Java: steps towards good quality code

Barry Cornelius
Computing Services, University of Oxford
Date: 14th May 2001
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

Good practice is best acquired from the start. Beginners find it difficult to learn one thing one day only to be told later that it is best done in some other way. This document describes some of the ideas I teach when introducing interfaces and classes to students. My aim is to get beginners to produce good quality code for interfaces and classes from the outset.

2 Providing a common *minimal public interface*

Often a class declaration is used to produce a new type. When teaching interfaces and classes, I stress that it is important that such classes have:

- methods called `equals`, `hashCode`, `toString`, and (if appropriate) a method called `compareTo`;
- a constructor that initializes the object from a parameter of type `String`;
- get methods;
- set methods (if appropriate);
- some means for cloning an object.

Some of these (e.g., `equals`, `hashCode`, `compareTo`) are needed if objects of the class are to be stored in a collection (e.g., a list). When we produce a class, we may not know what *clients* (users of our class) will want to do with objects of the class. I do not believe we should change the class later or produce a subclass later. Instead, we should provide the required features at the outset.

In some ways, a class is poorly constructed unless it has all of these methods and constructors.

In a book entitled 'Object-Oriented Design Heuristics' ([13]), Arthur Riel introduces the idea of the *minimal public interface*. He says: 'If the classes that a developer designs and implements are to be reused by other developers in other applications, it is often useful to provide a common minimal public interface. This minimal public interface consists of functionality that can be reasonably expected from each and every class.'

Bill Venners has also written an article on the topic of minimal public interfaces. The article is entitled 'The canonical object idiom', and it is available from the WWW at ([22]).

3 Providing `equals`

3.1 The need to provide `equals` with an `Object` parameter

I find it incredible that so many books on Java introduce classes as a means of representing real-world objects and yet do not provide each class with the ability to find out whether two objects of the class have the same value. In Java, this is normally done by providing a method called `equals`. This method is called in a way illustrated by:

```
final Date tTodaysDate = new Date(2000, 12, 25);
final Date tNoelDate = new Date(2000, 12, 25);
final boolean tIsChristmasDay = tTodaysDate.equals(tNoelDate);
```

Suppose a book introduces a class called `Date` for representing objects that are dates. Then, if the book does mention `equals`, often you will find it declared with a parameter of type `Date`:

```
public boolean equals(final Date pDate)
{
   return iYear==pDate.iYear && iMonth==pDate.iMonth && iDay==pDate.iDay;
}
```

If a client attempts to add objects of this class to an object of the Collections API (or `Hashtable` or `Vector`), they are in for a shock. None of the following methods will work with the above declaration of `equals`:

`Hashtable`	`contains`, `containsKey`, `get`, `put`, `remove`
`Vector`	`contains`, `indexOf`
`List`	`contains`, `indexOf`, `remove`
`Map`	`containsKey`, `containsValue`, `get`, `put`, `remove`
`Set`	`add`, `contains`, `remove`

For these methods to work, you need to provide an `equals` method that has a parameter of type `Object`. Currently, you will find very few books on Java that teach this.

3.2 Using `getClass` in the code of `equals`

For example, if we are providing a class called `Date`, we could declare the following:

```
public boolean equals(final Object pObject)
{
   if ( pObject==null || getClass()!=pObject.getClass() )
   {
      return false;
   }
   final Date tDate = (Date)pObject;
   return iYear==tDate.iYear && iMonth==tDate.iMonth && iDay==tDate.iDay;
}
```

This code uses a method called `getClass` (which is declared in the class `java.lang.Object`). This is a method that returns a value of type `java.lang.Class`, a value that describes the class of its target. In the above code for `equals`, the `getClass` method is called twice:

```
pObject==null || getClass()!=pObject.getClass()
```

In the second call, the target of the call of `getClass` is the object to which `pObject` is pointing. So the value that is returned is the class of this object.

In the first call, there is no explicit target. When `getClass` is called without a target (from the `equals` method), it will be applied to whatever object is the target of the call of `equals`. So the first call of `getClass` is finding out the class of the object to which `equals` is being applied.

Since the `equals` method appears in a class called `Date`, you would think that the target of `equals` must be an object of class `Date`, and so this kind of call of `getClass` always returns a value describing the class `Date`. However, suppose we derive a class called `NamedDate` from `Date`:

```
public class NamedDate extends Date
{
   private String iName:
   ...
}
```

Suppose that `NamedDate` does not override `equals`. If we write:

```
tFirstNamedDate.equals(tSecondNamedDate)
```

then this will be a call of `Date`'s `equals` method and both calls of `getClass` will return values describing the class `NamedDate`. So, even though this code appears in the class declaration for `Date`, in some circumstances the first call of `getClass` will return a value describing a subclass of the class `Date`.

3.3 Using `getClass` in preference to `instanceof`

When providing a proper version of `equals`, many authors (e.g., [4]) use code for `equals` that has the following form:

```
public boolean equals(final Object pObject)
{
   if ( ! (pObject instanceof Date) )
   {
      return false;
   }
   final Date tDate = (Date)pObject;
   return iYear==tDate.iYear && iMonth==tDate.iMonth && iDay==tDate.iDay;
}
```

Here the `instanceof` operator is being used instead of calling `getClass` twice.

Although the code using the `instanceof` operator looks easier to understand, it is inappropriate to use this code because it causes problems if you or someone else later uses inheritance to subclass this class and you want the subclass to override the version of `equals` provided by `Date`.

The problem is that this version of `equals` does not always satisfy one of the rules of the contract of `equals` mentioned in the WWW pages that document the class `Object` ([19]). This rule says that the `equals` method 'is *symmetric*: for any reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true'.

