

These are sample pages from Kari Laitinen's book  
"A Natural Introduction to Computer Programming with Java".  
For more information, please visit  
<http://www.naturalprogramming.com/javabook.html>

---

## CHAPTER 12

---

### INHERITANCE AND CLASS HIERARCHIES

The concept of inheritance is at the heart of object-oriented programming and object-oriented thinking. Just as a person in real life can inherit the features of his or her mother's face, a Java class can inherit the features of another class. In the computer world, the inheritance mechanism is more accurate and predictable than in real life. Inheritance makes it easier to declare new classes. That's why it is so important in object-oriented programming.

In this chapter, we shall first study some simple cases of inheritance. Later, we shall study some more complex cases and a special kind of methods called polymorphic methods. We shall see that polymorphic methods are a kind of exception to the general inheritance mechanism.

## 12.1 Base classes and derived classes

In the previous chapter, we studied a class named `Date`. Objects of the class `Date` contain the necessary data fields to store information about dates, and the class provides many useful methods to do various things with the information inside `Date` objects. With those methods we can compare `Date` objects, print `Date` objects, increment `Date` objects, etc.

One might think that class `Date` is so complete that we do not need other classes to be concerned with date information. Unfortunately things are not usually so simple. In different applications, slightly different date classes might be the most suitable classes to handle date information. In a large software project, for example, some people would like to modify class `Date` by adding new methods to it, while others would like to make it behave so that it automatically copies the computer system's date into the `Date` object being created. If different groups of people tried to modify class `Date` for their own purposes, there would most likely be different versions of the class. That might be dangerous in a software project because people might end up using the wrong class versions.

Inheritance is a mechanism that helps to manage classes which produce closely related objects. By using inheritance, it is easy to make slightly different versions of an existing class without having to modify the existing class. Inheritance is one of the key concepts in object-oriented programming. With inheritance it is possible to derive a new class from an existing class. Inheritance extends an existing class and produces a new class.

If a source program contains a class declaration that begins

```
class UpperClass
{
    ...
```

it is possible to write later in the same program or in some other program file

```
class LowerClass extends UpperClass
{
    ...
```

The latter declaration specifies that class `LowerClass` inherits class `UpperClass`. The keyword `extends` tells the compiler that the members (fields and methods) of class `UpperClass` must be inherited to `LowerClass`. Those members that are declared with access modifiers (keywords) `public` or `protected` in `UpperClass` become similar members in `LowerClass`. Those members that are declared without access modifiers in `UpperClass` become similar members in `LowerClass` provided that the classes belong to the same package of classes.

In the discussion above, the classes are named `UpperClass` and `LowerClass`. These names are chosen to indicate the fact that a class that inherits another class is on a lower level. It can exist only after the upper-level class has been declared. In the terminology of object-oriented programming, we say that `UpperClass` is a superclass or base class of `LowerClass`, `LowerClass` is a subclass of `UpperClass`, and `LowerClass` is derived from `UpperClass`. When a class inherits another class, the classes form a class hierarchy. The class that inherits is in a lower position in the hierarchy. Later on, we shall see that there can be more than just two classes in a class hierarchy.

Program `BetterDate.java` is an example where inheritance is used to derive a new class from class `Date`. The name of the derived class is `BetterDate`. Although the declaration of class `BetterDate` fits on a single page, `BetterDate` contains practically everything that is in class `Date` that needs ten pages to be described. The derived class `BetterDate` contains all fields of class `Date`, all methods excluding the constructors of class `Date`, its own constructor, and its own two methods `to_string_with_month_name()` and `to_american_format_string()`. It must be noted that a derived class needs, in most cases, a constructor of its own. Constructors are not inherited from the upper class.

Class `BetterDate` is better than class `Date` because it has two new methods. The first method allows a date to be shown so that the month is expressed with letters instead of numbers. The second method prints the date in the American format, also in the case when the object is created with the European format specifier. In addition to these new methods, it is possible to call the old class `Date` methods for `BetterDate` objects.

