Classes and Objects

CHAPTER 2

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.

All object-oriented programming languages have a way to represent an object as a variable in a program. In Java, a variable that represents an object is called an *object reference*.

CLASSES

A *class* is a software blueprint for implementing objects of a given type. An object is a single *instance* of the class. In a program there will often be several different instances of a given class type.

The current state of a given object is maintained in its *data fields* or *instance variables*, provided by the class. The *methods* of the class provide both the behaviors exhibited by the object and the operations that manipulate the object. Combining an object's data and methods into a single unit called a class is known as *encapsulation*.

Here is the framework for a simple bank account class:

```
public class BankAccount
{
    private String password;
    private double balance;
    public static final double OVERDRAWN_PENALTY = 20.00;

  //constructors
    /** Default constructor.
     *  Constructs bank account with default values. */
    public BankAccount()
    { /* implementation code */ }

    /** Constructs bank account with specified password and balance. */
    public BankAccount(String acctPassword, double acctBalance)
    { /* implementation code */ }

    //accessor
    /** @return balance of this account */
    public double getBalance()
    { /* implementation code */ }

    //mutators
    /** Deposits amount in bank account with given password.
     *  @param acctPassword the password of this bank account
     *  @param amount the amount to be deposited
   */
    public void deposit(String acctPassword, double amount)
    { /* implementation code */ }

    /** 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)
    { /* implementation code */ }
}
```

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.

METHODS

Headers

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

```
public      void     withdraw (String password, double amount)
```

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.

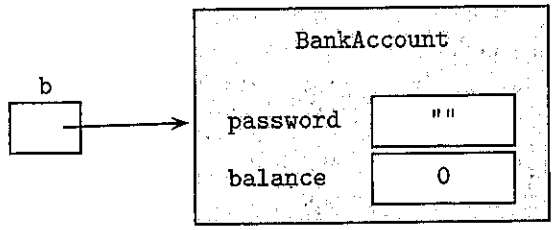
1. The *default constructor* has no arguments. It provides reasonable initial values for an object. Here is its implementation:

```
/** 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:



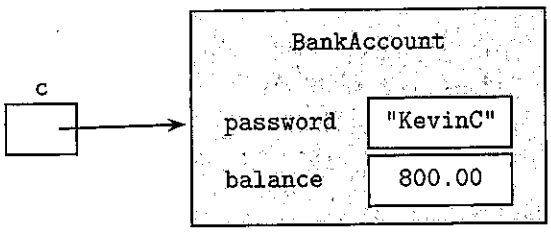
2. The constructor with parameters sets the instance variables of a BankAccount object to the values of those parameters.

   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.

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.
}
```

Suppose a client class has these lines of code:

```
Person p = new Person("Dan", 10);
p.printPerson();
```

The statement

```
System.out.println(this);
```

in the `printPerson` method means "print the current `Person` object." The output should be: `Dan 10`. Note that `System.out.println` invokes the `toString` method of the `Person` class.

Example 3

The `deposit` method of the `BankAccount` class can refer to `balance` as follows:

```
public void deposit(String acctPassword, double amount)
{
    this.balance += amount;
}
```

The use of `this` is unnecessary in the above example.

Example 4

Consider a rational number class called `Rational`, which has two private instance variables:

```
private int num;          //numerator
private int denom;        //denominator
```

Now consider a constructor for the `Rational` class:

```
public Rational(int num, int denom)
{
    this.num = num;
    this.denom = denom;
}
```

It is definitely *not* a good idea to use the same name for the explicit parameters and the private instance variables. But if you do, you can avoid errors by referring to `this.num` and `this.denom` for the current object that is being constructed. (This particular use of `this` will not be tested on the exam.)

REFERENCES

Reference vs. Primitive Data Types

All of the numerical data types, like `double` and `int`, as well as types `char` and `boolean`, are *primitive* data types. All objects are *reference* data types. The difference lies in the way they are stored.

Consider the statements

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

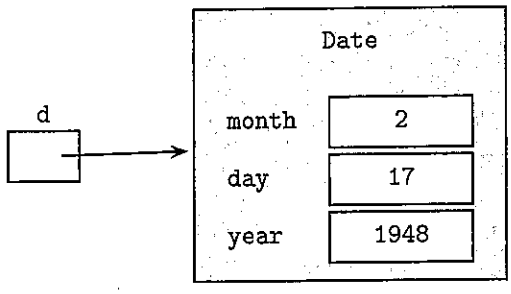
The variables num1 and num2 can be thought of as memory slots, labeled num1 and num2, respectively:

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);
```

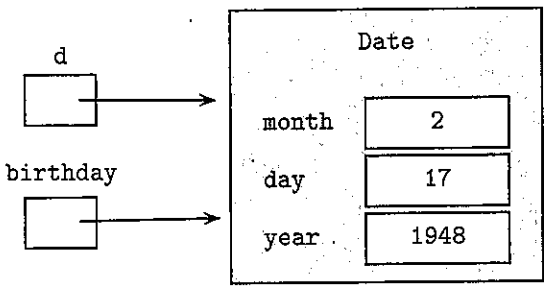
This declaration creates a reference variable d that refers to a Date object. The value of d is the address in memory of that object:



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.

The Null Reference

The declaration

```
BankAccount b;
```

defines a reference `b` that is uninitialized. (To construct the object that `b` refers to requires the `new` operator and a `BankAccount` constructor.) An uninitialized object variable is called a *null reference* or *null pointer*. You can test whether a variable refers to an object or is uninitialized by using the keyword `null`:

```
if (b == null)
```

If a reference is not null, it can be set to null with the statement

```
b = null;
```

An attempt to invoke an instance method with a null reference may cause your program to terminate with a `NullPointerException`. For example,

```
public class PersonalFinances
{
    BankAccount b;                      //b is a null reference
        ...
    b.withdraw(acctPassword, amt);      //throws a NullPointerException
        ...                             //if b not constructed with new
```

NOTE

If you fail to initialize a local variable in a method before you use it, you will get a compile-time error. If you make the same mistake with an instance variable of a class, the compiler provides reasonable default values for primitive variables (0 for numbers, `false` for booleans), and the code may run without error. However, if you don't initialize *reference* instance variables in a class, as in the above example, the compiler will set them to `null`. Any method call for an object of the class that tries to access the null reference will cause a run-time error: The program will terminate with a `NullPointerException`.

Do not make a method call with an object whose value is `null`.

Method Parameters

FORMAL VS. ACTUAL PARAMETERS

The header of a method defines the *parameters* of that method. For example, consider the `withdraw` method of the `BankAccount` class:

```
public class BankAccount
{   ...
    public void withdraw(String acctPassword, double amount)
        ...
```

This method has two explicit parameters, `acctPassword` and `amount`. These are *dummy* or *formal parameters*. Think of them as placeholders for the pair of *actual parameters* or *arguments* that will be supplied by a particular method call in a client program. For example,

```
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

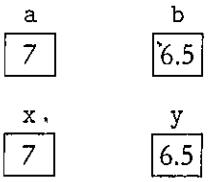
```
7  6.5
```

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

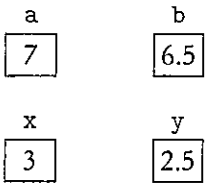
This can be understood by picturing the state of the memory slots during execution of the program.

Just before the `foo(a, b)` method call:

At the time of the `foo(a, b)` method call:



Just before exiting the method: Note that the values of x and y have been changed.



After exiting the method: Note that the memory slots for x and y have been reclaimed. The values of a and b remain unchanged.

PASSING OBJECTS AS PARAMETERS

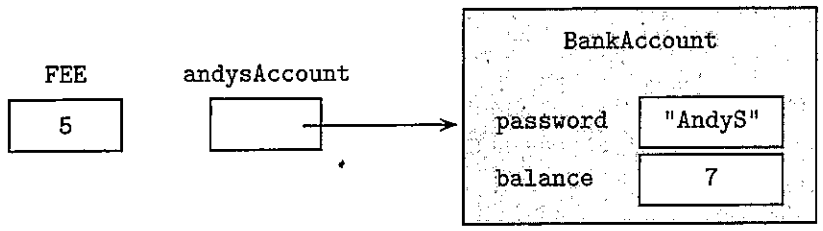
In Java both primitive types and object references are passed by value. When an object's reference is a parameter, the same mechanism of copying into local memory is used. The key difference is that the *address* (reference) is copied, not the values of the individual instance variables. As with primitive types, changes made to the parameters will not change the values of the matching arguments. What this means in practice is that it is not possible for a method to replace an object with another one—you can't change the reference that was passed. It is, however, possible to change the state of the object to which the parameter refers through methods that act on the object.

Example 1

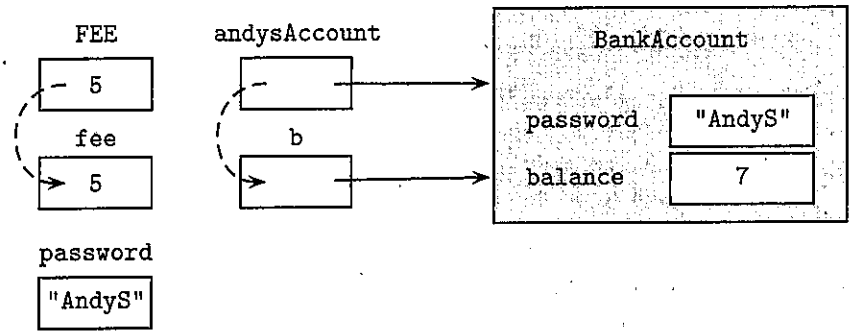
A method that changes the state of an object.

```
/** Subtracts fee from balance in b if current balance too low. */
public static void chargeFee(BankAccount b, String password,
        double fee)
{
    final double MIN_BALANCE = 10.00;
    if (b.getBalance() < MIN_BALANCE)
        b.withdraw(password, fee);
}

public static void main(String[] args)
{
    final double FEE = 5.00;
    BankAccount andysAccount = new BankAccount("AndyS", 7.00);
    chargeFee(andysAccount, "AndyS", FEE);
        ...
}
```

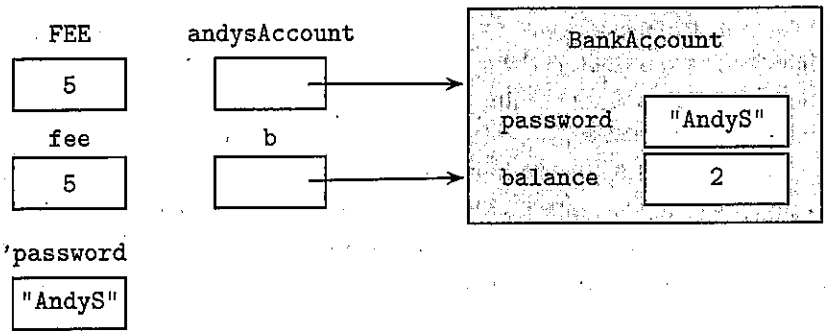
Here are the memory slots before the chargeFee method call:



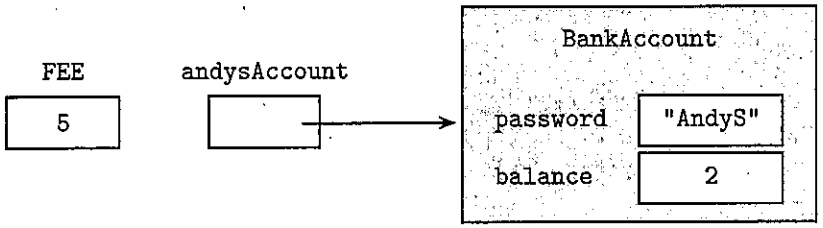
At the time of the chargeFee method call, copies of the matching parameters are made:



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.