Suppose we were to provide `Date` with the version of `equals` given above, and provided `NamedDate` with the following version of `equals`:

```
public boolean equals(final Object pObject)
{
   if ( ! (pObject instanceof NamedDate) )
   {
      return false;
   }
   return super.equals(pObject) && iName.equals(((NamedDate)pObject).iName);
}
```

Then we would have a problem when we use `equals` with two objects, one being of class `Date` and the other being of class `NamedDate`. This can be illustrated by executing the code:

```
Date tDate = new Date(2001, 5, 16);
NamedDate tNamedDate = new NamedDate(2001, 5, 16, "Hull");
System.out.println(tDate.equals(tNamedDate));
System.out.println(tNamedDate.equals(tDate));
```

The first call of `equals` is being applied to a `Date` object and so `Date`'s `equals` method will be used. This call will lead to `pObject` pointing to an object of class `NamedDate`. So what happens with the test `pObject instanceof Date`? Is this going to deliver `true` or `false` when `pObject` is pointing to a `NamedDate` object? Well, for a condition that has the form: '*RelationalExpression* `instanceof` *ReferenceType*' the definition of Java (given in 'Java Language Specification' ([10])) says: 'At run time, the result of the `instanceof` operator is `true` if the value of the *RelationalExpression* is not `null` and the reference could be cast to the *ReferenceType* without raising a `ClassCastException`'. Since the cast `(Date)pObject` is allowed when `pObject` is pointing to a `NamedDate` object, then `pObject instanceof Date` is allowed and has the value `true`. So the code moves on and produces the value `true` or `false` depending on whether the `iYear`, `iMonth` and `iDay` values of the two objects are the same. In the above example, they are the same, and so the value `true` will be produced.

The second call of `equals` uses `NamedDate`'s `equals` method. It will always produce the value `false`. This is because `tDate` does not satisfy the test that checks whether the parameter is an `instanceof NamedDate`.

So, when an `equals` method is written using `instanceof`, it does not always satisfy the symmetric rule. For more information about this, see:

- pages 44 to 59 of the book 'Practical Java' ([11]) by Peter Haggar;
- the WWW page at [14] containing an article by Mark Roulo entitled 'How to avoid traps and correctly override methods from `java.lang.Object`'.

Both of these authors point out that `instanceof` (rather than `getClass`) is used in the code of the `equals` methods of some of the classes of Java's Core APIs. And so you may run into the non-symmetric problem if you want to produce a subclass of one of these classes. In his book, Peter Haggar says: 'A quick glance through the source code of the Java libraries shows the use of `instanceof` in `equals` method implementations is common. You also find the use of `getClass`. The Java libraries are not consistent in how they implement the `equals` methods of their classes, thereby making consistent equality comparisons difficult'.

4 Providing hashCode

4.1 The need to provide hashCode

If you declare `equals` properly, you need also to declare `hashCode`. There are warnings about this in the documentation of some of the classes. For example, the WWW pages that document `java.util.Hashtable` ([21]) state that 'to successfully store and retrieve objects from a hashtable, the objects used as keys must implement the `hashCode` method and the `equals` method'. By this, it means that a class should override the methods called `hashCode` and `equals` that are declared in `java.lang.Object`. So you should declare methods with the following headers:

```
public int hashCode();
public boolean equals(Object pObject);
```

The `hashCode` method will get used behind the scenes by the following methods:

`Hashtable`	`contains`, `containsKey`, `get`, `put`, `remove`
`HashMap`	`containsKey`, `containsValue`, `get`, `put`, `remove`
`HashSet`	`add`, `contains`, `remove`

Few books that teach Java mention `hashCode`.

4.2 The contract that hashCode should satisfy

The contract that `hashCode` needs to satisfy is given in the WWW pages that document the class `Object` ([20]). They say that the general contract of `hashCode` is:

1. 'Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.'

2. 'If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.'

3. 'It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.'

4.3 Providing a hashCode method that returns the value 0

The full implications of these three rules are not particularly easy to understand. So, when teaching `equals` and `hashCode`, I find it easier, to begin with, to advise students to provide classes (for new types) with the following `hashCode` function:

```
public int hashCode()
{
   return 0;
}
```

When this function is called, it always returns the same value, and so this `hashCode` function satisfies the first two rules. What the third rule is saying is that we may get poor execution speeds by choosing this `hashCode` function. However, there are some subtle points that need to be considered if you wish to provide a more sophisticated form of `hashCode`.

4.4 Providing other code for the hashCode function

For the `Date` example, another possibility for the `hashCode` function is:

```
public int hashCode()
{
   return iMonth;
}
```

If we use this `hashCode` function, an integer in the range 1 to 12 is associated with each of the values of the class `Date`. For example:

```
final Date tNoelDate = new Date(2000, 12, 25);
final int tValue = tNoelDate.hashCode();
```

will assign the value 12 to `tValue`.

So, if we were to store values of the type `Date` in a collection (such as a `HashSet`), the designer of the collection class could arrange for the values of the collection to be stored in several *buckets*: all the values of the collection that have a hashcode of 1 would be stored in the first bucket; all those with a hashcode of 2 would be stored in the second bucket; and so on. So, for this `hashCode` function, 12 buckets would be used. When we later call the `contains` method to check whether a particular `Date` value is in the collection, the code of the `contains` method can find the hashcode of the `Date` value and then it need only look at the values in the appropriate bucket.