In one respect class `BetterDate` is worse than class `Date`. Because there is only one constructor method in class `BetterDate`, there remains only the following way to create `BetterDate` objects:

```
BetterDate date_of_reunification_of_germany
           = new BetterDate( "03.10.1990");
```

The constructor of `BetterDate` only accepts strings as its parameter. `Date` objects can also be constructed by supplying the initialization date as values of type `int`:

```
Date date_of_independence_of_finland
     = new Date( 6, 12, 1917 );
```

The sole constructor of class `BetterDate` passes the string that it receives as a parameter to the constructor of class `Date`. The constructor consists of the lines

```
public BetterDate( String date_as_string )
{
    super( date_as_string );
}
```

This constructor calls the constructor of class `Date` with the help of the `super` keyword. The keyword `super` refers to the constructor of the superclass. It is thus the constructor of class `Date` that actually processes the string and extracts the date information from it. Only the constructor of the immediate superclass can be called by using the `super` keyword. If there are higher classes above the immediate superclass, their constructors cannot be accessed in this way.

In general, it is possible to declare the constructor of a derived class in two ways. The first way is to call the constructor of the immediate superclass before the other statements of the constructor:

```
public DerivedClass( declarations of formal parameters )
{
    super( actual parameters to the constructor of superclass );
    zero or more other statements
}
```

The second possibility is to write the constructor without an explicit call to the constructor of the superclass:

```
public DerivedClass( declarations of formal parameters )
{
    zero or more statements
}
```

In the first case above, the constructor of the superclass is executed before the constructor of `DerivedClass`. In the second case, where the explicit call is missing, the compiler generates an implicit call to the default constructor of the superclass before executing the body of the constructor of `DerivedClass`. Implicit calling means that the call is not visible in the source program text, and the compiler generates it automatically. Because the compiler generates these automatic calls to the constructors of upper classes, there usually has to be a default constructor (i.e. a constructor that can be called without supplying parameters) in every class that is to be inherited.

Class `BetterDate` is better than class `Date` because it has two additional methods. We can say that class `BetterDate` is the same as class `Date` plus the two methods. `BetterDate` is thus an extended version of class `Date`.

By writing `extends Date` after the name of the new class, we can inherit the fields and methods from class `Date`.

```
// BetterDate.java
class BetterDate extends Date
{
    public BetterDate( String date_as_string )
    {
        super( date_as_string );
    }

    public String to_string_with_month_name()
    {
        String[] names_of_months =

            { "January", "February", "March", "April",
              "May", "June", "July", "August",
              "September", "October", "November", "December" };

        return ( names_of_months[ this_month - 1 ] + " "
                 + this_day + ", " + this_year );
    }

    public String to_american_format_string()
    {
        char saved_date_print_format = date_print_format ;

        date_print_format = 'A' ;

        String string_to_return = this.toString() ;

        date_print_format = saved_date_print_format ;

        return string_to_return ;
    }
}
```

Constructors are not inherited when a class inherits the members of another class. Constructors need to be written in most cases when new classes are derived from existing classes. Here the constructor of `BetterDate` is written so that it calls the constructor of class `Date`. The keyword `super` refers here to a constructor in the superclass.

`date_print_format` is a field in class `Date`. Because `BetterDate` inherits class `Date`, this field can be used in a method of `BetterDate` as if it was declared in class `BetterDate`.

Method `toString()` is inherited from class `Date`, and here it is called for "this" `BetterDate` object. `toString()` returns a string representation of "this" date in its current printing format. The method could also be called without the `this` keyword and the dot operator.

**BetterDate.java - 1: Deriving a new class from an existing class.**

The `toString()` method is called automatically for a `BetterDate` object when operator `+` is used as the string concatenation operator.

