Compare these two classes. Look for features that they have in common and ways in which they differ.

|  |  |
| --- | --- |
| public class StringIndex                       extends MyTestClass  {    private String myString;    private int myIndex;     public StringIndex( String s )    {      super( 0 );      myString = s;      myIndex = 0;    }     public String subString()    {      return myString.substring( myIndex );    }     public void increment()    {      myIndex++;    }     public void decrement()    {      myIndex--;    }     public void restart()    {      myIndex = 0;    }  } | public class ProductIncrementer                       extends Product  {    public ProductIncrementer()    {      super();    }     public void increment()    {      setProduct( getProduct() + "I" );      System.out.println( "Enlarging!" );    }     public void decrement()    {      setProduct( getProduct() + "d" );    }     public void restart()    {      resetProduct();    }  } |

Both classes are subclasses of other classes, but the direct superclasses in question are not the same. Among the methods in each class are three called increment, decrement, and restart. The methods with these names in one class appear to do very different things than the corresponding methods in the other class. It is noticeable, however, that corresponding methods have the same signatures and return data types.

Suppose we want to store a collection of instances of these two classes in a single array and then iterate through the array calling the increment or decrement methods. Would it be possible for us to write an abstract class that both these concrete classes would extend? Perhaps, but since both classes are already subclassed it would likely involve a good deal of work. And depending on the nature of the more distant superclasses higher up the respective hierarchies, above the specified direct superclasses, it could well be impossible without compromising the program design.

In light of our array-iterating desires, we want a way to say, "Look, Java, I know these classes are not necessarily related in any hierarchy, but I only want to use this small number of methods that I know they have in common. How about it?"

Java provides a new data type, called an *interface*, to handle just this situation. The role of an interface is to describe to Java the methods that one or more classes will implement without specifying how the implementations will actually be carried out, and without requiring any relationship to a class hierarchy.

In the case of the above sample classes, we could create a data type called Incrementable, using the keyword interface, like this:

public interface Incrementable   
{   
  public abstract void increment();   
  public abstract void decrement();   
  public abstract void restart();   
}

Notice that the three methods we mentioned as being present in both class definitions at the top of this page appear as abstract methods in the Incrementable interface. In fact, every method that is declared in an interface definition is considered to be implicitly both public and abstract. That being so, Java not only allows us to use the abbreviated form of abstract method declaration (in which an empty body is replaced by a statement-ending semicolon) but it also strongly encourages us to avoid redundancy by omitting the modifier keywords public and abstract. In this case, the definition abbreviates to:

public interface Incrementable   
{   
  void increment();   
  void decrement();   
  void restart();   
}

We now need a way of telling Java that a class contains the methods specified in the interface. We do so using the keyword "implements".

If the definition of a class C overrides all the methods of an interface I and if C's class header includes the clause "implements I", then we say that class C *implements* interface I.

Both conditions are required if C is to be a concrete class; it's not good enough just to override the interface's abstract methods — the implements clause has to be there too. So, for example, we could amend the definition of the StringIndex class on the previous page so as to make its relationship to the Incrementable interface apparent like this:

public class StringIndex    
                    extends MyTestClass   
                    implements Incrementable   
{   
  private String myString;   
  private int myIndex;   
  
  public StringIndex( String s )   
  {   
    super( 0 );   
    myString = s;   
    myIndex = 0;   
  }   
  
  public String subString()   
  {   
    return myString.substring( myIndex );   
  }   
  
  public void increment()   
  {   
    myIndex++;   
  }   
  
  public void decrement()   
  {   
    myIndex--;   
  }   
  
  public void restart()   
  {   
    myIndex = 0;   
  }   
}

By declaring that StringIndex implements Incrementable we make it possible for instances of StringIndex to be stored in variables of type Incrementable, thereby making available all the benefits of polymorphism:

public class MainClass

{

  public static void main( String[] args )

  {

    Incrementable t = new StringIndex( "fred" );

    t.increment();

    t.increment();

    System.out.println( ((StringIndex)t).subString() );