For example, suppose we want to store the dates when various composers died:

Bach	1750-08-28
Beethoven	1827-03-26
Cage	1992-08-12
Chopin	1849-10-17
Copland	1990-12-02
Elgar	1934-02-23
Handel	1759-04-14
Mendelssohn	1847-11-04
Purcell	1695-11-21
Sibelius	1957-09-20
Stanford	1924-03-29
Tallis	1585-11-23
Tchaikovsky	1893-11-06
Vaughan-Williams	1958-08-26
Walton	1983-03-08

Suppose we add each of these dates to a `HashSet`, e.g. for Bach:

```
Date tDeathOfBach = new Date(1750, 8, 28);
tHashSet.add(tDeathOfBach);
```

`HashSet`'s `add` method uses `Date`'s `hashCode` function, and so the values will be stored in 12 buckets:

1.
2. `1934-02-23`
3. `1924-03-29`, `1827-03-26`, `1983-03-08`
4. `1759-04-14`
5.
6.
7.
8. `1750-08-28`, `1992-08-12`, `1958-08-26`
9. `1957-09-20`
10. `1849-10-17`
11. `1847-11-04`, `1695-11-21`, `1893-11-06`, `1585-11-23`
12. `1990-12-02`

Then, when later we ask the collection object whether it has the value `1893-11-06` (the date when Tchaikovsky died), the `contains` method can call `hashCode` on this value and, because this produces the value 11, the `contains` method only checks the values in the 11th bucket. The code of the `contains` method uses `equals` on each of these values in turn returning the value `true` if and only if it finds the value (in this case, the value `1893-11-06`).

Besides the above coding of the `hashCode` function, there are many other possibilities we could choose instead. Here is another example:

```
public int hashCode()
{
   return iYear*10000 + iMonth*100 + iDay;
}
```

If we were to use this function, each value of the class `Date` would have its own unique hashcode.

4.5 The reason for using a zero-returning `hashCode`

However, there is one problem which we have not yet considered. If a client chooses to change a value after it has been put in a collection, the value will no longer be in the right bucket. So it will not be found if we later search for it.

For example, contrary to what it says above, Bach actually died on 28th July 1750 (rather than on 28th August 1750). So we might want to change this date:

```
tDeathOfBach.setMonth(7);
```

This would change the collection to:

1.
2. `1934-02-23`
3. `1924-03-29`, `1827-03-26`, `1983-03-08`
4. `1759-04-14`
5.
6.
7.
8. `1750-07-28`, `1992-08-12`, `1958-08-26`
9. `1957-09-20`
10. `1849-10-17`
11. `1847-11-04`, `1695-11-21`, `1893-11-06`, `1585-11-23`
12. `1990-12-02`

Suppose we now use `contains` to search for the value `1750-07-28`. Because, when `hashCode` is applied to this value it produces the value 7, the `contains` method will look in the 7th bucket, which is empty. So the method will not find the value as the appropriate value is in the wrong bucket.

Rule 1: Here is an important rule: a `hashCode` function should not be written in terms of fields that can be changed.

So, if the class `Date` provides `setYear`, `setMonth` and `setDay`, we should not provide a `hashCode` function that is written in terms of the `iYear`, `iMonth` and/or `iDay` fields. This is the reason why we might want to use:

```
public int hashCode()
{
   return 0;
}
```

If we use this `hashCode` function, a collection class will use one bucket for all the objects we put into the collection. Although this means that a method like `contains` will execute more slowly as all the values of the collection are in one bucket, it does mean that we need not worry about values being changed after they have been added to the collection.

4.6 It is not a problem for immutable classes

Of course, this problem will not occur if you are providing an *immutable class*, a class where the fields of each object of the class cannot be changed once an object has been created. In such circumstances, you will be able to choose a `hashCode` function that helps to speed up searching.

So if we removed `setYear`, `setMonth` and `setDay` from the `Date` class, its objects would now be immutable. We could then use either of the two `hashCode` functions that were given above. This would speed up the searching for dates when they have been stored in a `Hashtable`, a `HashSet` or a `HashMap`.

4.7 Another rule

Rule 2: Here is another important rule: the same hashcode values must be produced for any two objects that are equal (according to the `equals` method).

In practice, this means that a `hashCode` function must always return the same value (e.g., the value 0) or it must be written in terms of some or all of the fields used in the declaration of `equals` (and no other fields).

4.8 What is the effect on performance?

I measured the effect on performance by timing the execution of a program. The program creates a `HashSet` that contains 10000 elements that are objects of the `Date` class. The program times the execution of 10000 calls of `contains`. The following results were obtained for different codings of the `hashCode` method of the `Date` class:

the code of the `hashCode` *method*	*time taken*
`return 0;`	14826
`return iMonth;`	1235
`return iYear*10000 + iMonth*100 + iDay;`	36

The times are given in milliseconds. These results demonstrate how a carefully chosen `hashCode` method can affect the performance of some programs.

5 Providing `compareTo`

5.1 The need to provide `compareTo`

If the class that you are providing is for a type where there is a natural order for the values of the type, the class should also provide a means for finding out whether one value of the type is less than another value. Unfortunately, few books on Java teach this.

There are some parts of the Collections API (e.g., the classes `TreeSet` and `TreeMap`) that work better if your class implements the `Comparable` interface ([17]) from the package `java.lang`. This interface is simply:

```
public interface Comparable
{
   public int compareTo(Object pObject);
}
```

For `Date` to implement `Comparable`, we need to change it to something like the following:

```
public class Date implements java.lang.Comparable
{
   ...
   public int compareTo(final Object pObject)
   {
      final Date tDate = (Date)pObject;
      int tResult = iYear - tDate.iYear;
      if (tResult==0)
      {
         tResult = iMonth - tDate.iMonth;
         if (tResult==0)
         {
            tResult = iDay - tDate.iDay;
         }
      }
      return tResult;
   }
}
```

The `Comparable` interface became part of Java when the Java 2 platform was released in December 1998.

