CHAPTER 2

Classes and Objects

Work is the curse of the drinking classes.
—Oscar Wilde

Chapter Goals

- Objects and classes
- Encapsulation
- References
- Keywords `public`, `private`, and `static`
- Methods
- Scope of variables

OBJECTS

Every program that you write involves at least one thing that is being created or manipulated by the program. This thing, together with the operations that manipulate it, is called an *object*.

Consider, for example, a program that must test the validity of a four-digit code number that a person will enter to be able to use a photocopy machine. Rules for validity are provided. The object is a four-digit code number. Some of the operations to manipulate the object could be `readNumber`, `getSeparateDigits`, `testValidity`, and `writeNumber`.

Any given program can have several different types of objects. For example, a program that maintains a database of all books in a library has at least two objects:

1. A `Book` object, with operations like `getTitle`, `isOnShelf`, `isFiction`, and `goOutOfPrint`.
2. A `ListOfBooks` object, with operations like `search`, `addBook`, `removeBook`, and `sortByAuthor`.

An object is characterized by its *state* and *behavior*. For example, a book has a state described by its title, author, whether it's on the shelf, and so on. It also has behavior like going out of print.

Notice that an object is an idea, separate from the concrete details of a programming language. It corresponds to some real-world object that is being represented by the program.

PUBLIC, PRIVATE, AND STATIC

The keyword `public` preceding the class declaration signals that the class is usable by all *client programs*. If a class is not public, it can be used only by classes in its own package. In the AP Java subset, all classes are public.

Similarly, *public methods* are accessible to all client programs. Clients, however, are not privy to the class implementation and may not access the private instance variables and private methods of the class. Restriction of access is known as *information hiding*. In Java, this is implemented by using the keyword `private`. *Private methods and variables in a class can be accessed only by methods of that class.* Even though Java allows public instance variables, in the AP Java subset all instance variables are private.

A *static variable* (class variable) contains a value that is shared by all instances of the class. "Static" means that memory allocation happens once.

Typical uses of a static variable are to

- keep track of statistics for objects of the class.
- accumulate a total.
- provide a new identity number for each new object of the class.

For example:

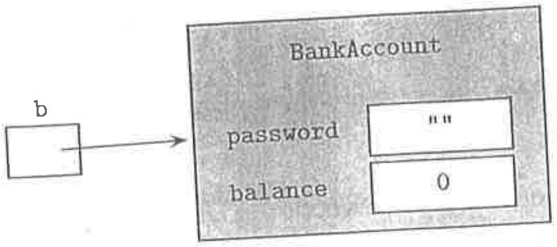
```
public class Employee
{
    private String name;
    private static int employeeCount = 0; //number of employees

    public Employee( < parameter list > )
    {
        < initialization of private instance variables >
        employeeCount++; //increment count of all employees
    }
        ...
}
```

Notice that the static variable was initialized outside the constructor and that its value can be changed.

Static final variables (constants) in a class cannot be changed. They are often declared public (see some examples of `Math` class constants on p. 183). The variable `OVERDRAWN_PENALTY` is an example in the `BankAccount` class. Since the variable is public, it can be used in any client method. The keyword `static` indicates that there is a single value of the variable that applies to the whole class, rather than a new instance for each object of the class. A client method would refer to the variable as `BankAccount.OVERDRAWN_PENALTY`. In its own class it is referred to as simply `OVERDRAWN_PENALTY`.

See p. 97 for static methods.



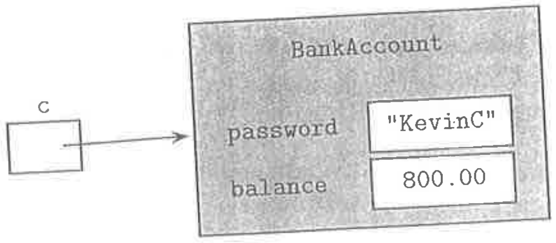
2. The constructor with parameters sets the instance variables of a BankAccount object to the values of those parameters.

   Here is the implementation:

```
/** Constructor. Constructs a bank account with
 *  specified password and balance. */
public BankAccount(String acctPassword, double acctBalance)
{
    password = acctPassword;
    balance = acctBalance;
}
```

   In a client program a declaration that uses this constructor needs matching parameters:

```
BankAccount c = new BankAccount("KevinC", 800.00);
```



NOTE

b and c are *object variables* that store the *addresses* of their respective BankAccount objects. They do not store the objects themselves (see References on p. 101).