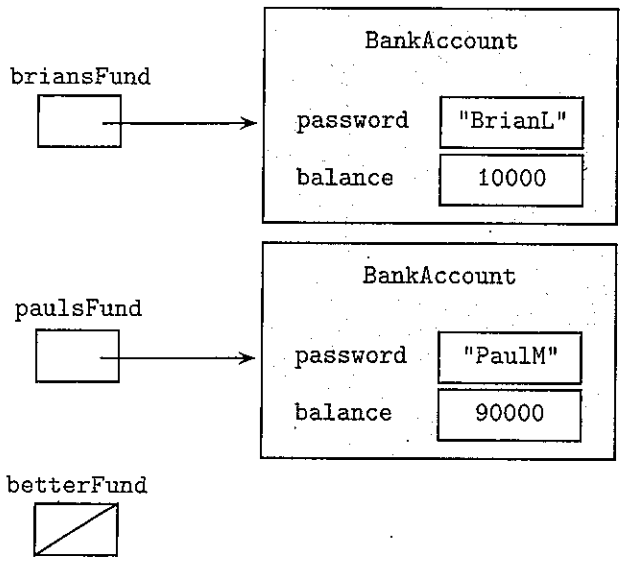
Example 2

A chooseBestAccount method attempts—erroneously—to set its betterFund parameter to the BankAccount with the higher balance:

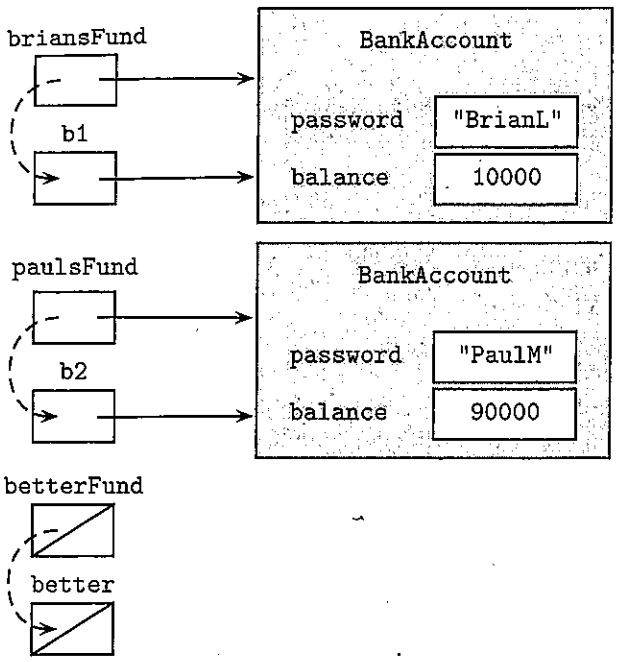
```
public static void chooseBestAccount(BankAccount better,
            BankAccount b1, BankAccount b2)
{
    if (b1.getBalance() > b2.getBalance())
        better = b1;
    else
        better = b2;
}

public static void main(String[] args)
{
    BankAccount briansFund = new BankAccount("BrianL", 10000);
    BankAccount paulsFund = new BankAccount("PaulM", 90000);
    BankAccount betterFund = null;

    chooseBestAccount(betterFund, briansFund, paulsFund);
        ...
}
```

The intent is that betterFund will be a reference to the paulsFund object after execution of the chooseBestAccount statement. A look at the memory slots illustrates why this fails.

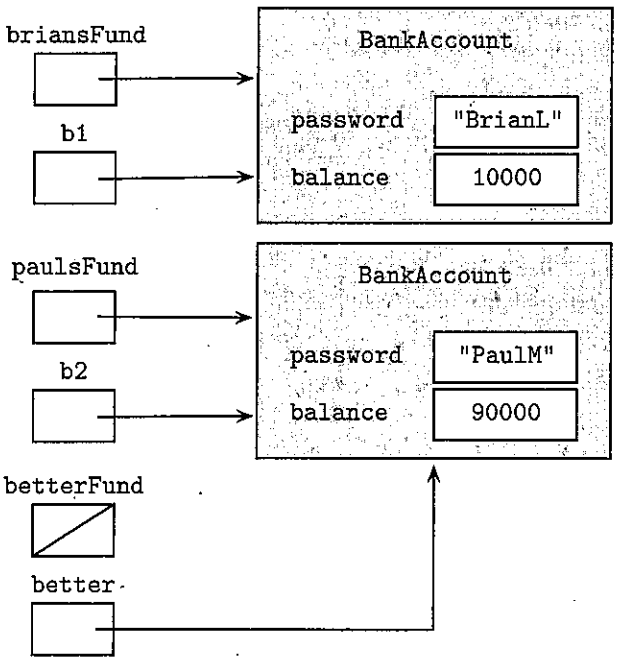
Before the chooseBestAccount method call:



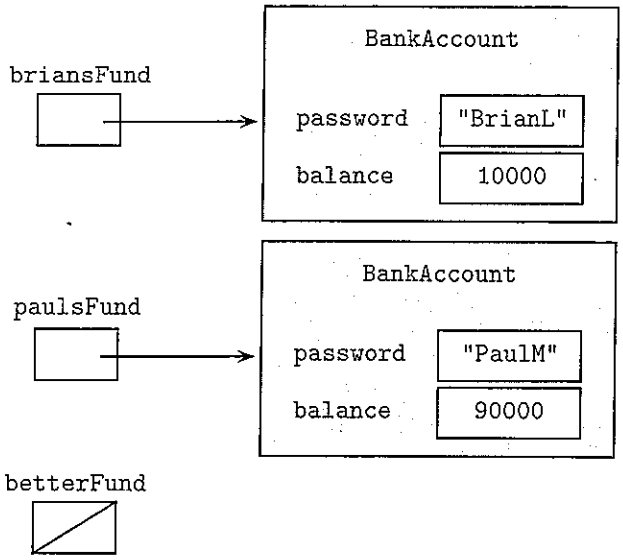
At the time of the chooseBestAccount method call: Copies of the matching references are made.



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.



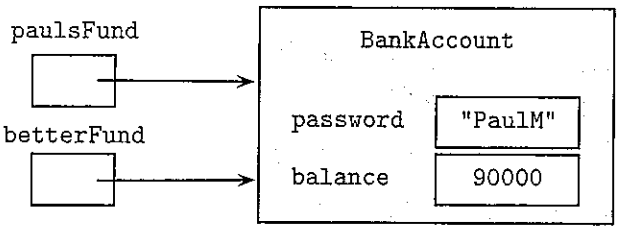
Note that the `betterFund` reference continues to be `null`, contrary to the programmer's intent.

The way to fix the problem is to modify the method so that it returns the better account. Returning an object from a method means that you are returning the address of the object.