If a class implements the `Comparable` interface, objects of this class can be stored in a `TreeSet` or a `TreeMap`. For example:

```
final Set tOccurrences = new TreeSet():
tOccurrences.add(tDeathOfBach);
```

5.2 Using the `Comparator` interface

If a class such as `Date` fails to implement the `Comparable` interface, or its implementation of `compareTo` provides inappropriate code, a client class can still store `Date` objects in a `TreeSet` or a `TreeMap` provided it creates the `TreeSet`/`TreeMap` using a constructor that is passed an object that implements the `Comparator` interface ([18]). This interface requires the object to provide the method:

```
public int compare(Object pObject1, Object pObject2);
```

This method returns a negative integer, zero, or a positive integer depending on whether the value of `pObject1` is less than, equal to, or greater than that of `pObject2`.

Here is an example. Suppose, bizarrely, we chose to compare dates only on the year field. We could provide:

```
public class MyDateComparator implements java.util.Comparator
{
   public int compare(final Object pObject1, final Object pObject2)
   {
      return ((Date)pObject1).getYear() - ((Date)pObject2).getYear();
   }
}
```

and then use:

```
final MyDateComparator tMyDateComparator = new MyDateComparator();
final Set tOccurrences = new TreeSet(tMyDateComparator):
tOccurrences.add(tDeathOfBach);
```

6 Providing a clone

6.1 The need to provide a cloning operation

Few books teaching Java explain that, when producing a class, it is desirable to provide a cloning operation. For example, when creating a `Person` object, we might let a client supply a `Date` object that is the person's date-of-birth:

```
Date tBirthDate = new Date(2000, 1, 24);
Person tSomePerson = new Person("Joe", tBirthDate);
```

where `Person` is as follows:

```
public class Person
{
   private String iName;
   private Date iDateOfBirth;
   public Person(final String pName, final Date pDateOfBirth)
   {
      iName = pName;
      iDateOfBirth = pDateOfBirth;                // share
      ...
```

If we do this, the `Person` object is sharing the `Date` object supplied by the client. If the `Date` class provides mutable objects, this may be undesirable.

Instead of sharing the `Date` object with the client, the `Person` object may prefer to have its own copy. The classes of Java's Core APIs use two different ways of producing a copy of an object:

- a class sometimes provides a method called `clone` that overrides `java.lang.Object`'s `clone`;
- a class sometimes provides a suitable constructor.

So, if `Date` provided `clone`, the `Person` constructor could use:

```
      iDateOfBirth = (Date)pDateOfBirth.clone();  // clone
```

Or, if `Date` provided a cloning constructor, the `Person` constructor could use:

```
      iDateOfBirth = new Date(pDateOfBirth);      // clone
```

It is best to provide a method called `clone` as this can be used when inheritance is involved. However, getting the code of a `clone` method completely right is difficult.

6.2 Providing a constructor for cloning

Because a `clone` method is difficult to get right, it is also difficult to teach. So, to begin with, I cheat by teaching students to provide a constructor that can be used for cloning:

```
public Date(final Date pDate)
{
   iYear = pDate.iYear;
   iMonth = pDate.iMonth;
   iDay = pDate.iDay;
}
```

6.3 Providing a method called `clone`

The class `java.lang.Object` provides a method called `clone`. The header of this method is:

```
public Object clone();
```

When it is used on an object, it returns a new instance of the object which contains a copy of all the fields of the object. If you want a class to support cloning, it is best to override this method.

To do this, your class needs to say that it implements the `Cloneable` interface and its `clone` method must catch the `CloneNotSupportedException` exception. Both of these need only be done if your class is a direct subclass of `Object`. For example, for the `Date` class, we may want to provide:

```
public class Date implements Cloneable
{
   ...
   public Object clone()
   {
      try
      {
         return super.clone();
      }
      catch(final CloneNotSupportedException pCloneNotSupportedException)
      {
         throw new InternalError();
      }
   }
}
```

`Object`'s `clone` method only produces a *shallow copy*. So, if a class has one or more fields that are of a reference type, we may want to provide a *deep copy* by cloning these fields. For example, for the `Person` class, we could provide:

```
public class Person implements Cloneable
{
   ...
   public Object clone()
   {
      try
      {
         final Person tPerson = (Person)super.clone();
         if (iDateOfBirth!=null)
         {
            tPerson.iDateOfBirth = (Date)iDateOfBirth.clone();
         }
         return tPerson;
      }
      catch(final CloneNotSupportedException pCloneNotSupportedException)
      {
         throw new InternalError();
      }
   }
}
```

6.4 Other information about cloning

For more information about how to clone objects, look at the 'The canonical object idiom' article by Bill Venners ([22]) or look at Section 19.8 of my book 'Understanding Java' ([5]).

7 An introduction to interfaces

7.1 Use of interfaces for GUI callbacks

Perhaps a Java programmer's first use of an interface occurs with handling the events of a GUI. For example, suppose a window (`JFrame`) has a button (`JButton`):

```
final JFrame tJFrame = new JFrame("some title");
final JButton tJButton = new JButton("click here");
final Container tContentPane = tJFrame.getContentPane();
tContentPane.add(tJButton, BorderLayout.CENTER);
tJFrame.pack():
tJFrame.setVisible(true);
```

If the program has to react to a click of the button, the program has to create an object and register it as a listener:

```
final JButtonListener tJButtonListener = new JButtonListener();
tJButton.addActionListener(tJButtonListener);
```

Here we have created an object of a class called `JButtonListener` and passed `tJButtonListener` as the argument of the `addActionListener` method. The documentation of `addActionListener` states that its parameter is of the interface type `ActionListener`. So `JButtonListener` must be a class that implements the `ActionListener` interface:

```
public class JButtonListener implements ActionListener
{
   ...
}
```

Looking again at the documentation of the `ActionListener` interface, you will see that this interface is simply:

```
public interface ActionListener
{
   public void actionPerformed(ActionEvent pActionEvent);
}
```