ACCESSORS

An *accessor method* accesses a class object without altering the object. An accessor returns some information about the object.

The BankAccount class has a single accessor method, getBalance(). Here is its implementation:

```
/** @return the  balance of this account */
public double getBalance()
{ return balance; }
```

A client program may use this method as follows:

```
BankAccount b1 = new BankAccount("MattW", 500.00);
BankAccount b2 = new BankAccount("DannyB", 650.50);
if (b1.getBalance() > b2.getBalance())
    ...
```

The implementation of a static method uses the keyword `static` in its header. There is no implied object in the code (as there is in an instance method). Thus, if the code tries to call an instance method or invoke a private instance variable for this nonexistent object, a syntax error will occur. A static method can, however, use a static variable in its code. For example, in the `Employee` example on p. 94, you could add a static method that returns the `employeeCount`:

```
public static int getEmployeeCount()
{ return employeeCount; }
```

Here's an example of a static method that might be used in the `BankAccount` class. Suppose the class has a static variable `intRate`, declared as follows:

```
private static double intRate;
```

The static method `getInterestRate` may be as follows:

```
public static double getInterestRate()
{
    System.out.println("Enter interest rate for bank account");
    System.out.println("Enter in decimal form:");
    intRate = IO.readDouble();          // read user input
    return intRate;
}
```

Since the rate that's read in by this method applies to all bank accounts in the class, not to any particular `BankAccount` object, it's appropriate that the method should be static.

Recall that an instance method is invoked in a client program by using an object variable followed by the dot operator followed by the method name:

```
BankAccount b = new BankAccount();  //invokes the deposit method for
b.deposit(acctPassword, amount);    //BankAccount object b
```

A static method, by contrast, is invoked by using the *class name* with the dot operator:

```
double interestRate = BankAccount.getInterestRate();
```

Static Methods in a Driver Class Often a class that contains the `main()` method is used as a driver program to test other classes. Usually such a class creates no objects of the class. So all the methods in the class must be static. Note that at the start of program execution, no objects exist yet. So the `main()` method must *always* be static.

For example, here is a program that tests a class for reading integers entered at the keyboard.

```
import java.util.*;
public class GetListTest
{
    /** @return a list of integers from the keyboard */
    public static List<Integer> getList()
    {
        List<Integer> a = new ArrayList<Integer>();
        < code to read integers into a>
        return a;
    }
```

SCOPE

The *scope* of a variable or method is the region in which that variable or method is visible and can be accessed.

The instance variables, static variables, and methods of a class belong to that class's scope, which extends from the opening brace to the closing brace of the class definition. Within the class all instance variables and methods are accessible and can be referred to simply by name (no dot operator!).

A *local variable* is defined inside a method. It can even be defined inside a statement. Its scope extends from the point where it is declared to the end of the block in which its declaration occurs. A *block* is a piece of code enclosed in a {} pair. When a block is exited, the memory for a local variable is automatically recycled.

Local variables take precedence over instance variables with the same name. (Using the same name, however, creates ambiguity for the programmer, leading to errors. You should avoid the practice.)

The this Keyword

An instance method is always called for a particular object. This object is an *implicit parameter* for the method and is referred to with the keyword `this`. You are expected to know this vocabulary for the exam.

In the implementation of instance methods, all instance variables can be written with the prefix `this` followed by the dot operator.

Example 1

In the method call `obj.doSomething("Mary",num)`, where `obj` is some class object and `doSomething` is a method of that class, `"Mary"` and `num`, the parameters in parentheses, are *explicit* parameters, whereas `obj` is an *implicit* parameter.

Example 2

Here's an example where `this` is used as a parameter.

```
public class Person
{
    private String name;
    private int age;

    public Person(String aName, int anAge)
    {
        name = aName;
        age = anAge;
    }

    /** @return the String form of this person */
    public String toString()
    { return name + " " + age; }

    public void printPerson()
    { System.out.println(this); }

    //Other variables and methods are not shown.
}
```

```
int num1 = 3;
int num2 = num1;
```

The variables num1 and num2 can be thought of as memory slots, labeled num1 and num2, respectively:

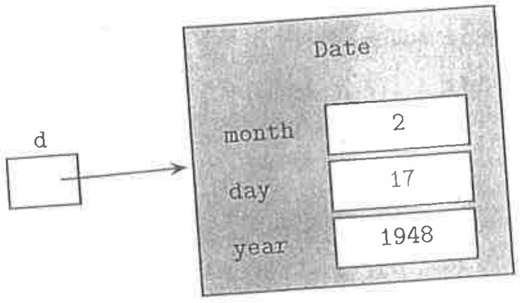
num1
3
num2
3

If either of the above variables is now changed, the other is not affected. Each has its own memory slot.

