Lesson 5: Introduction to Defining Classes

Updated for Java 1.5 (Additions and Modifications by Mr. Dave Clausen)

Lesson 5: Introduction to Defining Classes

Objectives:

- Design and implement a simple class from user requirements.
- Organize a program in terms of a view class and a model class.
- Use visibility modifiers to make methods visible to clients and restrict access to data within a class.

Lesson 5: Introduction to Defining Classes

Objectives:

- Write appropriate mutator methods, accessor methods, and constructors for a class.
- Understand how parameters transmit data to methods.
- Use instance variables, local variables, and parameters appropriately.
- Organize a complex task in terms of helper methods.

Lesson 5: Introduction to Defining Classes

Vocabulary:

- accessor
- actual parameter
- behavior
- constructor
- encapsulation
- formal parameter
- helper method
- identity
- instantiation
- lifetime
- mutator
- scope
- state
- visibility modifier

5.1 The Internal Structure of Classes and Objects

- An ***object*** is a run-time entity that contains data and responds to messages.
- A ***class*** is a software package or template that describes the characteristics of similar objects.
- These characteristics are of two sorts:
  - Variable declarations that define an object's data requirements (instance variables).
  - Methods that define its behavior in response to messages.

5.1 The Internal Structure of Classes and Objects

- The combining of data and behavior into a single software package is called ***encapsulation.***
- An object is an instance of its class, and the process of creating a new object is called ***instantiation.***

5.1 The Internal Structure of Classes and Objects

Classes, Objects, and Computer Memory

- When a Java program is executing, the computers memory must hold:
  - All class templates in their compiled form
  - Variables that refer to objects
  - Objects as needed
- Each method's compiled byte code is stored in memory as part of its class's template.

5.1 The Internal Structure of Classes and Objects

- Memory for data is allocated within objects.
- All class templates are in memory at all times, individual objects come and go.
- An object first appears and occupies memory when it is instantiated, and it disappears automatically when no longer needed.

5.1 The Internal Structure of Classes and Objects

- The JVM knows if an object is in use by keeping track of whether or not there are any variables referencing it.
- Because unreferenced objects cannot be used, Java assumes that it is okay to delete them from memory via ***garbage collection***.

5.1 The Internal Structure of Classes and Objects

Three Characteristics of an Object

- 1. An object has ***behavior*** as defined by the **methods** of its class
- 2. An object has ***state***, which is another way of saying that at any particular moment its instance variables have particular **values**.
  - Typically, the state changes over time in response to messages sent to the object.

5.1 The Internal Structure of Classes and Objects

- 3. An object has its own unique ***identity***, which distinguishes it from all other objects in the computers memory.
  - An objects identity is handled behind the scenes by the JVM and should not be confused with the variables that might refer to the object.
  - When there are no variables the garbage collector purges the object from memory.

5.1 The Internal Structure of Classes and Objects

Clients, Servers, and Interfaces

- When messages are sent, two objects are involved:
  - The sender (the client)
  - The receiver (the server)
- Client:
  - A client's interactions with a server are limited to sending it messages.
  - A client needs to know only a server's ***interface***, that is, the list of methods supported by the server.

5.1 The Internal Structure of Classes and Objects

- Server
  - The server's data requirements and the implementation of its methods are hidden from the client (***information hiding***).
  - Only the person who writes a class needs to understand its internal workings
  - A class's implementation details can be changed radically without affecting any of its clients provided its interface remains the same.

5.2 A Student Class

The student object stores a name and three test scores and responds to the message shown in Table 5-1.

METHODS	DESCRIPTIONS
`void setName(aString)`	Example: `stu.setName ("Bill");` Sets the name of `stu` to Bill.
`String getName()`	Example: `str = stu.getName();` Returns the name of `stu`.
`void setScore (whichTest, testScore)`	Example: `stu.setScore (3, 95);` Sets the score on test 3 to 95. If `whichTest` is not 1, 2, or 3, then 3 is substituted automatically.
`int getScore(whichTest)`	Example: `score = stu.getScore (3);` Returns the score on test 3. If `whichTest` is not 1, 2, or 3, then 3 is substituted automatically.
`int getAverage()`	Example : `average = stu.getAverage();` Returns the average of the test scores.
`int getHighScore()`	Example: `highScore = stu.getHighScore();` Returns the highest test score.
`String toString()`	Example: `str = stu.toString();` Returns a string containing the student's name and test scores.