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

public static void main(String[] args)
{
    BankAccount briansFund = new BankAccount("BrianL", 10000);
    BankAccount paulsFund = new BankAccount("PaulM", 90000);
    BankAccount betterFund = chooseBestAccount(briansFund, paulsFund);
        ...
}
```

NOTE

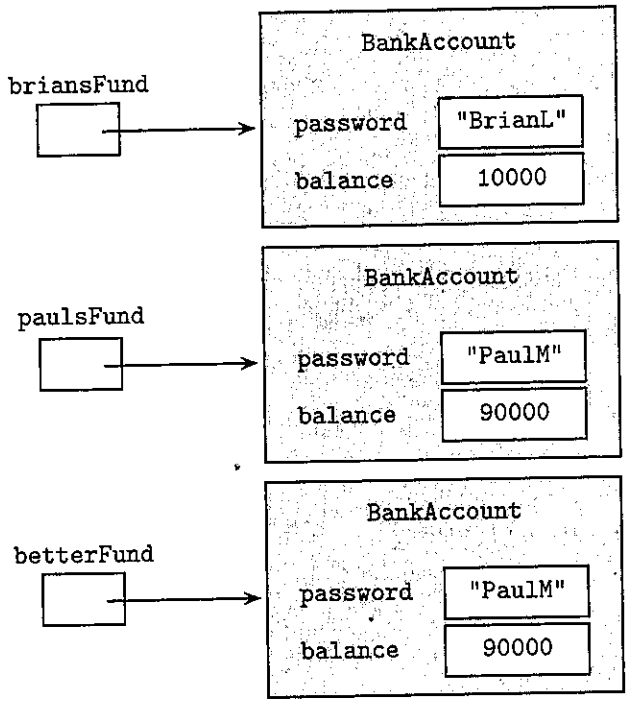
The effect of this is to create the `betterFund` reference, which refers to the same object as `paulsFund`:



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.

MULTIPLE-CHOICE QUESTIONS ON CLASSES AND OBJECTS

Questions 1–3 refer to the `Time` class declared below:

```
public class Time
{
    private int hrs;
    private int mins;
    private int secs;

    public Time()
    { /* implementation not shown */ }

    public Time(int h, int m, int s)
    { /* implementation not shown */ }

    /** Resets time to hrs = h, mins = m, secs = s. */
    public void resetTime(int h, int m, int s)
    { /* implementation not shown */ }

    /** Advances time by one second. */
    public void increment()
    { /* implementation not shown */ }

    /** @return true if this time equals t, false otherwise */
    public boolean equals(Time t)
    { /* implementation not shown */ }

    /** @return true if this time is earlier than t, false otherwise */
    public boolean lessThan(Time t)
    { /* implementation not shown */ }

    /** @return a String with the time in the form hrs:mins:secs */
    public String toString()
    { /* implementation not shown */ }
}
```

1. Which of the following is a *false* statement about the methods?
   (A) `equals`, `lessThan`, and `toString` are all accessor methods.
   (B) `increment` is a mutator method.
   (C) `Time()` is the default constructor.
   (D) The `Time` class has three constructors.
   (E) There are no static methods in this class.

2. Which of the following represents correct *implementation code* for the constructor with parameters?

(A)
```
hrs = 0;
mins = 0;
secs = 0;
```

(B)
```
hrs = h;
mins = m;
secs = s;
```

(C)
```
resetTime(hrs, mins, secs);
```

(D)
```
h = hrs;
m = mins;
s = secs;
```

(E)
```
Time = new Time(h, m, s);
```

3. A client class has a `display` method that writes the time represented by its parameter:

```
/** Outputs time t in the form hrs:mins:secs.
 *  @param t the time
 */
public void display (Time t)
{
    /* method body */
}
```

Which of the following are correct replacements for `/* method body */`?

I
```
Time T = new Time(h, m, s);
System.out.println(T);
```

II
```
System.out.println(t.hrs + ":" + t.mins + ":" + t.secs);
```

III
```
System.out.println(t);
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

4. Which statement about parameters is *false*?
(A) The scope of parameters is the method in which they are defined.
(B) Static methods have no implicit parameter `this`.
(C) Two overloaded methods in the same class must have parameters with different names.
(D) All parameters in Java are passed by value.
(E) Two different constructors in a given class can have the same number of parameters.

Questions 5–11 refer to the following `Date` class declaration:

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

    public Date()                              //default constructor
    {
        ...
    }

    public Date(int mo, int da, int yr)    //constructor
    {
        ...
    }

    public int month()       //returns month of Date
    {
        ...
    }

    public int day()         //returns day of Date
    {
        ...
    }

    public int year()       //returns year of Date
    {
        ...
    }

    //Returns String representation of Date as "m/d/y", e.g. 4/18/1985.
    public String toString()
    {
        ...
    }
}
```

5. Which of the following correctly constructs a `Date` object in a client class?

   (A) `Date d = new (2, 13, 1947);`

   (B) `Date d = new Date(2, 13, 1947);`

   (C)
   ```
   Date d;
   d = new (2, 13, 1947);
   ```

   (D)
   ```
   Date d;
   d = Date(2, 13, 1947);
   ```

   (E) `Date d = Date(2, 13, 1947);`

6. Which of the following will cause an error message?

I
```
Date d1 = new Date(8, 2, 1947);
Date d2 = d1;
```

II
```
Date d1 = null;
Date d2 = d1;
```

III
```
Date d = null;
int x = d.year();
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

7. A client program creates a `Date` object as follows:

```
Date d = new Date(1, 13, 2002);
```

Which of the following subsequent code segments will cause an error?

(A) `String s = d.toString();`
(B) `int x = d.day();`
(C) `Date e = d;`
(D) `Date e = new Date(1, 13, 2002);`
(E) `int y = d.year;`

8. Consider the implementation of a `write()` method that is added to the `Date` class:

```
/** Write the date in the form m/d/y, for example 2/17/1948. */
public void write()
{
    /* implementation code */
}
```

Which of the following could be used as `/* implementation code */`?