Contrast this with the declaration of a reference data type. Recall that an object is created using new:

```
Date d = new Date(2, 17, 1948);
```

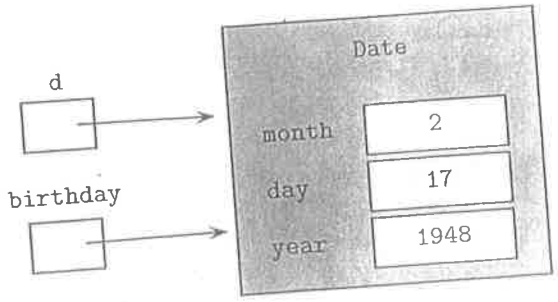
This declaration creates a reference variable d that refers to a Date object. The value of d is the address in memory of that object:



Suppose the following declaration is now made:

```
Date birthday = d;
```

This statement creates the reference variable birthday, which contains the same address as d:



Having two references for the same object is known as *aliasing*. Aliasing can cause unintended problems for the programmer. The statement

```
d.changeDate();
```

will automatically change the object referred to by birthday as well.

What the programmer probably intended was to create a second object called birthday whose attributes exactly matched those of d. This cannot be accomplished without using new. For example,

```
Date birthday = new Date(d.getMonth(), d.getDay(), d.getYear());
```

The statement d.changeDate() will now leave the birthday object unchanged.

```
BankAccount b = new BankAccount("TimB", 1000);
b.withdraw("TimB", 250);
```

Here "TimB" and 250 are the actual parameters that match up with `acctPassword` and `amount` for the `withdraw` method.

NOTE

1. The number of arguments in the method call must equal the number of parameters in the method header, and the type of each argument must be compatible with the type of each corresponding parameter.
2. In addition to its explicit parameters, the `withdraw` method has an implicit parameter, `this`, the `BankAccount` from which money will be withdrawn. In the method call

   ```
   b.withdraw("TimB", 250);
   ```

   the actual parameter that matches up with `this` is the object reference `b`.

PASSING PRIMITIVE TYPES AS PARAMETERS

Parameters are *passed by value*. For primitive types this means that when a method is called, a new memory slot is allocated for each parameter. The value of each argument is copied into the newly created memory slot corresponding to each parameter.

During execution of the method, the parameters are local to that method. *Any changes made to the parameters will not affect the values of the arguments in the calling program.* When the method is exited, the local memory slots for the parameters are erased.

Here's an example: What will the output be?

```
public class ParamTest
{
    public static void foo(int x, double y)
    {
        x = 3;
        y = 2.5;
    }

    public static void main(String[] args)
    {
        int a = 7;
        double b = 6.5;
        foo(a, b);
        System.out.println(a + "  " + b);
    }
}
```