Objects of derived classes are created in the normal way. We cannot see anything here indicating that `BetterDate` is derived from class `Date`.

```
class BetterDateTester
{
    public static void main( String[] not_in_use )
    {
        BetterDate birthday_of_einstein = new BetterDate("14.03.1879");

        System.out.print( "\n Albert Einstein was born on "
            + birthday_of_einstein );

        birthday_of_einstein.increment();

        System.out.print( "\n Albert was one day old on "
            + birthday_of_einstein.to_string_with_month_name() );

        birthday_of_einstein.increment();

        System.out.print( "\n Albert was two days old on "
            + birthday_of_einstein.to_american_format_string() );
    }
}
```

This statement is, for example, "hard evidence" confirming the fact that inheritance has indeed taken place. The object reference `birthday_of_einstein` is pointing to a `BetterDate` object, and method `increment()` is invoked for the object. Because the `BetterDate` class does not contain a method named `increment()`, the method must have been inherited from class `Date`.

### BetterDate.java - 2. Using a BetterDate object.

```
D:\javafiles3>java BetterDateTester

Albert Einstein was born on 14.03.1879
Albert was one day old on March 15, 1879
Albert was two days old on 03/16/1879
```

### BetterDate.java - X. Dates printed with different methods.

Program **CurrentDate.java** is another example where a new class is derived from class **Date**. The name of the class is **CurrentDate**. Classes **Date** and **CurrentDate** only differ from each other in that they have different kinds of constructors. The constructor of class **CurrentDate** reads the date information that is maintained by the operating system of the computer. The constructor gets this information by using the standard Java classes **Calendar** and **GregorianCalendar**. The used standard method gets the computer's date by interacting with the operating system of the computer which, in turn, interacts with the clock electronics of the computer.

Programs **Titanic.java** and **Weddingdates.java** are examples in which objects of class **CurrentDate** are created and used. When a **CurrentDate** object is created, its date is the current computer's date. It is, nonetheless, possible to change the initial date with the methods that class **CurrentDate** has inherited from class **Date**. For example, methods **increment()** and **decrement()** modify a **CurrentDate** object just as they modify a **Date** object.

**CurrentDate** objects are, by default, American style dates, which means that they are shown in format MM/DD/YYYY. The default print format in class **CurrentDate** is 'A'. If you want a **CurrentDate** object to be shown in the European format DD.MM.YYYY, you must create the object in the following way

```
CurrentDate date_of_today = new CurrentDate( 'E' );
```

The above object creation statement invokes the second constructor of class **CurrentDate**. There is no technical reason why **CurrentDate** objects are by default American dates. This book just tries to achieve some kind of "international balance". As the objects of class **Date** are by default shown in the European date format, objects of class **CurrentDate** are by default American dates. You may, of course, modify these classes to make their default settings to fit your own preferences.

Two new classes are derived from class **Date** in programs **BetterDate.java** and **CurrentDate.java**. Figure 12-1 shows how these class derivations can be described graphically as a UML class diagram. While studying Figure 12-1, you should note that also the empty spaces in the graphical class descriptions have a meaning.

A new class can be derived from an existing base class in the following basic ways

- The new class has the same members as its base class, but it has one or more new kinds of constructors. (Class **CurrentDate** is like this.)
- The new class has the same fields and methods as its base class, and it has some additional methods. (Class **BetterDate** is like this.)
- The new class has some new fields in addition to the inherited fields and methods. (We shall study these kinds of classes later in this book, and we shall also learn that it is not always necessary to inherit all the methods of a base class.)

In practice, new classes are usually formed by mixing all the above basic ways to derive new classes. Especially when new fields are added to a new class, it is usually necessary to add new methods as well. In most cases derived classes need their own kinds of constructors. When new fields are added to a derived class, the objects of the derived class consume more memory space from the heap memory than the objects of the base class.