  }

}

Ed

We have seen that both abstract classes and interfaces can have abstract methods. You may therefore be wondering in what other ways interfaces and abstract classes are similar and in what ways they differ. The following table provides just such a comparison:

|  |  |
| --- | --- |
| **Constructors** | |
| An **interface** has no constructor. | An **abstract class** either has one or more constructors that are explicitly defined, or it has a default, no-argument, empty-body constructor supplied by Java. |
| **Methods** | |
| All the methods of an **interface** are automatically public and abstract. (Consequently, Java programmers are strongly encouraged to *omit* the keywords public and abstract from the declaration of each interface method.) | An **abstract class** may have only abstract methods, or a mixture of abstract and non-abstract methods, or only non-abstract methods. (The only effect of including the modifier abstract in the definition of a class that has no abstract methods is to prevent the class from being instantiated.) |
| **Variables** | |
| Any variables that are declared in an **interface** definition are automatically public, static, and final, and they *must* be initialized. (Java programmers are allowed to *omit* any or all of the keywords public, static, and final in such declarations.) The keyword private may not appear in a variable declaration in an interface definition. | An **abstract class** may contain variables of any kind: instance variables, class variables (those marked as static), and class constants (variables marked as static and final). None of them has to be initialized, and they may be public or private. |
| **Hierarchies** | |
| **Interfaces** are totally independent of class hierarchies; the same interface can be implemented by classes that belong to entirely separate class hierarchies.  An **interface** can extend another or be extended by another (examples are provided on the pages that follow), and in consequence belong to a hierarchy of interfaces. Within such a hierarchy, the relationships between interfaces are expressed using words that are similar to those used in class hierarchies: superinterface, direct superinterface, subinterface, direct subinterface. | An **abstract class** can extend another abstract class or be extended by another (abstract or concrete) class, and in consequence belong to a hierarchy of classes.  An **abstract class** can, by means of an appropriate implements clause, implement one or more interfaces. (So too can a concrete class, provided it implements *all* the interface's methods. The pages that follow provide examples.) In particular, a class C that declares that it implements one or more interfaces *must* be marked as abstract if one or more methods from the implemented interfaces do not have corresponding non-abstract definitions either in C or in a superclass of C. |
| **Inheritance** | |
| In order for a concrete class C to implement an **interface**, somewhere within C's definition and the definitions of its superclasses there must appear non-abstract definitions of all the methods declared in the interface and all its superinterfaces. | If an **abstract class** (or a concrete class) C implements an interface, then every subclass of C also implements that interface. |

**Exercise 157**

Which of the following code fragments would compile without error, according to the descriptions in the above table? For those that would not, explain why not.

|  |  |
| --- | --- |
|  | public interface Adder  {    int addLittle( int a );    int addLot();  }   public abstract class AObject implements Adder  {    private int myInt;     public AObject( int x )    {      myInt = x;    }  } |
|  | public interface Upper  {    void methodA();  }   public interface Lower extends Upper  {    void methodB();  }   public class Thing implements Lower  {    public void methodB()    {      System.out.println( "B" );    }  } |
|  | public interface Upper  {    void methodA();  }   public interface Lower extends Upper  {    void methodB();  }   public abstract class ThingB implements Lower  {    public void methodB()    {      System.out.println( "B" );    }  }   public class ThingA extends ThingB  {    public void methodA()    {      System.out.println( "A" );    }  } |
|  | public interface Upper  {    void methodA();  }   public interface Lower extends Upper  {    void methodB();  }   public class ThingA implements Upper  {    public void methodA()    {      System.out.println( "A" );    }  }   public class ThingB extends ThingA                      implements Lower  {    public void methodB()    {      System.out.println( "B" );    }  } |
|  | public interface Alias  {    private String myName;     String changeName( String newName );  }   public class Witness implements Alias  {    public String changeName( String newName )    {      myName = newName;    }  } |
|  | public interface Alias  {    String LASTNAME = "Johnson";     void changeName( String newName );  }   public class Witness  {    private String myName;     public Witness( String name )    {      myName = name;    }     public void changeName( String newName )    {      myName = newName;    }  }   public class ProtectedWitness extends Witness                                implements Alias  {    public ProtectedWitness( String name )    {      super( name );    }     public void changeName( String newName )    {      super.changeName( newName + " " + LASTNAME );    }  } |