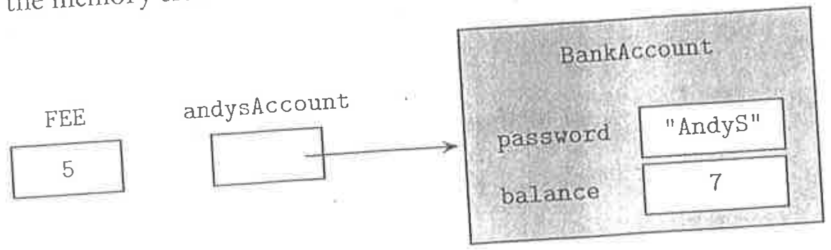
The output will be

```
7  6.5
```

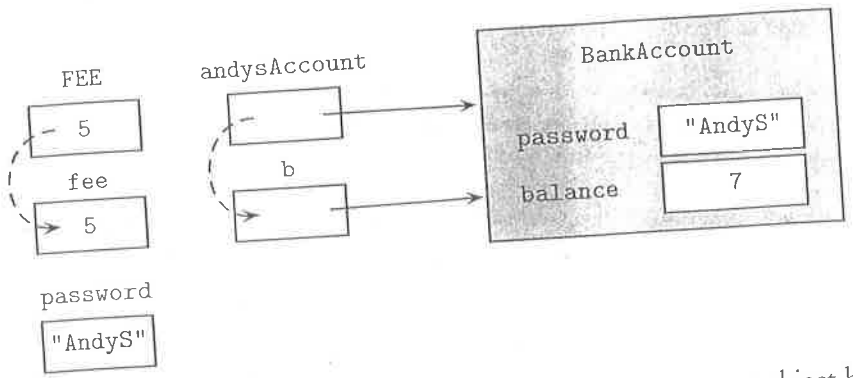
The arguments `a` and `b` remain unchanged, despite the method call!

This can be understood by picturing the state of the memory slots during execution of the program.

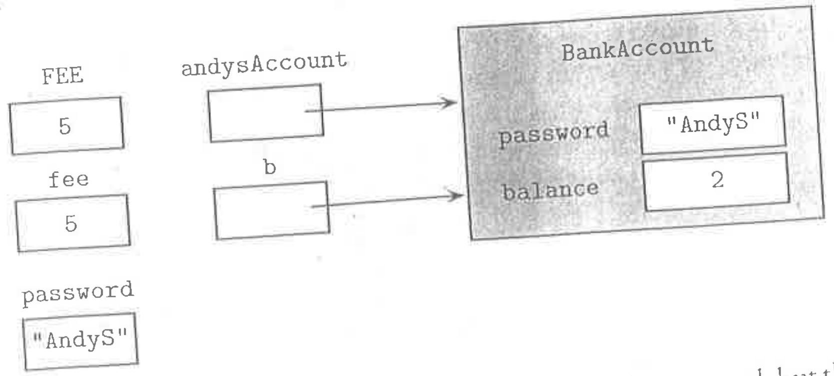
Here are the memory slots before the `chargeFee` method call:



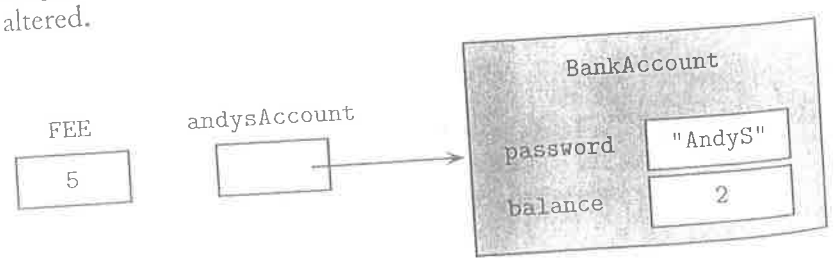
At the time of the `chargeFee` method call, copies of the matching parameters are made:



Just before exiting the method: The `balance` field of the `BankAccount` object has been changed.

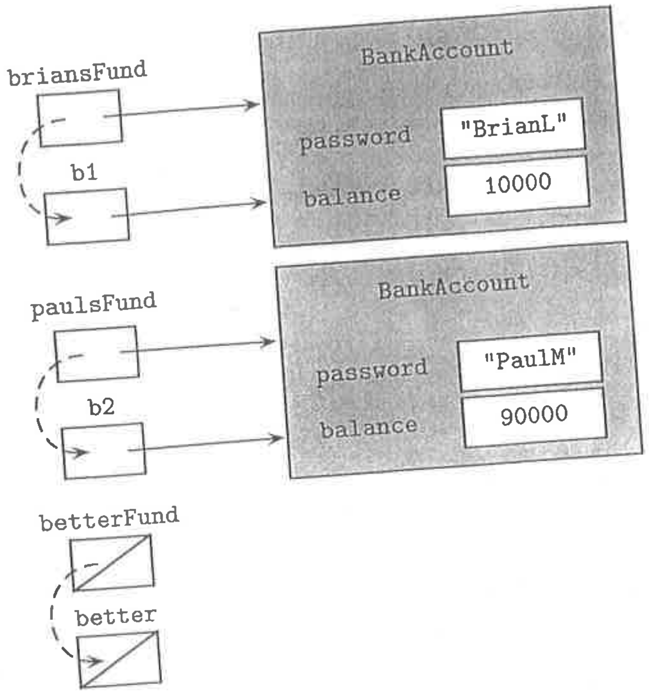


After exiting the method: All parameter memory slots have been erased, but the object remains altered.

