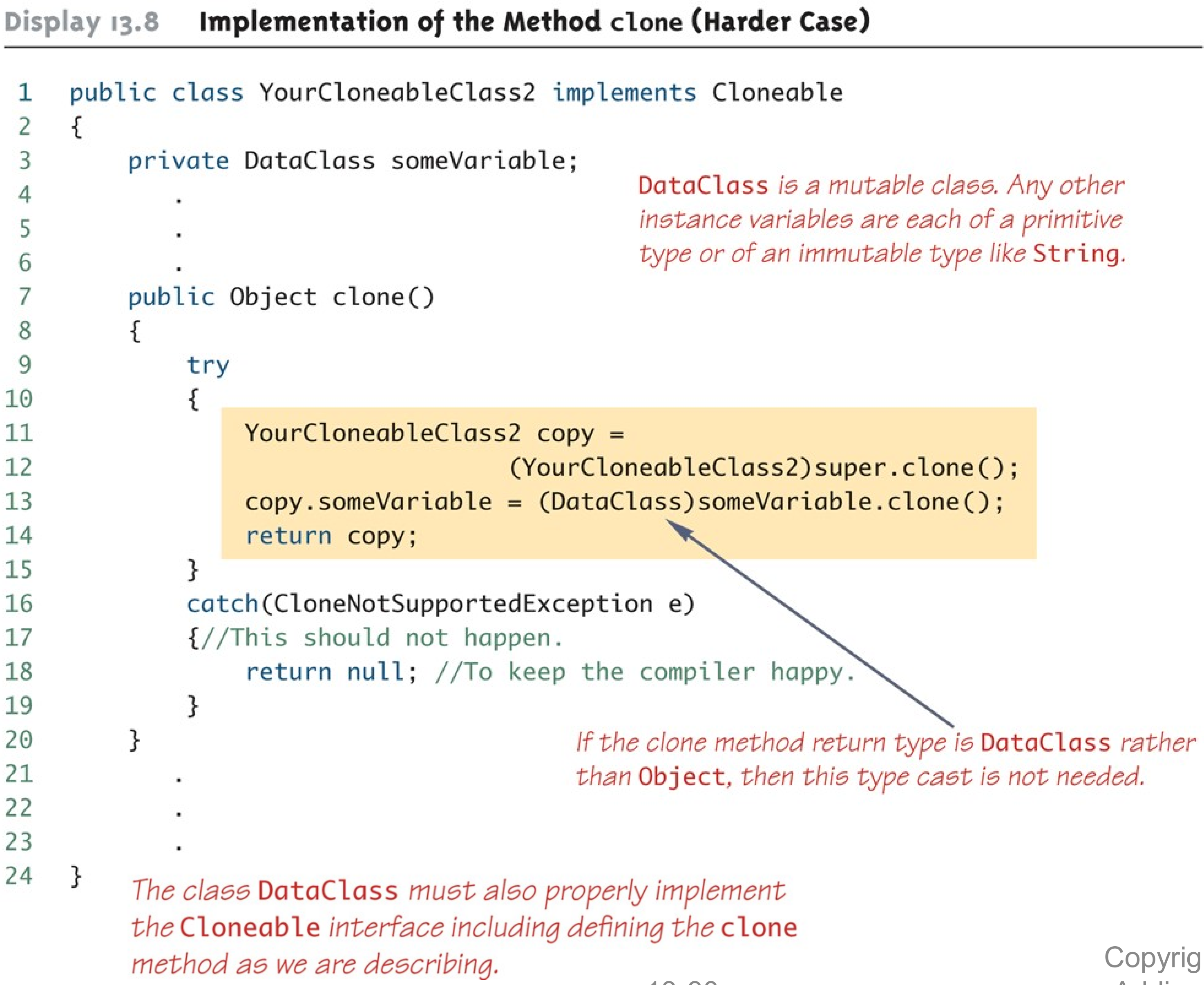
The `Cloneable` Interface

- If the data in the object to be cloned includes instance variables whose type is a mutable class, then the simple implementation of `clone` would cause a *privacy leak*
- When implementing the `Cloneable` interface for a class like this:
  - First invoke the `clone` method of the base class `Object` (or whatever the base class is)
  - Then reset the values of any new instance variables whose types are mutable class types
  - This is done by making copies of the instance variables by invoking *their* clone methods

The `Cloneable` Interface

- Note that this will work properly only if the `Cloneable` interface is implemented properly for the classes to which the instance variables belong
  - And for the classes to which any of the instance variables of the above classes belong, and so on and so forth
- The following shows an example

Implementation of the Method **clone**: Harder Case



Display 13.8 Implementation of the Method clone (Harder Case)