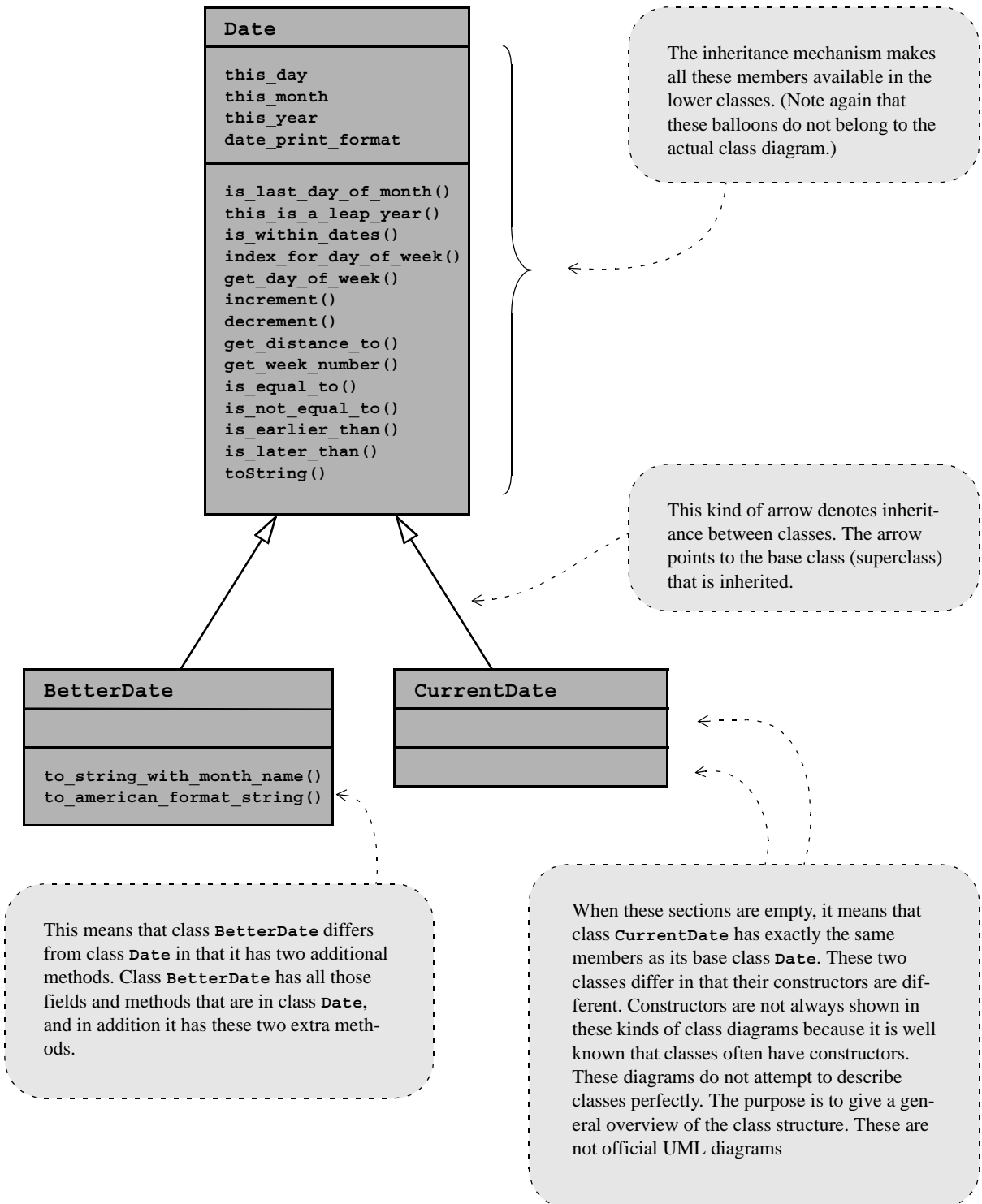


Figure 12-1. Class diagram that describes relationships between three classes.