I
```
System.out.println(month + "/" + day + "/" + year);
```

II
```
System.out.println(month() + "/" + day() + "/" + year());
```

III
```
System.out.println(this);
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

9. Here is a client program that uses `Date` objects:

```
public class BirthdayStuff
{
    public static Date findBirthdate()
    {
        /* code to get birthDate */
        return birthDate;
    }

    public static void main(String[] args)
    {
        Date d = findBirthdate();
            ...
    }
}
```

Which of the following is a correct replacement for
`/* code to get birthDate */`?

I
```
System.out.println("Enter birthdate: mo, day, yr: ");
int m = IO.readInt();                    //read user input
int d = IO.readInt();                    //read user input
int y = IO.readInt();                    //read user input
Date birthDate = new Date(m, d, y);
```

II
```
System.out.println("Enter birthdate: mo, day, yr: ");
int birthDate.month() = IO.readInt();       //read user input
int birthDate.day() = IO.readInt();         //read user input
int birthDate.year() = IO.readInt();        //read user input
Date birthDate = new Date(birthDate.month(), birthDate.day(),
    birthDate.year());
```

III
```
System.out.println("Enter birthdate: mo, day, yr: ");
int birthDate.month = IO.readInt();       //read user input
int birthDate.day = IO.readInt();         //read user input
int birthDate.year = IO.readInt();        //read user input
Date birthDate = new Date(birthDate.month, birthDate.day,
    birthDate.year);
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I and III only

10. A method in a client program for the Date class has this declaration:

```
Date d1 = new Date(mo, da, yr);
```

where mo, da, and yr are previously defined integer variables. The same method now creates a second Date object d2 that is an exact copy of the object d1 refers to. Which of the following code segments will *not* do this correctly?

I
```
Date d2 = d1;
```

II
```
Date d2 = new Date(mo, da, yr);
```

III
```
Date d2 = new Date(d1.month(), d1.day(), d1.year());
```

(A) I only
(B) II only
(C) III only
(D) II and III only
(E) I, II, and III

11. The Date class is modified by adding the following mutator method:

```
public void addYears(int n)    //add n years to date
```

Here is part of a poorly coded client program that uses the Date class:

```
public static void addCentury(Date recent, Date old)
{
    old.addYears(100);
    recent = old;
}

public static void main(String[] args)
{
    Date oldDate = new Date(1, 13, 1900);
    Date recentDate = null;
    addCentury(recentDate, oldDate);
        ...
}
```

Which will be true after executing this code?
(A) A NullPointerException is thrown.
(B) The oldDate object remains unchanged.
(C) recentDate is a null reference.
(D) recentDate refers to the same object as oldDate.
(E) recentDate refers to a separate object whose contents are the same as those of oldDate.

12. Here are the private instance variables for a Frog object:

```
public class Frog
{
    private String species;
    private int age;
    private double weight;
    private Position position;      //position (x,y) in pond
    private boolean amAlive;
        ...
```

Which of the following methods in the Frog class is the best candidate for being a static method?

(A) `swim` `//frog swims to new position in pond`

(B) `getPondTemperature` `//returns temperature of pond`

(C) `eat` `//frog eats and gains weight`

(D) `getWeight` `//returns weight of frog`

(E) `die` `//frog dies with some probability based`
`//on frog's age and pond temperature`

13. What output will be produced by this program?

```
public class Mystery
{
    public static void strangeMethod(int x, int y)
    {
        x += y;
        y *= x;
        System.out.println(x + " " + y);
    }

    public static void main(String[] args)
    {
        int a = 6, b = 3;
        strangeMethod(a, b);
        System.out.println(a + " " + b);
    }
}
```

(A) 36
9

(B) 3 6
9

(C) 9 27
9 27

(D) 6 3
9 27

(E) 9 27
6 3

Questions 14–17 refer to the following definition of the `Rational` class:

```
public class Rational
{
    private int numerator;
    private int denominator;

    /** default constructor */
    Rational()
    { /* implementation not shown */ }

    /** Constructs a Rational with numerator n and
     *  denominator 1. */
    Rational(int n)
    { /* implementation not shown */ }

    /** Constructs a Rational with specified numerator and
     *  denominator. */
    Rational(int numer, int denom)
    { /* implementation not shown */ }

    /** @return numerator */
    int numerator()
    { /* implementation not shown */ }

    /** @return denominator */
    int denominator()
    { /* implementation not shown */ }

    /** Returns (this + r). Leaves this unchanged.
     *  @return this rational number plus r
     *  @param r a rational number to be added to this Rational
     */
    public Rational plus(Rational r)
    { /* implementation not shown */ }

    //Similarly for times, minus, divide
          ...
    /** Ensures denominator > 0. */
    private void fixSigns()
    { /* implementation not shown */ }

    /** Ensures lowest terms. */
    private void reduce()
    { /* implementation not shown */ }
}
```

14. The method `reduce()` is not a public method because
    (A) methods whose return type is `void` cannot be public.
    (B) methods that change `this` cannot be public.
    (C) the `reduce()` method is not intended for use by clients of the `Rational` class.
    (D) the `reduce()` method is intended for use only by clients of the `Rational` class.
    (E) the `reduce()` method uses only the private data fields of the `Rational` class.

15. The constructors in the `Rational` class allow initialization of `Rational` objects in several different ways. Which of the following will cause an error?

(A) `Rational r1 = new Rational();`
(B) `Rational r2 = r1;`
(C) `Rational r3 = new Rational(2,-3);`
(D) `Rational r4 = new Rational(3.5);`
(E) `Rational r5 = new Rational(10);`

16. Here is the implementation code for the `plus` method:

```
/** Returns (this + r). Leaves this unchanged.
 *  @return this rational number plus r
 *  @param r a rational number to be added to this Rational
 */
public Rational plus(Rational r)
{
    fixSigns();
    r.fixSigns();
    int denom = denominator * r.denominator;
    int numer = numerator * r.denominator
                + r.numerator * denominator;
    /* more code */
}
```

Which of the following is a correct replacement for `/* more code */`?

(A)
```
Rational rat(numer, denom);
rat.reduce();
return rat;
```

(B)
```
return new Rational(numer, denom);
```

(C)
```
reduce();
Rational rat = new Rational(numer, denom);
return rat;
```

(D)
```
Rational rat = new Rational(numer, denom);
Rational.reduce();
return rat;
```

(E)
```
Rational rat = new Rational(numer, denom);
rat.reduce();
return rat;
```

17. Assume these declarations:

```
Rational a = new Rational();
Rational r = new Rational(numer, denom);
int n = value;
//numer, denom, and value are valid integer values
```

Which of the following will cause a compile-time error?

