Class definition

Based on text of M. Smith: “Java, an object-oriented language”. McGraw Hill.

Consider the following class definition

```
class Account {
   private double theBalance = 0.0;
   private double theMinBalance = 0.0;

   public Account() {
      theBalance = theMinBalance = 0.0;
   }

   public double accountBalance() {
      return theBalance;
   }

   public double withdraw(final double money) {
      if (theBalance - money >= theMinBalance) {
         theBalance = theBalance - money;
         return money;
      }
      else {
         return 0.0;
      }
   } // withdraw

   public void deposit (final double money {
      theBalance = theBalance + money;
   }

   public void setMinBalance (final double money) {
      theMinBalance = money;
   }
} // account
```

The following is a Java application that tests class Account.

```
class TestAccount {
   public static void main (String args []) {
      Account mike;
      mike = new Account();
      Account corinna = new Account();
      double obtained;
      System.out.println("Mike's balance =" + 
mike.accountBalance());
      mike.deposit(100.0);
      System.out.println("Mike's balance =" + 
mike.accountBalance());
      obtained = mike.withdraw(20.0);
      System.out.println("Mike has withdrawn : " + 
obtained);
      System.out.println("Mike's balance =" + 
mike.accountBalance());
      corinna.deposit(50.0);
      System.out.println("Corinna's balance =" + 
corinna.accountBalance());
   } // main
} //TestAccount
```

Inheritance

Suppose we wish to create a new class, called AccountWithStatement, that has all the properties (methods and variables) of Account, plus additional methods and variables.

Additional method:

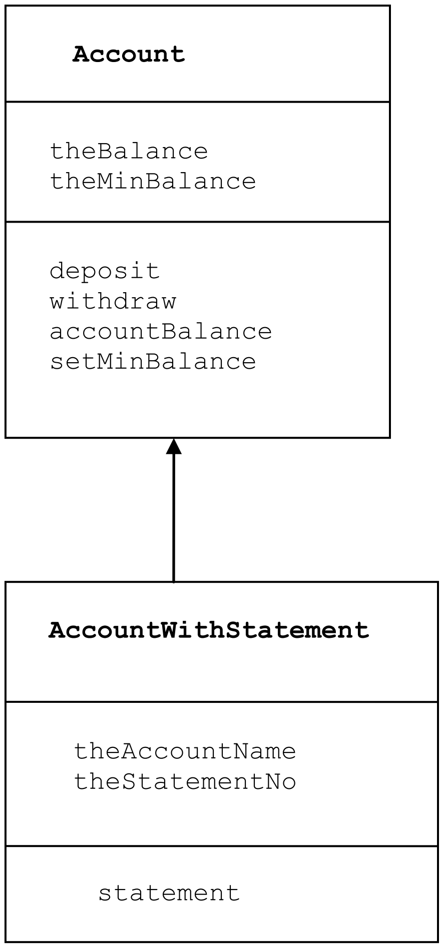
Statement: returns a string representing a mini-statement for the account

Additional variables:

theAccountName: holds the account holder’s name.
theStatementNo: unique number identifying the statement

The class Account is the super class, or base class. Class AccountWithStatement is the subclass or derived class.

Class diagram



Use keyword **extends** to indicate inheritance.

```
class AccountWithStatement extends Account {
   private String theAccountName;
   private long theStatementNo;

   public AccountWithStatement() { //first constructor
      theAccountName = "Anonymous";
      theStatementNo = 1;
   }

   public AccountWithStatement( final String name ) {
//second constructor
      theAccountName = name;
      theStatementNo = 1;
   }
   public String statement( ) {
     return "Statement # " + theStatementNo++ + " for " +
theAccountName + "\n" +
               " The balance of your account is : $" +
accountBalance( ) + "\n";
     }
} // AccountWithStatement
```

A signature of a method is its parameter list. Methods in the same visibility scope can have same names as long as their signatures are different.

    AccountWithStatement( )
    AccountWithStatement(final String name)

The subclass has (inherits) the implementations of the methods and variables defined in the super class (that is, we can use them without redefining them in the subclass.)

