Design of classes

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

1 *minimal public interface*

When teaching interfaces and classes, I stress that it is important that each class has:

- 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. When we produce a class, we may be uncertain as to 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 these things 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' ([4]), 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: it is entitled *The canonical object idiom* ([11]).

2 Providing `equals`

2.1 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.

Suppose a book introduces a class called `Date`. Then, if it 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`, `remove`, `indexOf`
`Map`	`containsKey`, `containsValue`, `get`, `put`, `remove`
`Set`	`add`, `contains`, `remove`

Instead, I teach that for the `equals` method to be useful it requires a parameter of type `Object`.

2.2 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 the `equals` method must be an object of class `Date`, and so this kind of call of `getClass` will always return 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 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 that is a subclass of the class `Date`.

2.3 Using `getClass` in preference to `instanceof`

When providing a proper version of `equals`, many authors (including [1]) 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` ([8]). 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, 1, 22);
NamedDate tNamedDate = new NamedDate(2001, 1, 22, "JICC5");
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 the book 'Java Language Specification' ([2])) 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' ([3]) by Peter Haggar.
- At [5], there is a WWW page 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'.

3 Providing `hashCode`

3.1 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` ([10]) 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`

3.2 The contract that `hashCode` should satisfy

The contract that `hashCode` needs to satisfy is given in the WWW pages that document the class `Object` ([9]). They say that the general contract of `hashCode` is:

- '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.'
- '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.'
- '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.'

3.3 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 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 want to provide a more sophisticated form of `hashCode`.

3.4 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 12 *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. 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 set up a `HashSet` containing 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:

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

The `add` method could use `Date`'s `hashCode` function to store the values 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 class 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 need only check 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.

3.5 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 altered.

If the class Date provides setYear, setMonth and setDay, we ought not to provide a hashCode function that is written in terms of the iYear, iMonth and/or iDay fields. So 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 a collection.

3.6 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 altered 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.

3.7 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 (at least) be dependent on the values of the fields used in the definition of equals.

3.8 What is the effect on performance?

The effect on performance was measured 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.

4 Providing compareTo

4.1 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. In fact, there are some parts of the Collections API (e.g., TreeSet and TreeMap) that work better if your class implements the Comparable interface from the package java.lang ([6]).

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 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);
```

4.2 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 ([7]). 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`.

If, 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:

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

5 Providing a clone

5.1 The need to provide a cloning operation

Few books 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.

5.2 Providing a constructor for cloning

Because it 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;
}
```

5.3 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 may want to 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();
      }
   }
}
```

5.4 Other information about cloning

For more information about how to clone objects, look at the *canonical object* article by Bill Venners ([11]).

6 References

1. Barry Cornelius, ‘Teaching a Course on Understanding Java’,
   http://www.ics.ltsn.ac.uk/pub/Jicc4/
2. James Gosling, Bill Joy, Guy Steele, Gilad Bracha, ‘The Java Language Specification, Second Edition’,
   Addison-Wesley, 2000, 0-201-31008-2
3. Peter Haggar, ‘Practical Java’,
   Addison-Wesley, 2000, 0-201-61646-7
4. Arthur Riel, ‘Object-Oriented Design Heuristics’,
   Addison-Wesley, 1996, 0-201-63385-X
5. 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
6. Sun Microsystems, ‘java.lang.Comparable’,
   http://java.sun.com/j2se/1.3/docs/api/java/lang/Comparable.html
7. Sun Microsystems, ‘java.util.Comparator’,
   http://java.sun.com/j2se/1.3/docs/api/java/util/Comparator.html
8. Sun Microsystems, ‘java.lang.Object.equals’,
   http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#equals(java.lang.Object)
9. Sun Microsystems, ‘java.lang.Object.hashCode’,
   http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html#hashCode()
10. Sun Microsystems, ‘java.util.Hashtable’,
    http://java.sun.com/j2se/1.3/docs/api/java/util/Hashtable.html
11. Bill Venners, ‘The canonical object idiom’,
    http://www.javaworld.com/javaworld/jw-10-1998/jw-10-techniques.html