5.2 A Student Class

Using Student Objects

- Some portions of code illustrate how a client instantiates and manipulates student objects. First we declare several variables, including two variables of type Student.

```
Student s1, s2;  // Declare the variables
String str;
int i;
```

- As usual, we do not use variables until we instantiate them. We assign a new student object to s1 using the operator new.

```
s1 = new Student();  // Instantiate a student and associate it with
                     // the variable s1
```

5.2 A Student Class

- We could declare and instantiate a `Student` object in one step:

```
Student s1 = new Student();
```

  - It is important to emphasize that the variable s1 is a reference to a student object and is *not* a student object itself.

  - A student object keeps track of the name and test scores of an actual student. Thus, for a brand new student object, what are the values of these data attributes?

5.2 A Student Class

- That depends on the class's internal implementation details, but we can find out easily by sending messages to the student object via its associated variable s1:

```
str = s1.getName();
System.out.println (str);    // yields ""
i = s1.getHighScore();
System.out.println (i);      // yields 0
```

5.2 A Student Class

- Apparently, the name was initialized to an empty string and the test scores to zero. Now we set the object's data attributes by sending it some messages:

```
s1.setName ("Bill");  // Set the student's name to "Bill"
s1.setScore (1,84);   // Set the score on test 1 to 84
s1.setScore (2,86);   // Set the score on test 2 to 86
s1.setScore (3,88);   // Set the score on test 3 to 88
```

5.2 A Student Class

- Messages that change an object's state are called ***mutators***. (modifiers or transformers in C++)
- To see if the mutators worked correctly, we use other messages to access the object's state (called ***accessors***):

```
str = s1.getName();        // str equals "Bill"
i = s1.getScore (1);       // i equals 84
i = s1.getHighScore();     // i equals 88
i = s1.getAverage();       // i equals 86
```

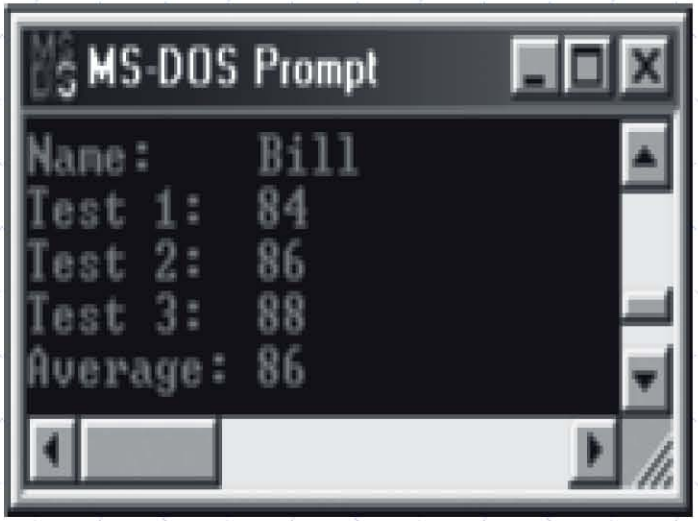
5.2 A Student Class

- The object's string representation is obtained by sending the toString message to the object.

```
str = s1.toString();
// str  now equals
// "Name: Bill\nTest 1:  84\nTest2:
//86\nTest3: 88\nAverage: 86"
```

5.2 A Student Class

- When displayed in a terminal window (Figure 5-1), the string is broken into several lines as determined by the placement of the newline characters ('\n').



5.2 A Student Class

- In addition to the explicit use of the toString method, there are other situations in which the method is called automatically. For instance, toString is called implicitly when a student object is concatenated with a string or is in an argument to the method println:

```
str = "The best student is: \n" + s1;
   // Equivalent to: str = "The best student is: \n" + s1.toString();
System.out.println (s1);
   // Equivalent to: System.out.println (s1.toString());
```

5.2 A Student Class

Objects, Assignments, and Aliasing

- We close this demonstration by associating a student object with the variable s2. Rather than instantiating a new student, we assign s1 to s2:

```
s2 = s1;          // s1 and s2 now refer to the same student
```

5.2 A Student Class

- The variables s1 and s2 now refer to the *same* student object.
- This might come as a surprise because we might reasonably expect the assignment statement to create a second student object equal to the first, but that is not how Java works.
- To demonstrate that s1 and s2 now refer to the same object, we change the students name using s2 and retrieve the same name using s1:

```
s2.setName ("Ann"); // Set the name
str = s1.getName();
// str equals "Ann". Therefore, s1 and s2 refer to the same object.
```