Here is a Java application to test the above class.

```
class TestAccountWithStatement {
   public static void main (String args []){
      AccountWithStatement mike = new
AccountWithStatement("Mike");

      double obtained;
      System.out.println("Mike's balance = " +
mike.accountBalance());
      System.out.println("Mike has deposit $100");
      mike.deposit(100.0);
      System.out.println("Mike's balance = " +
mike.accountBalance());
      obtained = mike.withdraw(20.0);
      System.out.println("Mike has withdrawn : " +
obtained);
      System.out.println(mike.statement());

      System.out.println("Mike has deposit $50");
      mike.deposit(50.0);
      System.out.println(mike.statement());
   }// main
}// class TestAccountWithStatement
```

Method overriding

Class A implements method M.
Class B extends class A.
In class B, we can implement M is a different way.

```
/* class RestrictedAccount is a subclass of class Account,
it inherits all variables and methods of class Account, it
may contain other variables and methods, it may also
override a method of class Account
*/
class RestrictedAccount extends Account {
   private static final int MAXWITHDRAWALS = 3;
   private int theNoWithdrawToGo = MAXWITHDRAWALS;

   public void reset( ) {
     theNoWithdrawToGo = MAXWITHDRAWALS;
   }

   public double withdraw(final double amount) {
//overriding the super class method

      if (theNoWithdrawToGo > 0){
         theNoWithdrawToGo--;
         return super.withdraw(amount); // apply the super
class’ implementation
      }
      else {
         return 0.0;
      }
   } // withdraw
} // RestrictedAccount
```

// a static variable is a class variable, there is only one for each class
// non-static variable: one for each object.

In the above, the default constructor is created to call the constructor in the super class’ method.

Here is an application to test class RestrictedAccount

```
class TestRestrictedAccount {
   public static void main (String args []) {
      RestrictedAccount mike;
      mike = new RestrictedAccount();

      Account corinna = new Account();

      double obtained;

      mike.deposit(100.0);
      System.out.println("Mike's balance =  " +
mike.accountBalance() );

     for (int i = 1; i <= 4; i++){
          obtained = mike.withdraw(20.0);
               System.out.println("Mike has withdrawn " +
obtained );
      } //for
      System.out.println("Mike's balance = " +
mike.accountBalance());
      System.out.println();

      corinna.deposit(100.0);
      System.out.println("Corinna's balance = " +
corinna.accountBalance());

      for (int i = 1; i <= 4; i++) {
               obtained = corinna.withdraw(20.0);
               System.out.println("Corinna has withdrawn "
+ obtained );
      } // for
      System.out.println("Corinna's balance = " +
corinna.accountBalance());
      System.out.println();
   }// main
}  // TestRestrictedAccount
```

Abstract methods and classes

Method M in class C is abstract if it is declared but not implemented in C. Subclasses of C are expected to implement M.

A class is abstract if it contains an abstract method. We cannot make objects of an abstract class (why?).

Here is an example of an abstract class.

```
abstract class AbstractAccount {
   private double theBalance = 0.0;

   public double accountBalance(){
      return theBalance;
   }

   public double withdraw(final double money){
      if (theBalance - money >= 0.0){
         theBalance = theBalance - money;
         return money;
      }
      else {
         return 0.0;
      }
   }//withdraw

   public void deposit (final double money){
      theBalance = theBalance + money;
   }

   abstract public String statement( ); //abstract method,
to be implemented in subclass
}  // AbtractAccount
```

Here is an example of a class that is a subclass of an abstract class.

```
class NormalAccount extends AbstractAccount {
     private String theName = "";

     NormalAccount( String name ){
          theName = name;
     }
     public String statement( ) { // implementation of the
abstract method of super class
          return theName + " balance is " + accountBalance(
);
     }
} //NormalAccount
```

Here is an application to test class NormalAccount

```
class TestNormalAccount {
     public static void main( String args[ ] ) {
          NormalAccount mike = new NormalAccount(" Mike's
");

          mike.deposit(100.0);
          System.out.println("Mike's balance = " +
mike.accountBalance());
          double obtained = mike.withdraw(20.0);
          System.out.println("Mike has withdrawn : " +
obtained);
          System.out.println(mike.statement() );
      } //main
    }// TestNormalAccount
```

Interface

An interface is a set of methods such that if a class implements the interface, it must provide definitions (implementations) for all methods specified by the interface.

