

4C

Classes: Access Control

1 Access control

Each method, constructor, and variable within a class, whether static or dynamic, may be marked as either `public` or `private`. If neither is stated, then the default is `public` (but Java requires an explicit writing of `public` in the declaration of `main()`, and in some other situations which we will mention when we meet them). These keywords do not bring any new power, in the sense that they do not enable us to write programs that we couldn't write without them. Their role is stylistic

The basic rule is that `private` entities cannot be accessed *outside the class* in which they are defined. For example:

```
class Pair {
    int x;          // x is public by default
    private int y; // y is private

    private void meth1(Pair p) {
        .....
        int tempx = p.x; // okay
        int tempy = p.y; // okay (we're still inside the class where y is defined)
        .....
    }
    .....
}

class UsePair {
    public static void main(String args[] ) {
        Pair r = new Pair ();
    }
}
```

```

Pair q = new Pair();
r.x = 1;      // okay as x public
r.y = 2;      // ILLEGAL as y private and we're outside the class
r.meth1(p);   // ILLEGAL as meth1 private and we're outside the class
    }
}

```

Remember that the attributes `public` and `private` only constrain code *outside* the class in which they occur.

It is standard programming practice to make all the instance variables in a class private. Methods should be marked private if they are written solely to help us to write another method in the same class – in other words, if they were not part of the formal requirements for the class. For example, recall the first example of `Date` we wrote earlier; it is reproduced in outline below with its variables and methods marked as private where appropriate:

```

class Date {
    private int day; private int month; private int year;

    void getDate() {
        ...
    }

    private String monthName() { // not intended to be invoked from outside
        ...
    }

    void putDate() { // in form 23rd March 2006
        ...
    }
}

```

The instance variables `day`, `month`, and `year` are marked private as is usually the case. In addition, method `monthName` is marked private because it was introduced only during the coding process to help us write `putDate`.

It is not easy for beginners to see the value of `public` and `private`. Indeed, if we replace all occurrences of `private` in a program with `public`, then the program will compute the same result. However, if you earn your living updating other peoples programs, then you will pray that they have employed `private` variables and methods as much as possible. Professional programmers always privatise when possible. Regular changes in large working programs are the norm, and it is easier to change programs where the internal details of classes are kept hidden (by making them `private`). If hidden internal details of a class are changed, it will have no impact on code outside the class. If `public` details are changed, however, it is likely that other parts of the program will have to be changed also, and that can be time-consuming and difficult in large programs.

The private/public trap

Only variables declared at class level may be qualified as `private` or `public`. Local variables (i.e. those declared inside methods) are never so marked:

```
class Duration {  
  
    private int counter = 0; // okay  
  
    public static void main(String[] args) {  
        private int length = 0;    // WRONG!  
        .....  
    }  
}
```

2 Getters and setters

When instance variables are `private`, code outside the class cannot access them directly either to inspect them or to change them. We may include for some or all instance variables a function which simply returns its value; such functions are called “getters”. We may also include functions which assign a new value to an instance variable; these functions are called “setters”. In the following example, `getYear` is a getter and `setYear` is a setter.

```
class Date {  
  
    private int day; private int month; private int year;  
  
    int getYear() { return day;}           // getter
```

```
void setYear(int y) { year = y;}    // setter

.....
}
```

You should hesitate a moment before you write a getter or setter, and you should use them sparingly and only for very simple purposes.

The getter/setter trap

The use of setters and getters sometimes reveals a poor programming style, where the programmer is using them to avoid a correct object-oriented design. They should never be used to code an action outside a class that should properly be written as a method within it.

For example, suppose we are writing a program about dates, for which we are using the `Date` class as described above. Suppose in the main program we want to determine whether some date `p` is a leap year in the 21st century; we might be tempted to write:

```
int y = p.getYear();
if ((2000<=y && y<2100 && y%4==0) ....
```

While this code is technically correct it is very bad style. The object-oriented style insists that we code all the operations on dates within the `Date` class. So we should add the following method to the `Date` class:

```
boolean isLeap() {
    return (2000<=year && year<2100 && year%4==0);
}
```

and change the code in the main program to

```
if (p.isLeap()) ....
```