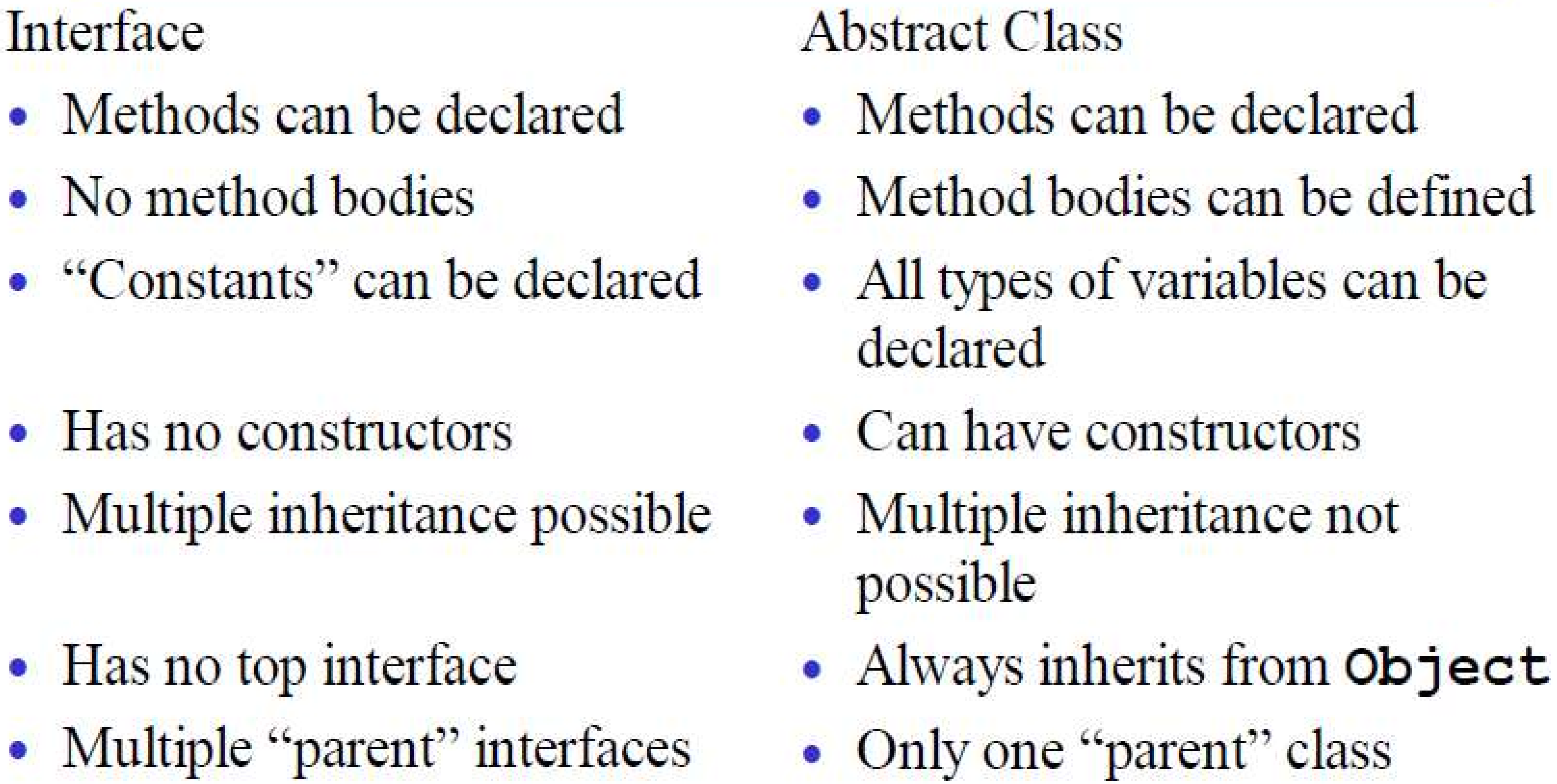
```
public class YourCloneableClass2 implements Cloneable
{
    private DataClass someVariable;
        .
        .
        .
    public Object clone()
    {
        try
        {
            YourCloneableClass2 copy =
                              (YourCloneableClass2)super.clone();
            copy.someVariable = (DataClass)someVariable.clone();
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep the compiler happy.
        }
    }
        .
        .
        .
}
```

`DataClass` is a mutable class. Any other instance variables are each of a primitive type or of an immutable type like `String`.

If the clone method return type is `DataClass` rather than `Object`, then this type cast is not needed.

The class `DataClass` must also properly implement the `Cloneable` interface including defining the `clone` method as we are describing.

Summary (Abstract vs Interface)



Interface	Abstract Class
• Methods can be declared	• Methods can be declared
• No method bodies	• Method bodies can be defined
• “Constants” can be declared	• All types of variables can be declared
• Has no constructors	• Can have constructors
• Multiple inheritance possible	• Multiple inheritance not possible
• Has no top interface	• Always inherits from `Object`
• Multiple “parent” interfaces	• Only one “parent” class

Design guidelines

- Use classes for specialization and generalization
- Use interfaces to add properties to classes.