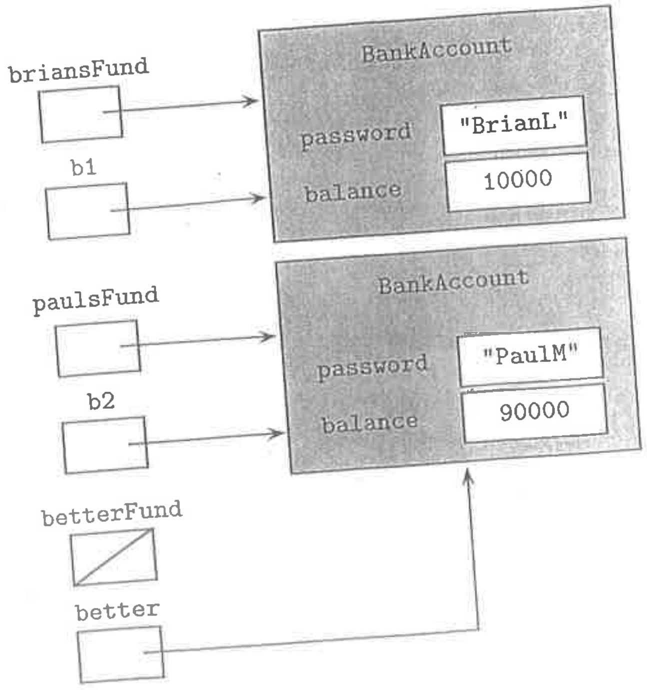


NOTE

The `andysAccount` reference is unchanged throughout the program segment. The object to which it refers, however, has been changed. This is significant. Contrast this with Example 2 below in which an attempt is made to replace the object itself.



Just before exiting the method: The value of better has been changed; betterFund, however, remains unchanged.

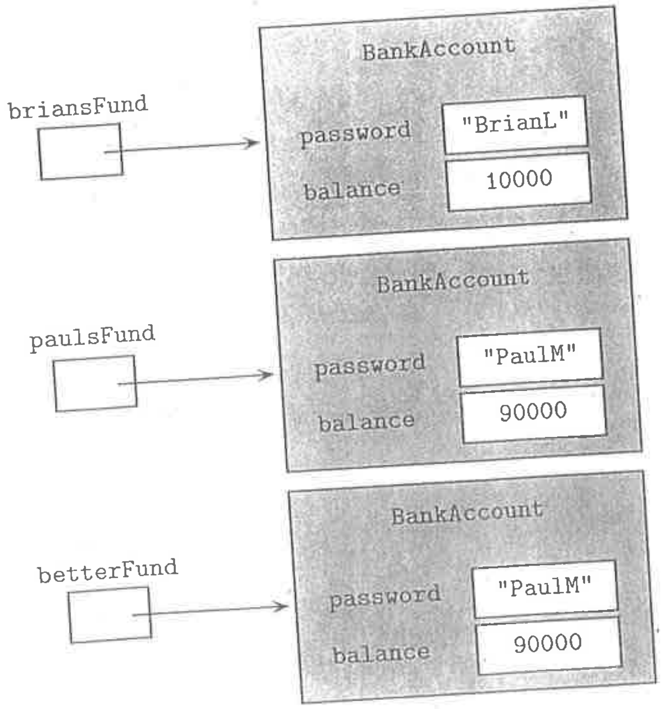


After exiting the method: All parameter slots have been erased.

What the method does *not* do is create a new object to which `betterFund` refers. To do that would require the keyword `new` and use of a `BankAccount` constructor. Assuming that a `getPassword()` accessor has been added to the `BankAccount` class, the code would look like this:

```
public static BankAccount chooseBestAccount(BankAccount b1,
        BankAccount b2)
{
    BankAccount better;
    if (b1.getBalance() > b2.getBalance())
        better = new BankAccount(b1.getPassword(), b1.getBalance());
    else
        better = new BankAccount(b2.getPassword(), b2.getBalance());
    return better;
}
```

Using this modified method with the same `main()` method above has the following effect:



Modifying more than one object in a method can be accomplished using a *wrapper class* (see p. 180).

Chapter Summary

By now you should be able to write code for any given object, with its private data fields and methods encapsulated in a class. Be sure that you know the various types of methods—static, instance, and overloaded.

You should also understand the difference between storage of primitive types and the references used for objects.