1. Yes. Since AObject is declared to be abstract, it does not have to provide non-abstract definitions for the methods of the Adder interface it is implementing.
2. No. The Thing class is not declared to be abstract, and yet it does not implement the methodA method of the Upper interface, which is the direct superinterface of the interface (Lower) it claims to be implementing.
3. Yes. The ThingB class is declared to be abstract. So it does not have to implement all the methods of the Lower interface that it implements or those of the Upper interface, which is Lower's direct superinterface. In fact, the methodA method is not implemented in ThingB. The ThingA class is a concrete class that extends ThingB and provides a non-abstract definition for the methodA method, thereby completing the implementation of Lower.
4. Yes. The concrete ThingA class provides a non-abstract definition for the only method of the Upper interface it claims to implement. The concrete ThingB class extends ThingA and therefore inherits ThingA's implementation of methodA. So by providing a non-abstract definition of the methodB method from the Lower interface, ThingB completes the implementation of Lower.
5. No. The Alias interface includes an instance variable, myName. This is not allowed. The only variables that may be declared in an interface definition are public, static, final class constants that are initialized.
6. Yes. The Alias interface includes a class constant, LASTNAME, that is initialized to the String "Johnson". The ProtectedWitness class implements the changeName method of the Alias interface (and, incidentally, at the same time overrides the changeName method of the Witness class by calling on that superclass's method).

As we mentioned in passing on the previous page, a class may implement more than one interface. If it does, then the interfaces that are implemented are listed, separated by commas, in the class header after the implements keyword, as in this example:

public interface Stampable   
{   
  void stamp();   
  void flatten();   
}   
  
public interface Collectible   
{   
  void pickMeUp( ArrayList a );   
}   
    
public class BrassAlloy implements Stampable, Collectible   
{   
  public BrassAlloy()   
  {   
  }   
  
  public void stamp()   
  {   
    System.out.println( "kerplunk!" );   
  }   
  
  public void flatten()   
  {   
    System.out.println( "plink!" );   
  }   
  
  public void pickMeUp( ArrayList a )   
  {   
    a.add( this );   
  }   
}

An object may be cast to any of the types named by the interfaces it implements, or to any of their superinterfaces (if they have any), or to any of its superclasses, all the way up to Object. In the above example, since the BrassAlloy class does not specify an explicit direct superclass, an object of type BrassAlloy may be cast to Stampable, Collectible, and Object:

BrassAlloy b = new BrassAlloy();   
  
Stampable s = (Stampable)b;   
Collectible c = (Collectible)b;   
Object o = (Object)b;

**Exercise 158**

Study the following code:

public interface Identifiable   
{   
  String whoAmI();   
}   
  
public interface Traceable extends Identifiable   
{   
  String whereAmI();   
}   
  
public interface Punishable   
{   
  void communityService( int days );   
  void probation( int days );   
  void incarceration( int days );   
}   
  
public abstract class Person   
{   
  public Person()   
  {   
  }   
  
  public abstract int getAge();   
}   
  
public class ExCon extends Person implements Identifiable   
{   
  // <code not shown>   
}   
  
public class Felon extends Person implements Traceable, Punishable   
{   
  // <code not shown>   
}

1. What methods must the ExCon class implement if it is to be a concrete class?
2. What methods must the Felon class implement if it is be a concrete class?
3. To what types may an ExCon object be cast?
4. To what types may a Felon object be cast?
5. whoAmI and getAge.
6. whoAmI, whereAmI, communityService, probation, incarceration, and getAge.
7. Identifiable, Person, and Object.
8. Identifiable, Traceable, Punishable, Person, and Object.