Here, class `CurrentDate` inherits class `Date`. The methods of class `Date` are available methods for the objects of class `CurrentDate`.

```
// CurrentDate.java (c) Kari Laitinen
import java.util.Calendar ;
import java.util.GregorianCalendar ;

class CurrentDate extends Date
{
    public CurrentDate()
    {
        Calendar current_system_date = new GregorianCalendar() ;

        this_day = current_system_date.get( Calendar.DAY_OF_MONTH ) ;
        this_month = current_system_date.get( Calendar.MONTH ) + 1 ;
        this_year = current_system_date.get( Calendar.YEAR ) ;

        date_print_format = 'A' ;
    }

    public CurrentDate( char given_date_print_format )
    {
        this() ;
        date_print_format = given_date_print_format ;
    }
}
```

`CurrentDate` differs from its superclass `Date` only in that its constructors are different. The first constructor assumes the American date print format.

The first constructor creates an object of the standard Java class `GregorianCalendar` from which it can read the date information that is maintained by the operating system of the computer. After these assignment statements have been executed, this `CurrentDate` object contains the date of the computer.

`Calendar` and `GregorianCalendar` are standard Java classes for handling information related to dates and time in Java programs. These classes will be discussed in a later chapter.

When you compile programs which use the `CurrentDate` class (e.g. `Titanic.java` and `Weddingdates.java`), you should have the files `CurrentDate.java`, `Date.java`, and `DateDistance.java` in the same folder (directory) as the file being compiled. The file `DateDistance.java` is always needed because `Date.java` cannot be compiled without it.

The second constructor of class `CurrentDate` first calls the first constructor with the help of the `this` keyword, and then sets the `date_print_format` to the given value.

### `CurrentDate.java` - 1. The declaration of class `CurrentDate`.

Objects of class `CurrentDate` are usually created without supplying any parameters to the constructor of the class. This declaration results in that the object referenced by `date_of_today` contains the date information that is maintained by the operating system of the computer where this program is being executed.

```
// Titanic.java (c) Kari Laitinen

class Titanic
{
    public static void main( String[] not_in_use )
    {
        Date date_when_titanic_sank = new Date( "04/15/1912" ) ;

        CurrentDate date_of_today = new CurrentDate() ;

        DateDistance time_from_sinking =

            date_of_today.get_distance_to( date_when_titanic_sank ) ;

        System.out.print( "\n Today it is " + date_of_today
            + ".\n On " + date_when_titanic_sank
            + ", the famous ship \"Titanic\" went to"
            + "\n the bottom of Atlantic Ocean."
            + "\n That happened "
            + time_from_sinking.years + " years, "
            + time_from_sinking.months + " months, and "
            + time_from_sinking.days + " days ago. \n\n" ) ;
    }
}
```

Because of inheritance, the methods of class `Date` can be called for `CurrentDate` objects. Here, for example, methods `get_distance_to()` and `toString()` are invoked for the `CurrentDate` object pointed by `date_of_today`.

**Titanic.java - 1.** A program that uses both a `Date` object and a `CurrentDate` object.

```
D:\javafiles3>java Titanic

Today it is 03/01/2005.
On 04/15/1912, the famous ship "Titanic" went to
the bottom of Atlantic Ocean.
That happened 92 years, 10 months, and 16 days ago.
```

**Titanic.java - X.** Here the program is executed on March 1, 2005.

```
// Weddingdates.java (c) Kari Laitinen

class Weddingdates
{
    public static void main( String[] not_in_use )
    {
        CurrentDate date_to_increment = new CurrentDate() ;

        int number_of_dates_printed = 0 ;

        System.out.print( "\n These are easy-to-remember dates for weddings and"
            + "\n other important events because the days and months"
            + "\n consist of the digits used in the year: \n" ) ;

        while ( number_of_dates_printed < 60 )
        {
            String day_as_string =
                String.format( "%02d", date_to_increment.day() ) ;
            String month_as_string =
                String.format( "%02d", date_to_increment.month() ) ;
            String year_as_string = "" + date_to_increment.year() ;

            if ( year_as_string.indexOf( day_as_string.charAt( 0 ) ) != -1 &&
                year_as_string.indexOf( day_as_string.charAt( 1 ) ) != -1 &&
                year_as_string.indexOf( month_as_string.charAt( 0 ) ) != -1 &&
                year_as_string.indexOf( month_as_string.charAt( 1 ) ) != -1 )
            {
                // Now we have found a date that meets our requirements.

                if ( number_of_dates_printed % 5 == 0 )
                {
                    System.out.print( "\n" ) ;
                }

                System.out.print( " " + date_to_increment ) ;

                number_of_dates_printed ++ ;
            }

            date_to_increment.increment() ;
        }
    }
}
```