Below is an interface.

```
public interface AccountProtocol {
     public double accountBalance( );
     public void deposit( final double money );
     public double withdraw( final double money );
}
```

Below is a class that implements the interface AccountProtocol.

```
public class SimpleAccount implements AccountProtocol {
     private double theBalance = 0.0d;

     public double accountBalance(){
      return theBalance;
     }

   public double withdraw(final double money){
        if (theBalance - money >= 0.0){
         theBalance = theBalance - money;
         return money;
      }
      else {
         return 0.0;
      }
   }

   public void deposit (final double money){
      theBalance = theBalance + money;
   }

   //method transfer may be passed an object implementing
AccoutProtocol

   public double transfer( AccountProtocol other, final
double money ) {
          if (money > 0.0) {
               double obtained = other.withdraw(money);
               if (obtained != 0.0 ) {
                    deposit(money);
                    return money;
               }
          }
          return 0.0;
     }//transfer
}// end class
```

Follows is an application that tests the interface and the class implementing it.

```
class TestSimpleAccount {
   public static void main (String args []) {
      SimpleAccount mike = new SimpleAccount( );

      SimpleAccount corinna = new SimpleAccount();
      double obtained;
      System.out.println("Mike's balance = " +
mike.accountBalance());

      mike.deposit(100.0);
      System.out.println("Mike's balance = " +
mike.accountBalance());
      obtained = mike.withdraw(20.0);
      System.out.println("Mike has withdrawn : " +
obtained);
      System.out.println("Mike's balance = " +
mike.accountBalance());

      corinna.deposit(50.0);
      System.out.println("Corinna's balance = " +
corinna.accountBalance());

      obtained = 15.0;
      System.out.println("Transfering  " + obtained + "
from Corinna's to Mike's ");
      mike.transfer(corinna, obtained);
      System.out.println("Mike's balance = " +
mike.accountBalance());
      System.out.println("Corinna's balance = " +
corinna.accountBalance());
      System.out.println();
   }
}
```

Classes: more details (from the text)

The **this** Parameter

- All instance variables are understood to have **<the calling object>.** in front of them
- If an explicit name for the calling object is needed, the keyword **this** can be used
  - **myInstanceVariable** always means and is always interchangeable with **this.myInstanceVariable**
- **this *must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

int someVariable = this.someVariable

someVariable is local, this.someVariable is the instance's variable

Testing Methods

- Each method should be tested in a program in which it is the only untested program
  - A program whose only purpose is to test a method is called a *driver program*
- One method often invokes other methods, so one way to do this is to first test all the methods invoked by that method, and then test the method itself
  - This is called *bottom-up testing*
- Sometimes it is necessary to test a method before another method it depends on is finished or tested
  - In this case, use a simplified version of the method, called a *stub,* to return a value for testing
  -

The Fundamental Rule for Testing Methods

- ***Every method should be tested in a program in which every other method in the testing program has already been fully tested and debugged***

Information Hiding and Encapsulation

- *Information hiding* is the practice of separating how to use a class from the details of its implementation
  - *Abstraction* is another term used to express the concept of discarding details in order to avoid information overload
- *Encapsulation* means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them `private`

Accessor and Mutator Methods

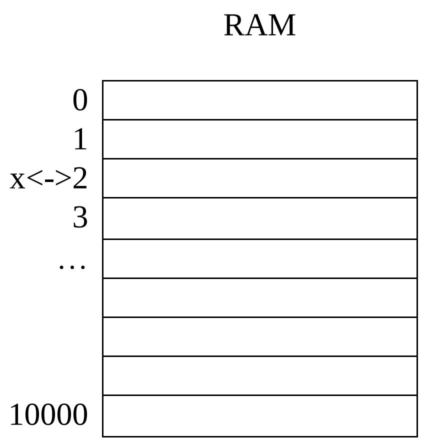
- *Accessor* methods (getters) allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **get**
- *Mutator* methods (setters) allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **set**