5.2 A Student Class

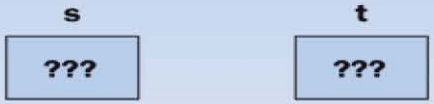
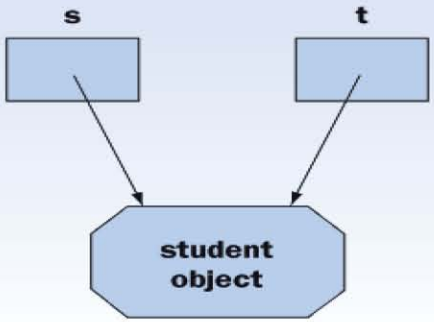
- Table 5-2 shows code and a diagram that clarify the manner in which variables are affected by assignment statements. At any time, it is possible to break the connection between a variable and the object it references. Simply assign the value null to the variable:

```
Student s1;
s1 = new Student();
// s1 references the newly instantiated student
...// Do stuff with the student
s1 = null;  // s1 no longer references anything
```

5.2 A Student Class

- Table 5-2 demonstrates that assignment to variables of numeric types such as int produces genuine copies, whereas assignment to variables of object types does not.

5.2 A Student Class

CODE	DIAGRAM	COMMENTS
`int i, j;`		`i` and `j` are memory locations that have not yet been initialized, but which will hold integers.
`i = 3;` `j = i;`		`i` holds the integer 3. `j` holds the integer 3.
`Student s, t;`		`s` and `t` are memory locations that have not yet been initialized, but which will hold references to student objects.
`s = new Student();` `t = s;`		`s` holds a reference to a student object. `t` holds a reference to the same student object.

5.2 A Student Class

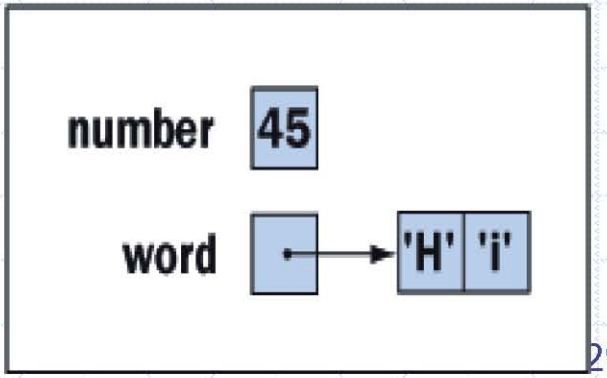
Primitive Types, Reference Types, and the null Value

- Two or more variables can refer to the same object.
- In Java, all types fall into two fundamental categories
  - **Primitive types**: int, double, boolean, char, and the shorter and longer versions of these
  - **Reference types**: all classes, for instance, String, Student, Scanner, and so on

5.2 A Student Class

- A variable of a primitive type is best viewed as a box that contains a value of that primitive type.
- A variable of a reference type is thought of as a box that contains a pointer to an object.
- The state of memory after the following code is executed is shown in figure 5-2

```
int number = 45;
String word = "Hi";
```



