





METHODS

Headers

All method headers, with the exception of constructors (see below) and static methods (p. 97), look like this:

<code>public</code>	<code>void</code>	<code>withdraw</code>	<code>(String password, double amount)</code>
			
access specifier	return type	method name	parameter list

NOTE

1. The *access specifier* tells which other methods can call this method (see Public, Private, and Static on the previous page).
2. A *return type* of void signals that the method does not return a value.
3. Items in the *parameter list* are separated by commas.

The implementation of the method directly follows the header, enclosed in a {} block.

Types of Methods

CONSTRUCTORS

A *constructor* creates an object of the class. You can recognize a constructor by its name—always the same as the class. Also, a constructor has no return type.

Having several constructors provides different ways of initializing class objects. For example, there are two constructors in the `BankAccount` class.

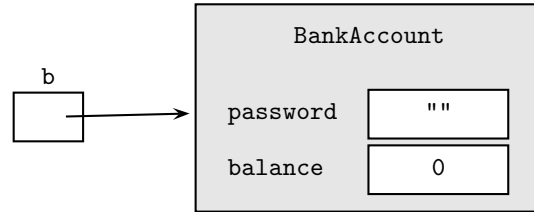
1. The *default constructor* has no arguments. It provides reasonable initial values for an object. Here is its implementation:

```
/** Default constructor.
 * Constructs a bank account with default values. */
public BankAccount()
{
    password = "";
    balance = 0.0;
}
```

In a client method, the declaration

```
BankAccount b = new BankAccount();
```

constructs a `BankAccount` object with a balance of zero and a password equal to the empty string. The `new` operator returns the address of this newly constructed object. The variable `b` is assigned the value of this address—we say “`b` is a *reference* to the object.” Picture the setup like this:



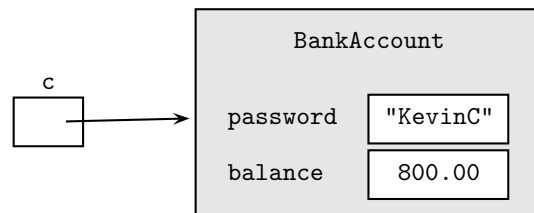
2. The constructor with parameters sets the instance variables of a `BankAccount` object to the values of those parameters.

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())
    ...
```

NOTE

The *. operator* (dot operator) indicates that `getBalance()` is a method of the class to which `b1` and `b2` belong, namely the `BankAccount` class.

MUTATORS

A *mutator method* changes the state of an object by modifying at least one of its instance variables.

Here are the implementations of the `deposit` and `withdraw` methods, each of which alters the value of `balance` in the `BankAccount` class:

```
/** Deposits amount in a bank account with the given password.
 * @param acctPassword the password of this bank account
 * @param amount the amount to be deposited
 */
public void deposit(String acctPassword, double amount)
{
    if (!acctPassword.equals(password))
        /* throw an exception */
    else
        balance += amount;
}

/** Withdraws amount from bank account with given password.
 * Assesses penalty if balance is less than amount.
 * @param acctPassword the password of this bank account
 * @param amount the amount to be withdrawn
 */
public void withdraw(String acctPassword, double amount)
{
    if (!acctPassword.equals(password))
        /* throw an exception */
    else
    {
        balance -= amount;        //allows negative balance
        if (balance < 0)
            balance -= OVERDRAWN_PENALTY;
    }
}
```

A mutator method in a client program is invoked in the same way as an accessor: using an object variable with the dot operator. For example, assuming valid `BankAccount` declarations for `b1` and `b2`:

```
b1.withdraw("MattW", 200.00);
b2.deposit("DannyB", 35.68);
```

STATIC METHODS

Static Methods vs. Instance Methods The methods discussed in the preceding sections—constructors, accessors, and mutators—all operate on individual objects of a class. They are called *instance methods*. A method that performs an operation for the entire class, not its individual objects, is called a *static method* (sometimes called a *class method*).

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;
    }
}
```

```

/** Write contents of List a.
 * @param a the list
 */
public static void writeList(List<Integer> a)
{
    System.out.println("List is : " + a);
}

public static void main(String[] args)
{
    List<Integer> list = getList();
    writeList(list);
}
}

```

NOTE

1. The calls to `writeList(list)` and `getList()` do not need to be preceded by `GetListTest` plus a dot because `main` is not a client program: It is in the same class as `getList` and `writeList`.
2. If you omit the keyword `static` from the `getList` or `writeList` header, you get an error message like the following:

```

Can't make static reference to method getList()
in class GetListTest

```

The compiler has recognized that there was no object variable preceding the method call, which means that the methods were static and should have been declared as such.

Method Overloading

Overloaded methods are two or more methods in the same class that have the same name but different parameter lists. For example,

```

public class DoOperations
{
    public int product(int n) { return n * n; }
    public double product(double x) { return x * x; }
    public double product(int x, int y) { return x * y; }
    ...
}

```

The compiler figures out which method to call by examining the method's *signature*. The signature of a method consists of the method's name and a list of the parameter types. Thus, the signatures of the overloaded `product` methods are

```

product(int)
product(double)
product(int, int)

```

Note that for overloading purposes, the return type of the method is irrelevant. You can't have two methods with identical signatures but different return types. The compiler will complain that the method call is ambiguous.

Having more than one constructor in the same class is an example of overloading. Overloaded constructors provide a choice of ways to initialize objects of the class.