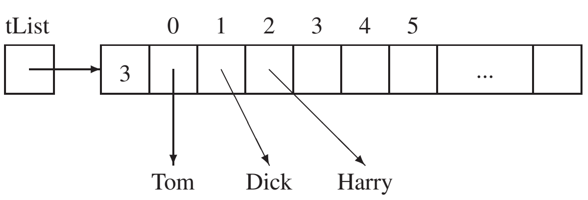
So this means that `JButtonListener` could be something like:

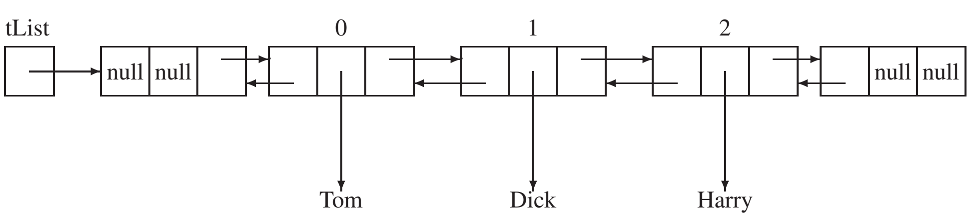
```
public class JButtonListener implements ActionListener
{
   public void actionPerformed(final ActionEvent pActionEvent)
   {
      final Date tDate = new Date();
      System.out.println(tDate);
   }
}
```

7.2 Use of interfaces with collection classes

Java's Collections API provides classes that can be used for representing collections of objects, such as lists, sets and maps. It provides two classes for each of these. For lists, it provides classes called `ArrayList` and `LinkedList`; for sets, it provides classes called `HashSet` and `TreeSet`; and, for maps, it provides classes called `HashMap` and `TreeMap`. Often your choice of the class will be determined by the particular operations that you want to perform: one of the classes performs faster than the other.

Suppose we want to represent a list of objects. If we are using the Collections API, we have a choice between using an `ArrayList` and a `LinkedList`:





The `ArrayList` class performs well in most situations. However, the `LinkedList` class performs better than `ArrayList` when the operations that dominate involve adding or removing items at the ends of the list.

Having made a choice over the class, then the operations that you can perform on the list are the same. So, if `tList` points to an `ArrayList` object or a `LinkedList` object, we can perform operations such as:

```
tList.add("Tom");
tList.add("Harry");
tList.add(1, "Dick");
tList.remove(1);
tList.set(1, "Dick");
String tString = (String)tList.get(1);
```

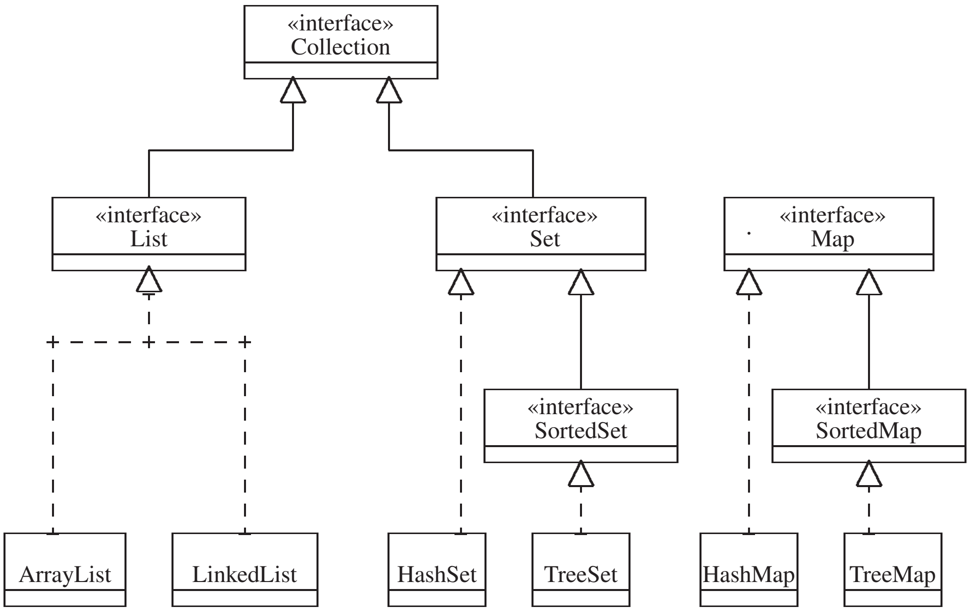
Before executing this code, we need to create the list object. The code we use for this depends on whether we want an `ArrayList` or a `LinkedList`. So we can choose between:

```
ArrayList tList = new ArrayList();
```

or:

```
LinkedList tList = new LinkedList();
```

However, the above is not regarded as good programming style. Sun's documentation for the Collections API encourages you to code in terms of interfaces called `List`, `Set` and `Map` instead of writing code in terms of specific classes (such as `ArrayList` and `LinkedList`).



So use either:

```
List tList = new ArrayList();
```

or:

```
List tList = new LinkedList();
```

Instead of using a variable `tList` which is of a specific class type (`ArrayList` or `LinkedList`), we are now using a variable `tList` which is of the interface type `List`. A variable that is of an interface type may point to any object that is of a class that implements that interface.

If a list object has to be passed as an argument to a method, then this style of programming recommends that the parameter be of the interface type rather than of the specific class type. So, instead of the method having a header like:

```
public void processList(ArrayList pArrayList);
```

it should be:

```
public void processList(List pList);
```

With this style of programming, most of the code is written in terms of the `List` interface. The only place where either `ArrayList` or `LinkedList` has to be mentioned is when we create the list object, i.e., the commitment to use an `ArrayList` instead of a `LinkedList` (or vice-versa) only appears where the object is created.

7.3 Providing your own interfaces

So far, the examples have involved interfaces that others have provided. So, when might we want to provide our own interface declarations?