Right after its creation, the object referenced by `date_to_increment` contains the current date of the computer. The object can be used like a `Date` object.

The object referenced by `date_to_increment` is incremented to the next date. This program simply checks hundreds of dates to see which dates fulfil the criteria for a nice wedding date.

A valid wedding date has been found here. A newline is printed after every 5th date. The string concatenation operator `+` converts the `CurrentDate` object to a string. The method `toString()` that is provided in class `Date`, and inherited to class `CurrentDate`, is invoked automatically in the conversion operation.

**Weddingdates.java - 1.+ Using a `CurrentDate` object to find the next best wedding dates.**

This boolean expression tests whether the object referenced by `date_to_increment` contains a nice wedding date. The beginning of the `if` construct could be put into words like: "If the day of the date and the month of the date contain only such digits that can be found in the year of the date, ...". Method `indexOf()` returns the value -1 only when it cannot find the given character in the string for which it was called.

For example, date 02/06/2006 would be found to be an acceptable wedding date because all digits in strings "02" and "06" can be found in string "2006".

The three "components" of the date are converted to `String` objects here. The format specifier `%02d` ensures that single digit days and months are converted to strings that have a leading zero digit. The `format()` method of class `String` uses the same format specifiers as the method `System.out.printf()`.

```
while ( number_of_dates_printed < 60 )
{
    String day_as_string =
        String.format( "%02d", date_to_increment.day() );
    String month_as_string =
        String.format( "%02d", date_to_increment.month() );
    String year_as_string = "" + date_to_increment.year();

    if ( year_as_string.indexOf( day_as_string.charAt( 0 ) ) != -1 &&
        year_as_string.indexOf( day_as_string.charAt( 1 ) ) != -1 &&
        year_as_string.indexOf( month_as_string.charAt( 0 ) ) != -1 &&
        year_as_string.indexOf( month_as_string.charAt( 1 ) ) != -1 )
    {
        // Now we have found a date that meets our requirements.
    }
}
```

### Weddingdates.java - 1-1. The mechanism to find a nice wedding date.

```
D:\javafiles3>java weddingdates
```

These are easy-to-remember dates for weddings and other important events because the days and months consist of the digits used in the year:

```
02/02/2006  02/06/2006  02/20/2006  02/22/2006  02/26/2006
06/02/2006  06/06/2006  06/20/2006  06/22/2006  06/26/2006
02/02/2007  02/07/2007  02/20/2007  02/22/2007  02/27/2007
07/02/2007  07/07/2007  07/20/2007  07/22/2007  07/27/2007
02/02/2008  02/08/2008  02/20/2008  02/22/2008  02/28/2008
08/02/2008  08/08/2008  08/20/2008  08/22/2008  08/28/2008
02/02/2009  02/09/2009  02/20/2009  02/22/2009  09/02/2009
09/09/2009  09/20/2009  09/22/2009  09/29/2009  01/01/2010
01/02/2010  01/10/2010  01/11/2010  01/12/2010  01/20/2010
01/21/2010  01/22/2010  02/01/2010  02/02/2010  02/10/2010
02/11/2010  02/12/2010  02/20/2010  02/21/2010  02/22/2010
10/01/2010  10/02/2010  10/10/2010  10/11/2010  10/12/2010
```

### Weddingdates.java - X. The program prints 60 dates in the MM/DD/YYYY format.