(A) `r = a.plus(r);`
(B) `a = r.plus(new Rational(n));`
(C) `r = r.plus(r);`
(D) `a = n.plus(r);`
(E) `r = r.plus(new Rational(n));`

Questions 18–20 refer to the Temperature class shown below:

```
public class Temperature
{
    private String scale;  //valid values are "F" or "C"
    private double degrees;

    /** constructor with specified degrees and scale */
    public Temperature(double tempDegrees, String tempScale)
    { /* implementation not shown */ }

    /** Mutator. Converts this Temperature to degrees Fahrenheit.
     *  Precondition: Temperature is a valid temperature
     *                in degrees Celsius.
     *  @return this temperature in degrees Fahrenheit
     */
    public Temperature toFahrenheit()
    { /* implementation not shown */ }

    /** Mutator. Converts this Temperature to degrees Celsius.
     *  Precondition: Temperature is a valid temperature
     *                in degrees Fahrenheit.
     *  @return this temperature in degrees Celsius
     */
    public Temperature toCelsius()
    { /* implementation not shown */ }

    /** Mutator.
     *  @param amt the number of degrees to raise this temperature
     *  @return this temperature raised by amt degrees
     */
    public Temperature raise(double amt)
    { /* implementation not shown */ }

    /** Mutator.
     *  @param amt the number of degrees to lower this temperature
     *  @return this temperature lowered by amt degrees
     */
    public Temperature lower(double amt)
    { /* implementation not shown */ }

    /** @param tempDegrees the number of degrees
     *  @param tempScale the temperature scale
     *  @return true if tempDegrees is a valid temperature
     *  in the given temperature scale, false otherwise
     */
    public static boolean isValidTemp(double tempDegrees,
                                      String tempScale)
    { /* implementation not shown */ }

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

A client method contains this code segment:

```
Temperature t1 = new Temperature(40, "C");
Temperature t2 = t1;
Temperature t3 = t2.lower(20);
Temperature t4 = t1.toFahrenheit();
```

Which statement is *true* following execution of this segment?

(A) `t1`, `t2`, `t3`, and `t4` all represent the identical temperature, in degrees Celsius.
(B) `t1`, `t2`, `t3`, and `t4` all represent the identical temperature, in degrees Fahrenheit.
(C) `t4` represents a Fahrenheit temperature, while `t1`, `t2`, and `t3` all represent degrees Celsius.
(D) `t1` and `t2` refer to the same `Temperature` object; `t3` refers to a `Temperature` object that is 20 degrees lower than `t1` and `t2`, while `t4` refers to an object that is `t1` converted to Fahrenheit.
(E) A `NullPointerException` was thrown.

Consider the following code:

```
public class TempTest
{
    public static void main(String[] args)
    {
        System.out.println("Enter temperature scale: ");
        String tempScale = IO.readString();     //read user input
        System.out.println("Enter number of degrees: ");
        double tempDegrees = IO.readDouble();   //read user input
        /* code to construct a valid temperature from user input */
    }
}
```

Which is a correct replacement for */* code to construct... */*?

I
```
Temperature t = new Temperature(tempDegrees, tempScale);
if (!t.isValidTemp(tempDegrees,tempScale))
    /* error message and exit program */
```

II
```
if (isValidTemp(tempDegrees,tempScale))
    Temperature t = new Temperature(tempDegrees, tempScale);
else
    /* error message and exit program */
```

III
```
if (Temperature.isValidTemp(tempDegrees,tempScale))
    Temperature t = new Temperature(tempDegrees, tempScale);
else
    /* error message and exit program */
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I and III only

20. The formula to convert degrees Celsius C to Fahrenheit F is

$$F = 1.8C + 32$$

For example, 30° C is equivalent to 86° F.

An `inFahrenheit()` accessor method is added to the `Temperature` class. Here is its implementation:

```
/** Precondition:  The temperature is a valid temperature
 *                 in degrees Celsius.
 *  Postcondition:
 *   - An equivalent temperature in degrees Fahrenheit has been
 *     returned.
 *   - Original temperature remains unchanged.
 * @return an equivalent temperature in degrees Fahrenheit
 */
public Temperature inFahrenheit()
{
    Temperature result;
    /* more code */
    return result;
}
```

Which of the following correctly replaces `/* more code */` so that the postcondition is achieved?

I
```
result = new Temperature(degrees * 1.8 + 32, "F");
```

II
```
result = new Temperature(degrees * 1.8, "F");
result = result.raise(32);
```

III
```
degrees *= 1.8;
this = this.raise(32);
result = new Temperature(degrees, "F");
```

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

21. Consider this program:

```
public class CountStuff
{
    public static void doSomething()
    {
        int count = 0;
            ...
        //code to do something - no screen output produced
        count++;
    }

    public static void main(String[] args)
    {
        int count = 0;
        System.out.println("How many iterations?");
        int n = IO.readInt();     //read user input
        for (int  i = 1; i <= n; i++)
        {
            doSomething();
            System.out.println(count);
        }
    }
}
```

If the input value for `n` is `3`, what screen output will this program subsequently produce?

(A)
```
0
0
0
```

(B)
```
1
2
3
```

(C)
```
3
3
3
```

(D)
```
?
?
?
```
where `?` is some undefined value.

(E) No output will be produced.

22. This question refers to the following class:

```
public class IntObject
{
    private int num;

    public IntObject()        //default constructor
    { num = 0; }

    public IntObject(int n)   //constructor
    { num = n; }

    public void increment()   //increment by 1
    { num++; }
}
```

Here is a client program that uses this class:

```
public class IntObjectTest
{
    public static IntObject someMethod(IntObject obj)
    {
        IntObject ans = obj;
        ans.increment();
        return ans;
    }

    public static void main(String[] args)
    {
        IntObject x = new IntObject(2);
        IntObject y = new IntObject(7);
        IntObject a = y;
        x = someMethod(y);
        a = someMethod(x);
    }
}
```

Just before exiting this program, what are the object values of x, y, and a, respec
tively?