We may want to do this if we have many classes that provide the same set of operations and we want to execute code that processes various objects of these different classes.

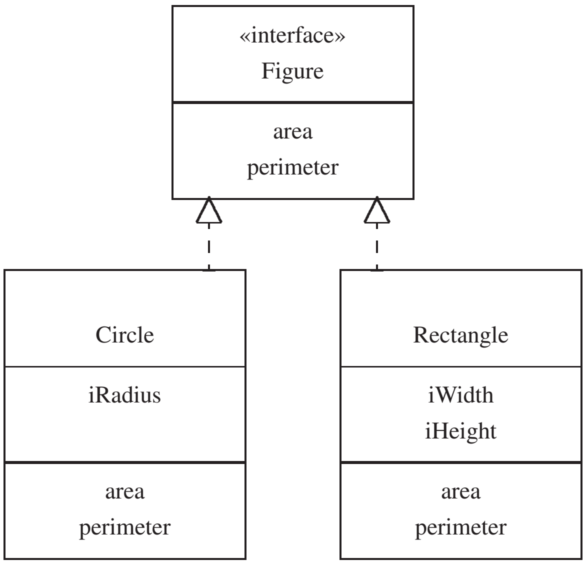
Suppose we are processing geometrical figures, some of which are circles and others are rectangles. Suppose we want to find out the area or the perimeter of these figures. We could provide the interface:

```
public interface Figure
{
   public double area();
   public double perimeter();
}
```

and provide classes called `Circle` and `Rectangle`. Here is some code for `Circle`:

```
public class Circle implements Figure
{
   private double iRadius;
   public Circle(final double pRadius)
   {
      iRadius = pRadius;
   }
   public double area()
   {
      return Math.PI*iRadius*iRadius;
   }
   public double perimeter()
   {
      return 2.0*Math.PI*iRadius;
   }
   ...
}
```

The code for `Rectangle` is similar.



Suppose we store several objects of these two classes in a list:

```
final List tList = new ArrayList();
tList.add( new Circle(100.0) );
tList.add( new Rectangle(200.0, 300.0) );
...
```

We can then process the elements of the list using code like:

```
final Iterator tIterator = tList.iterator();
while ( tIterator.hasNext() )
{
   final Figure tFigure = (Figure)tIterator.next();
   System.out.println( tFigure.area() );
}
```

By coding the classes as implementations of an interface we can perform operations on objects that are of different classes.

8 Looking at other programming languages

When I came to examine Java for the first time, it was from a background in many programming languages including Modula-2, Ada 83 and C++.

8.1 Modula-2

In Modula-2, ‘a separate module is split into two parts: a definition part and an implementation part. The definition module gives information as to *what* services are provided; and the implementation module contains the full code of the module: it specifies *how* the services are to be provided. These two parts of a separate module are stored in two different files’ ([3]). By using an *opaque type*, clients can be written that are not dependent on the code of the implementation module.

8.2 Ada 83

In a similar way, in Ada 83, a package can be split into two parts: a *package specification* and a *package body*. A *limited private type* can be used in much the same way as an opaque type can be used in Modula-2.

8.3 C++

Both Modula-2 and Ada 83 provide different constructs for the interface and the implementation. This is not so with C++ where, if you wish to use this style of programming, the class construct is used for both.

In his fascinating book 'The Design and Evolution of C++' ([16]), Stroustrup (the designer of C++) says: 'I ... made matters worse for the C++ community by not properly explaining the use of derived classes to achieve the separation of interface and implementation. I tried (see for example Section 7.6.2 of [15]), but somehow I never got the message across. I think the reason for this failure was primarily that it never occurred to me that many (most?) C++ programmers and non C++ programmers looking at C++ thought that because you *could* put the representation right in the class declaration that specified the interface, you *had* to.'

To make it easier to express this separation, Stroustrup added the concept of an *abstract class* to C++. 'The very last feature added to Release 2.0 before it shipped [in June 1989] was abstract classes. Late modifications to releases are never popular, and late changes to the definition of what will be shipped are even less so. My impression was that several members of management thought I had lost touch with the real world when I insisted on this feature.'

'An abstract class represents an interface. Direct support for abstract classes

- helps catch errors that arise from confusion of classes' role as interfaces and their role in representing objects;
- supports a style of design based on separating the specification of interfaces and implementations.

'

In C++, an abstract class is a class that has one or more *pure virtual functions*:

```
class figure
{
   public:
      virtual double area() = 0;
      virtual double perimeter() = 0;
};
```

9 Providing an interface to describe a type

In Modula-2 and Ada 83, it is easy to separate the interface from the implementation. So having this background, and having taught Modula-2 in this way for many years ([3]), when I started to look at Java I wondered whether the same style of programming could be achieved (and should be adopted) in Java.

If we need to represent some objects in a Java program, at the same time as producing a class declaration that describes the implementation, we could also produce an interface declaration that describes the operations provided by the methods of the class.