PARAMETER PASSING

Pointers, references, memory addresses

int   x; // variable declaration

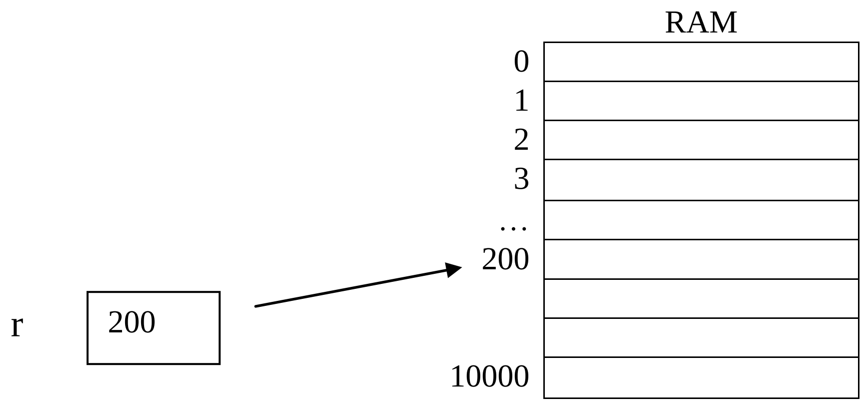
Every variable (object) is assigned a memory location (address, reference in Java) in RAM . x may be assigned, say, location 2.



Compiler constructs a symbol table to keep tracks of the addresses of variables.

Name	Address
x	2
y	5

When we refer to x in our program, the compiler refers to (address) location 2.
x = 124; means 124 is assigned to location 2.
Content of x is an integer
A reference is a variable whose content is an address of some other variables or objects. Suppose r is a reference and r contains 200



A class is a description of all objects that share the same member variables and member methods. An object of a class is created with the keyword **new**. For example:

Account temp = new Account( );

An object has physical existence (it occupies memory space, ie. RAM)
A class has no physical existence. A class is a muold used to create objects.
Objects of the same class have the same structure, but occupy difference memory locations.
Objects are reference types.

Recall: actual parameters (actuals) are the parameters of the caller, formal parameters (formals) are the parameters of the callee. In call by reference, the actuals and the formals refer to the same memory locations; thus changes to the formals apply as well to the actuals. In call by value, when the subroutine is invoked a copy of the formals are created, each occupying a memory location different than that of the actuals, the values of the actuals are copied into the formals.

In C, most parameters are passed by value with a few exceptions, one of which is arrays which are passed by reference.

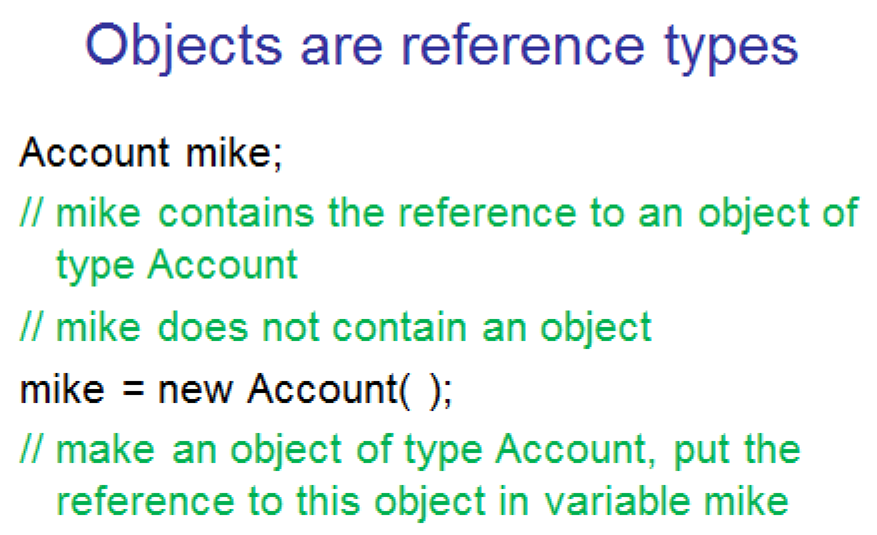
```
#include <stdio.h>
void add( int a[ ], int n );
void main( void )
{
   int i;
   int n = 2;
   int x[3];
   for(  i = 0; i < 3; i++ )
        x[i] = i;
   add( x, n);
   printf( " n  =  %i \n", n );
   for(  i = 0; i < 3; i++ )
    printf(" x[%i]  =  %i  \n", i, x[i] );
}

void add( int a[ ], int n )
{
      for( int i = 0; i < 3; i ++ )
           a[i]++;
      n = n + 5;
}
```

For the call "`add( x, n )`", the array `x` is passed by reference and the integer `n` is passed by value. Thus, `x` and `a` refer to the same array. While the variable `n` in `main` is different from the variable `n` in `add`.