(A) 9, 9, 9
(B) 2, 9, 9
(C) 2, 8, 9
(D) 3, 8, 9
(E) 7, 8, 9

23. Consider the following program:

```
public class Tester
{
    public void someMethod(int a, int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

public class TesterMain
{
    public static void main(String[] args)
    {
        int x = 6, y = 8;
        Tester tester = new Tester();
        tester.someMethod(x, y);
    }
}
```

Just before the end of execution of this program, what are the values of `x`, `y`, and `temp`, respectively?

(A) 6, 8, 6
(B) 8, 6, 6
(C) 6, 8, ?, where ? means undefined
(D) 8, 6, ?, where ? means undefined
(E) 8, 6, 8

ANSWER KEY

1. D	9. A	17. D
2. B	10. A	18. B
3. C	11. C	19. C
4. C	12. B	20. D
5. B	13. E	21. A
6. C	14. C	22. A
7. E	15. D	23. C
8. E	16. E	

ANSWERS EXPLAINED

1. (D) There are just two constructors. Constructors are recognizable by having the same name as the class, and no return type.
2. (B) Each of the private instance variables should be assigned the value of the matching parameter. Choice B is the only choice that does this. Choice D confuses the order of the assignment statements. Choice A gives the code for the *default* constructor, ignoring the parameters. Choice C would be correct if it were `resetTime(h, m, s)`. As written, it doesn't assign the parameter values `h`, `m`, and `s` to `hrs`, `mins`, and `secs`. Choice E is wrong because the keyword `new` should be used to create a new object, not to implement the constructor!
3. (C) Replacement III will automatically print time `t` in the required form since a `toString` method was defined for the `Time` class. Replacement I is wrong because it doesn't refer to the parameter, `t`, of the method. Replacement II is wrong because a client program may not access private data of the class.
4. (C) The parameter names can be the same—the *signatures* must be different. For example,

```
public void print(int x)        //prints x
public void print(double x)     //prints x
```

   The signatures (method name plus parameter types) here are `print(int)` and `print(double)`, respectively. The parameter name `x` is irrelevant. Choice A is true: All local variables and parameters go out of scope (are erased) when the method is exited. Choice B is true: Static methods apply to the whole class. Only instance methods have an implicit `this` parameter. Choice D is true even for object parameters: Their references are passed by value. Note that choice E is true because it's possible to have two different constructors with different signatures but the same number of parameters (e.g., one for an `int` argument and one for a `double`).
5. (B) Constructing an object requires the keyword `new` and a constructor of the `Date` class. Eliminate choices D and E since they omit `new`. The class name `Date` should appear on the right-hand side of the assignment statement, immediately following the keyword `new`. This eliminates choices A and C.

6. (C) Segment III will cause a `NullPointerException` to be thrown since `d` is a null reference. You cannot invoke a method for a null reference. Segment II has the effect of assigning `null` to both `d1` and `d2`—obscure but not incorrect. Segment I creates the object reference `d1` and then declares a second reference `d2` that refers to the same object as `d1`.

7. (E) A client program cannot access a private instance variable.

8. (E) All are correct. Since `write()` is a `Date` instance method, it is OK to use the private data members in its implementation code. Segment III prints `this`, the current `Date` object. This usage is correct since `write()` is part of the `Date` class. The `toString()` method guarantees that the date will be printed in the required format (see p. 175).

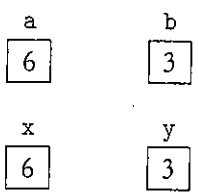
9. (A) The idea here is to read in three separate variables for month, day, and year and then to construct the required date using `new` and the `Date` class constructor with three parameters. Code segment II won't work because `month()`, `day()`, and `year()` are accessor methods that access existing values and may not be used to read new values into `bDate`. Segment III is wrong because it tries to access private instance variables from a client program.

10. (A) Segment I will not create a second object. It will simply cause `d2` to refer to the *same* object as `d1`, which is not what was required. The keyword `new` *must* be used to create a new object.

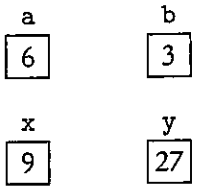
11. (C) When `recentDate` is declared in `main()`, its value is null. Recall that a method is not able to replace an object reference, so `recentDate` remains null. Note that the intent of the program is to change `recentDate` to refer to the updated `oldDate` object. The code, however, doesn't do this. Choice A is false: No methods are invoked with a null reference. Choice B is false because `addYears()` is a mutator method. Even though a method doesn't change the address of its object parameter, it can change the contents of the object, which is what happens here. Choices D and E are wrong because the `addCentury()` method cannot change the value of its `recentDate` argument.

12. (B) The method `getPondTemperature` is the only method that applies to more than one frog. It should therefore be static. All of the other methods relate directly to one particular `Frog` object. So `f.swim()`, `f.die()`, `f.getWeight()`, and `f.eat()` are all reasonable methods for a single instance `f` of a `Frog`. On the other hand, it doesn't make sense to say `f.getPondTemperature()`. It makes more sense to say `Frog.getPondTemperature()`, since the same value will apply to all frogs in the class.

13. (E) Here are the memory slots at the start of `strangeMethod(a, b)`:



Before exiting `strangeMethod(a, b)`:



Note that `9 27` is output before exiting. After exiting `strangeMethod(a, b)`, th memory slots are

The next step outputs `6 3`.

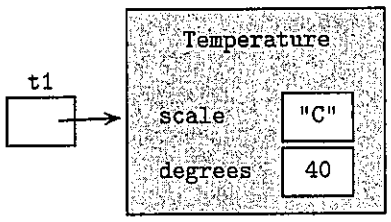
14. (C) The `reduce()` method will be used only in the implementation of the in stance methods of the `Rational` class.

15. (D) None of the constructors in the `Rational` class takes a real-valued param ter. Thus, the real-valued parameter in choice D will need to be converted to a integer. Since in general truncating a real value to an integer involves a loss precision, it is not done automatically—you have to do it explicitly with a cas Omitting the cast causes a compile-time error.