An interface declaration for dates could be:

```
public interface Date
{
   public int getYear();
   public int getMonth();
   public int getDay();
   public void setYear(int pYear);
   public void setMonth(int pMonth);
   public void setDay(int pDay);
   public boolean equals(Object pObject);
   public int hashCode();
   public String toString();
}
```

And a class declaration for dates could be:

```
import java.util.StringTokenizer;
public class DateImpl implements Date
{
   private int iYear;
   private int iMonth;
   private int iDay;
   public DateImpl()
   {
      this(1970, 1, 1);
   }
   public DateImpl(final Date pDate)
   {
      final DateImpl tDateImpl = (DateImpl)pDate;
      iYear = tDateImpl.iYear;
      iMonth = tDateImpl.iMonth;
      iDay = tDateImpl.iDay;
```

```
   }
   public DateImpl(final int pYear, final int pMonth, final int pDay)
   {
      iYear = pYear;
      iMonth = pMonth;
      iDay = pDay;
   }
   public DateImpl(final String pDateString)
   {
      try
      {
         final StringTokenizer tTokens =
                       new StringTokenizer(pDateString, "-");
         final String tYearString = tTokens.nextToken();
         iYear = Integer.parseInt(tYearString);
         final String tMonthString = tTokens.nextToken();
         iMonth = Integer.parseInt(tMonthString);
         final String tDayString = tTokens.nextToken();
         iDay = Integer.parseInt(tDayString);
      }
      catch(final Exception pException)
      {
         iYear = 1970;
         iMonth = 1;
         iDay = 1;
         throw new IllegalArgumentException();
      }
   }
   public int getYear()                    { return   iYear;  }
   public int getMonth()                   { return   iMonth; }
   public int getDay()                     { return   iDay;   }
   public void setYear(final int pYear)    { iYear  = pYear;  }
   public void setMonth(final int pMonth)  { iMonth = pMonth; }
   public void setDay(final int pDay)      { iDay   = pDay;   }
   public boolean equals(final Object pObject)
   {
      if ( pObject==null || getClass()!=pObject.getClass() )
      {
         return false;
      }
      final DateImpl tDateImpl = (DateImpl)pObject;
      return iYear==tDateImpl.iYear &&
             iMonth==tDateImpl.iMonth && iDay==tDateImpl.iDay;
   }
   public int hashCode()
   {
      return 0;
   }
   public String toString()
   {
      return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
   }
}
```

Whenever possible, the code of a client should use the interface (rather than the class). So:

- reference variables should be declared to be of the interface type `Date` rather than of the class type `DateImpl`:

  ```
  Date tDate;
  ```

- parameters of any methods (of the client) should be declared to be of the interface type `Date` rather than of the class type `DateImpl`:

  ```
  public boolean iIsLeap(final Date pDate);
  ```

The only time we have to use the class type (`DateImpl`) is when we want to create an object:

```
Date tDate = new DateImpl(2000, 12, 25);
```

10 Pros and cons of providing an interface

Introducing an interface as well as a class declaration is preferable because:

1. an interface declaration provides a clearer statement than a class declaration as to the contract (between the client and the supplier);
2. writing the client mainly in terms of the interface means that it will be easier for the client to switch to a different implementation of a class;

3. writing the client so that it only uses the interface and not the class implementing the interface means that it is not necessary to recompile the client when the class changes.

Cymerman ([6]) says: 'The only drawbacks to this scheme are the fact that there is some overhead associated with the creation and casting of objects, and the fact that more code is required to first specify the interface and then code the class that implements the interface. These two drawbacks seem insignificant, though, compared to the vast number of benefits.'

The book 'The Java Programming Language' by Arnold, Gosling and Holmes ([1]) says: 'Any major class you expect to be extended, whether abstract or not, should be an implementation of an interface. Although this approach requires a little more work on your part, it enables a whole category of use that is otherwise precluded.' In their book, they produce a class with the header:

```
class AttributedBody extends Body implements Attributed
```

They say: 'suppose we had created an `Attributed` class instead of an `Attributed` interface with an `AttributedImpl` implementation class. In that case, programmers who wanted to create new classes that extended other existing classes could never use `Attributed`, since you can extend only one class: the class `AttributedBody` could never have been created.'

In the book 'UML Distilled' ([7]), Martin Fowler writes: 'Programming languages [other than Java] use a single construct, the class, which contains both interface and implementation. When you subclass, you inherit both. Using the interface as a separate construct is rarely used, which is a shame.'

In the 1990s, some useful work was done on understanding how complex systems are built: this work recognized the importance of the use of *patterns*. Currently, the principal book in this area is 'Design Patterns: Elements of Reusable Object-Oriented Software' by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides ([9]). These four people are affectionately known as the *Gang of Four* (or the *GoF*). In their book, they say: 'This ... leads to the following principle of reusable object-oriented design:

- *Program to an interface, not an implementation.*

Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class [or an interface in Java]. You will find this to be a common theme of the design patterns in this book.'

On the WWW at [8], you will find an excellent document entitled 'ChiMu OO and Java Development: Guidelines and Resources'. Mark Fussell says: 'Use interfaces as the glue throughout your code instead of classes: define interfaces to describe the exterior of objects (i.e., their Type) and type all variables, parameters, and return values to interfaces. The most important reason to do this is that interfaces focus on the client's needs: interfaces define what functionality a client will receive from an Object without coupling the client to the Object's implementation. This is one of the core concepts to OO.'

Stroustrup ([16]) says: 'The importance of the abstract class concept is that it allows a cleaner separation between a user and an implementer than is possible without it. An abstract class is purely an interface to the implementations supplied as classes derived from it. This limits the amount of recompilation necessary after a change as well as the amount of information necessary to compile an average piece of code. By decreasing the coupling between a user and an implementer, abstract classes provide an answer to people complaining about long compile times and also serve library providers, who must worry about the impact on users of changes to a library implementation. I have seen large systems in which the compile times were reduced by a factor of ten by introducing abstract classes into the major subsystem interfaces.'

11 The factory pattern

Although this programming style encourages the client to use the interface rather than the class, the client is still using the class if it has to create an instance of the class:

```
Date tDate = new DateImpl(2000, 12, 25);
```

Suppose we now need to represent people and the constructor for the class `PersonImpl` needs to create a `DateImpl` object for a person's date of birth:

```
public PersonImpl(String pName, int pYear, int pMonth, int pDay)
{
   iName = pName;
   iDateOfBirth = new DateImpl(pYear, pMonth, pDay);
}
```

Including this code in `PersonImpl` means that it is dependent on `DateImpl`. If we make a change to the `DateImpl` class, we will need to recompile (and re-test) `PersonImpl` (as well as `DateImpl`).