The following is printed by main:

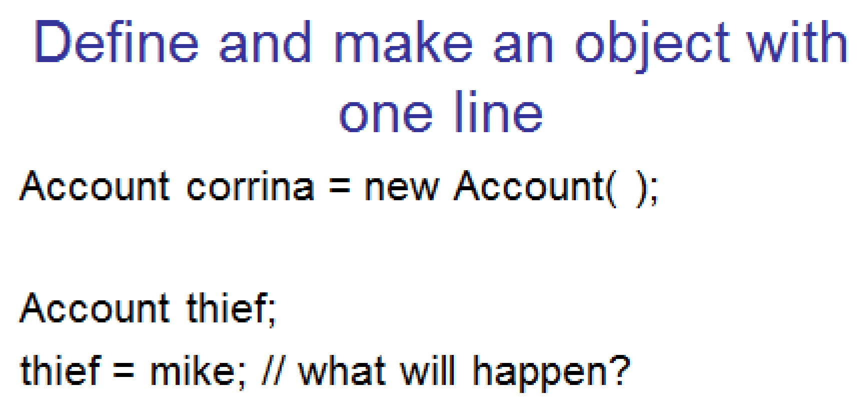
```
n = 2
x[0] = 1
x[1] = 2
x[2] = 3
```

In Java, parameters are always passed by value. If the parameter is reference type, then the actual and formal refer to the same object (because they contain the same value which is the address of the object); thus this has the same effect as pass-by-reference.



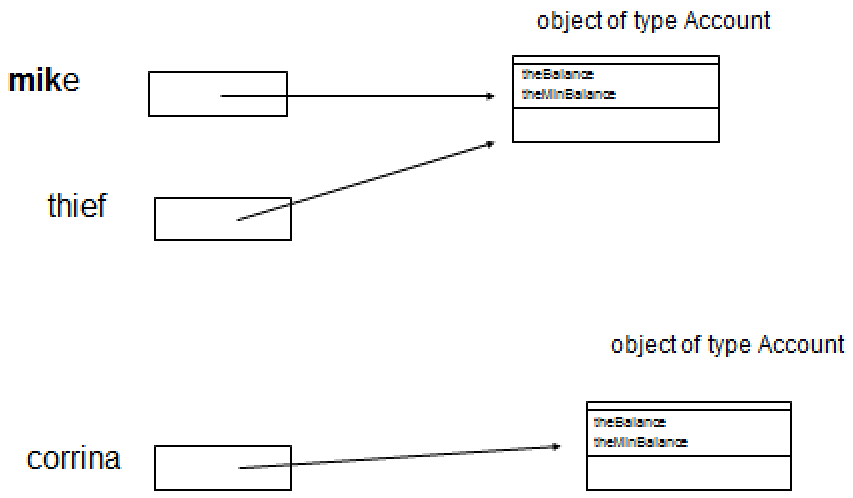
Objects are reference types

```
Account mike;
// mike contains the reference to an object of
   type Account
// mike does not contain an object
mike = new Account( );
// make an object of type Account, put the
   reference to this object in variable mike
```



Define and make an object with one line

```
Account corrina = new Account( );

Account thief;
thief = mike; // what will happen?
```



- mike.deposit(100.0);
- thief.withdraw(70.0);
- // the thief steals $70 from mike
- System.out.println(mike.accountBalance());
- prints 30.0

```
import java.io.*;

class QueueNode {
      protected int element;
      protected QueueNode next;
  }


public class Parameter {
      protected static void change(QueueNode t, int n){
            t.element = 7;
            n = 9;
      }
      public static void main(String args[]) {
            int n = 3;
            QueueNode One = new QueueNode( );
            One.element = 5;
            change( One, n );
            System.out.println(One.element  );
            System.out.println(n  );

      }
}
```

Output:

```
7
3
```