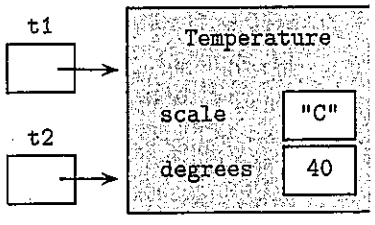
16. (E) A new `Rational` object must be created using the newly calculated `nume` and `denom`. Then it must be reduced before being returned. Choice A is wror because it doesn't correctly create the new object. Choice B returns a correctl constructed object, but one that has not been reduced. Choice C reduces th current object, `this`, instead of the new object, `rat`. Choice D is wrong becau it invokes `reduce()` for the `Rational` class instead of the specific `rat` object.

17. (D) The `plus` method of the `Rational` class can only be invoked by `Rationa` objects. Since `n` is an `int`, the statement in choice D will cause an error.

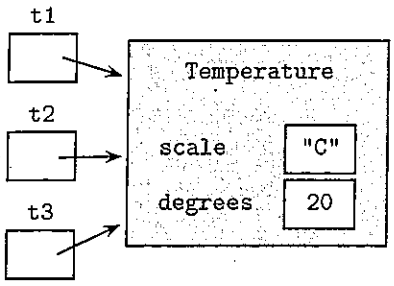
18. (B) This is an example of *aliasing*. The keyword `new` is used just once, whic means that just one object is constructed. Here are the memory slots after eac declaration:



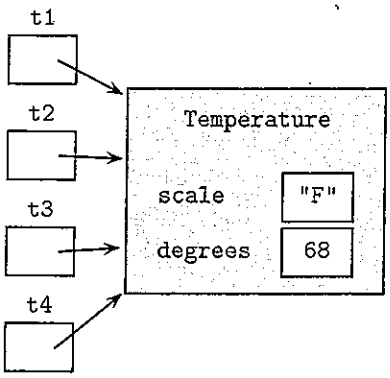
After declaration for `t1`



After declaration for `t2`



After declaration for `t3`



After declaration for `t4`

19. (C) Notice that `isValidTemp` is a static method for the `Temperature` class, which means that it cannot be invoked with a `Temperature` object. Thus, segment I is incorrect: `t.isValidTemp` is wrong. Segment II fails because `isValidTemp` is not a method of the `TempTest` class. It therefore must be invoked with its class name, which is what happens (correctly) in segment III: `Temperature.isValidTemp`.

20. (D) A new `Temperature` object must be constructed to prevent the current `Temperature` from being changed. Segment I, which applies the conversion formula directly to `degrees`, is the best way to do this. Segment II, while not the best algorithm, does work. The statement

```
result = result.raise(32);
```

has the effect of raising the `result` temperature by 32 degrees, and completing the conversion. Segment III fails because

```
degrees *= 1.8;
```

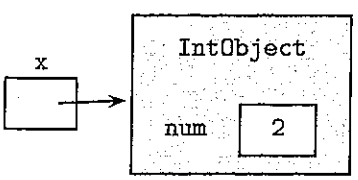
alters the `degrees` instance variable of the current object, as does

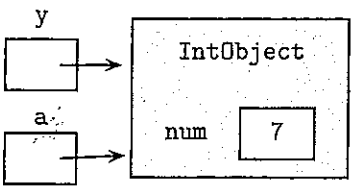
```
this = this.raise(32);
```

To be correct, these operations must be applied to the `result` object.

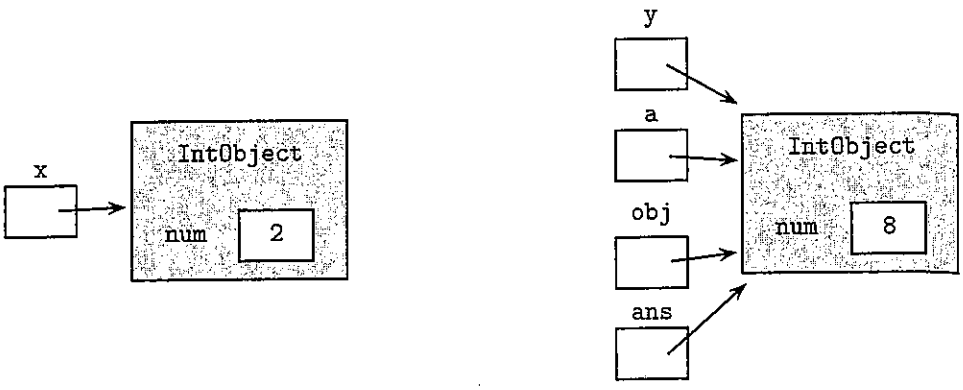
21. (A) This is a question about the scope of variables. The scope of the `count` variable that is declared in `main()` extends up to the closing brace of `main()`. In `doSomething()`, `count` is a local variable. After the method call in the `for` loop, the local variable `count` goes out of scope, and the value that's being printed is the value of the `count` in `main()`, which is unchanged from `0`.

22. (A) Here are the memory slots before the first `someMethod` call:





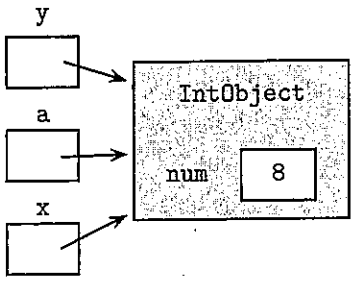
Just before exiting `x = someMethod(y)`:



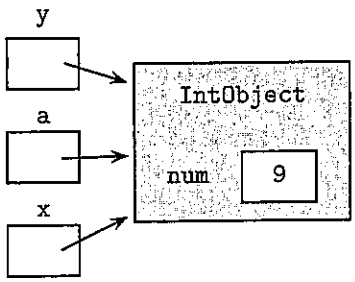
After exiting

```
x = someMethod(y);
```

x has been reassigned, so the object with `num = 2` has been recycled:



After exiting `a = someMethod(x)`:



23. (C) Recall that when primitive types are passed as parameters, copies are made of the actual arguments. All manipulations in the method are performed on the copies, and the arguments remain unchanged. Thus x and y retain their values of 6 and 8. The local variable `temp` goes out of scope as soon as `someMethod` is exited and is therefore undefined just before the end of execution of the program.