There is one other problem. Suppose we now wish to switch to using some other class that implements the `Date` interface. We need to detect all the occurrences of:

```
new DateImpl( ... )
```

and change them to create instances of the new class. One of the principles of software engineering is that the commitment to some decision should be made in one place and not all over the place.

One way to do this which also overcomes the recompilation problem is to use the *factory pattern* (as described in [9]). We can introduce an interface (e.g., `Factory`) and a class (e.g., `FactoryImpl`) that are responsible for creating objects. They could have a method called `createDate`:

```
public Date createDate(final int pYear, final int pMonth, final int pDay)
{
   return new DateImpl(pYear, pMonth, pDay);
}
```

Clients should be written in terms of this method. For example, `PersonImpl`'s constructor contains:

```
iDateOfBirth = new DateImpl(pYear, pMonth, pDay);
```

This should be replaced by:

```
iDateOfBirth = iFactory.createDate(pYear, pMonth, pDay);
```

where `iFactory` is a variable of the interface type `Factory`.

If the `FactoryImpl` object is created in the program's program class, then only the program class, `FactoryImpl` and `DateImpl` will need to be recompiled if the code of the `DateImpl` class needs to be changed. Or, if `FactoryImpl` chooses to use some class other than `DateImpl`, then only the program class and `FactoryImpl` will need to be recompiled.

12 The names of interfaces and classes

In their book 'Exploring Java' ([12]), Niemeyer and Peck say: 'Interfaces define capabilities, so it's common to name interfaces after their capabilities in a passive sense. `Driveable` is a good example; `runnable` and `updatable` would be two more.'

In the book 'Java Design: Building Better Apps and Applets' ([2]), Coad and Mayfield say: 'Requiring interface names to end in `able` or `ible` is a bit too complicated a convention. ... Choose whatever prefix convention you prefer: `I`, `I_`, `Int_`; whatever. We prefer `I`.'

In 'ChiMu OO and Java Development: Guidelines and Resources' ([8]), Mark Fussell says: 'Interfaces should be given no suffixes or prefixes: they have the *normal* name space. Classes are given a suffix of `Class` if they are meant to be instantiated or are given a suffix of `AbsClass` if they are an abstract class that provides inheritable implementation but is not complete and instantiable by itself.'

In my teaching, I follow Fussell's advice for interfaces and use a suffix of `Impl` for classes.

13 Conclusions

This document has made some suggestions about the ways in which classes should be coded. It has also shown how a Java interface declaration can be used to separate out the interface and implementation aspects of a class.

The document also points out that this material is not usually taught in books that teach Java. The document is really a plug for 'Understanding Java', the author's book on Java ([5]).

14 References

1. Ken Arnold, James Gosling and David Holmes, 'The Java Programming Language, Third Edition', Addison-Wesley, 2000, 0-201-70433-1.

2. Peter Coad and Mark Mayfield, 'Java Design: Building Better Apps and Applets', Prentice Hall, 1996, 0-13-271149-4.

3. Barry Cornelius, 'Programming with TopSpeed Modula-2', Addison-Wesley, 1991, 0-201-41679-4.

4. Barry Cornelius, 'Teaching a Course on Understanding Java', http://www.ics.ltsn.ac.uk/pub/Jicc4/

5. Barry Cornelius, 'Understanding Java', Addison-Wesley, 2001, 0-201-71107-9.

6. Michael Cyberman, 'Smarter Java Technology Development', http://developer.java.sun.com/developer/technicalArticles/GUI/Interfaces/

7. Martin Fowler (with Kendall Scott), ‘UML Distilled’,
Addison-Wesley, 1997, 0-201-32563-2.

8. Mark Fussell, ‘ChiMu OO and Java Development: Guidelines and Resources’,
http://www.chimu.com/publications/javaStandards/

9. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, ‘Design Patterns: Elements of Reusable Object-Oriented Software’,
Addison-Wesley, 1995, 0-201-63361-2.

10. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, ‘The Java Language Specification, Second Edition’,
Addison-Wesley, 2000, 0-201-31008-2.

11. Peter Haggar, ‘Practical Java’,
Addison-Wesley, 2000, 0-201-61646-7.

12. Patrick Niemeyer and Joshua Peck, ‘Exploring Java’,
O’Reilly, 1996, 1-56592-184-4.

13. Arthur Riel, ‘Object-Oriented Design Heuristics’,
Addison-Wesley, 1996, 0-201-63385-X.

14. Mark Roulo, ‘How to avoid traps and correctly override methods from `java.lang.Object`’,
http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html

15. Bjarne Stroustrup, ‘The C++ Programming Language, First Edition’,
Addison-Wesley, 1986, 0-201-12078-X.

16. Bjarne Stroustrup, ‘The Design and Evolution of C++’,
Addison-Wesley, 1994, 0-201-54330-3.

17. Sun Microsystems, ‘java.lang.Comparable’,
http://java.sun.com/j2se/1.3/docs/api/java/lang/Comparable.html

18. Sun Microsystems, ‘java.util.Comparator’,
http://java.sun.com/j2se/1.3/docs/api/java/util/Comparator.html

19. Sun Microsystems, ‘java.lang.Object.equals’,
http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#equals(java.lang.Object)

20. Sun Microsystems, ‘java.lang.Object.hashCode’,
http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#hashCode()

21. Sun Microsystems, ‘java.util.Hashtable’,
http://java.sun.com/j2se/1.3/docs/api/java/util/Hashtable.html

22. Bill Venners, ‘The canonical object idiom’,
http://www.javaworld.com/javaworld/jw-10-1998/jw-10-techniques.html