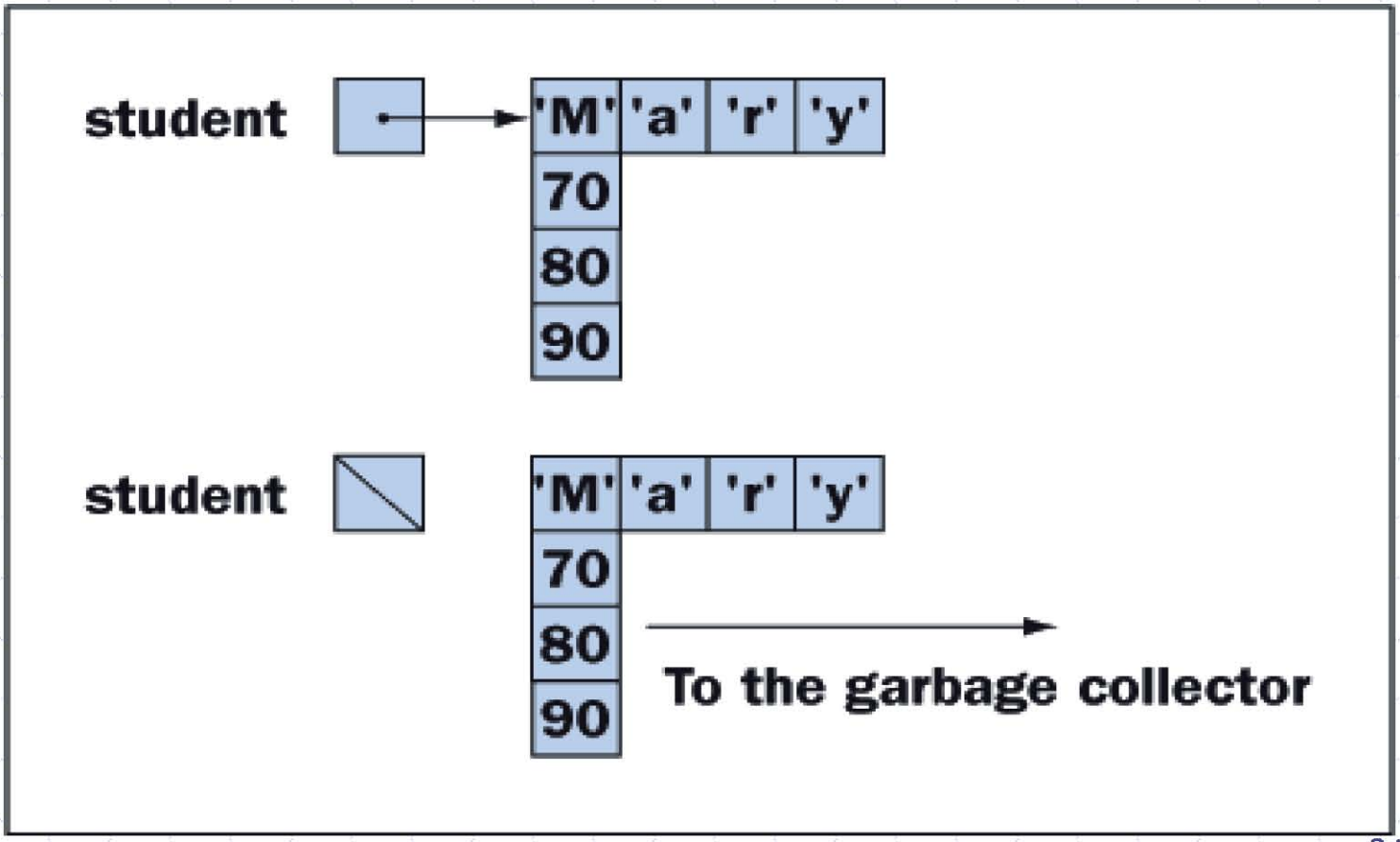
5.2 A Student Class

- Reference variables can be assigned the value null.
- If a reference variable previously pointed to an object, and no other variable currently points to that object, the computer reclaims the object's memory during garbage collection.

```
Student student = new Student ("Mary", 70,
    80, 90);
student = null;
```

5.2 A Student Class

The Student variable before and after it has been assigned the value null



5.2 A Student Class

- A reference variable can be compared to the null value, as follows:

```
if (student == null)
      // Don't try to run a method with that student!
else
      // Process the student
while (student != null)
{
      // Process the student
      // Obtain the next student from
whatever source
}
```

5.2 A Student Class

- When a program attempts to run a method with an object that is null, Java throws a ***null pointer exception***, as in the following example:

```
String str = null;
System.out.println (str.length());
// OOPS! str is null, so Java throws a
// null pointer exception
```

5.2 A Student Class

The Structure of a Class Template

- All classes have a similar structure consisting of four parts:
  - 1. The class's name and some modifying phrases
  - 2. A description of the instance variables
  - 3. One or more methods that indicate how to initialize a new object (called ***constructor*** methods)
  - 4. One or more methods that specify how an object responds to messages

5.2 A Student Class

- The following is an example of a class template:

```
public class <name of class> extends <some other class>
{
    // Declaration of instance variables
    private <type> <name>;
    ...
    // Code for the constructor methods
    public <name of class>()
    {
       // Initialize the instance variables
       ...
    }
    ...
    // Code for the other methods
    public <return type> <name of method> (<parameter
list>){
       ...
    }
    ...
}
```

5.2 A Student Class

- Class definitions usually begin with the keyword public, indicating that the class is accessible to all potential clients.
  - public class
- Class names are user-defined symbols, and must adhere to the rules for naming variables and methods.
- It is common to start class names with a capital letter and variable and method names with a lowercase letter.
  - <name of class>

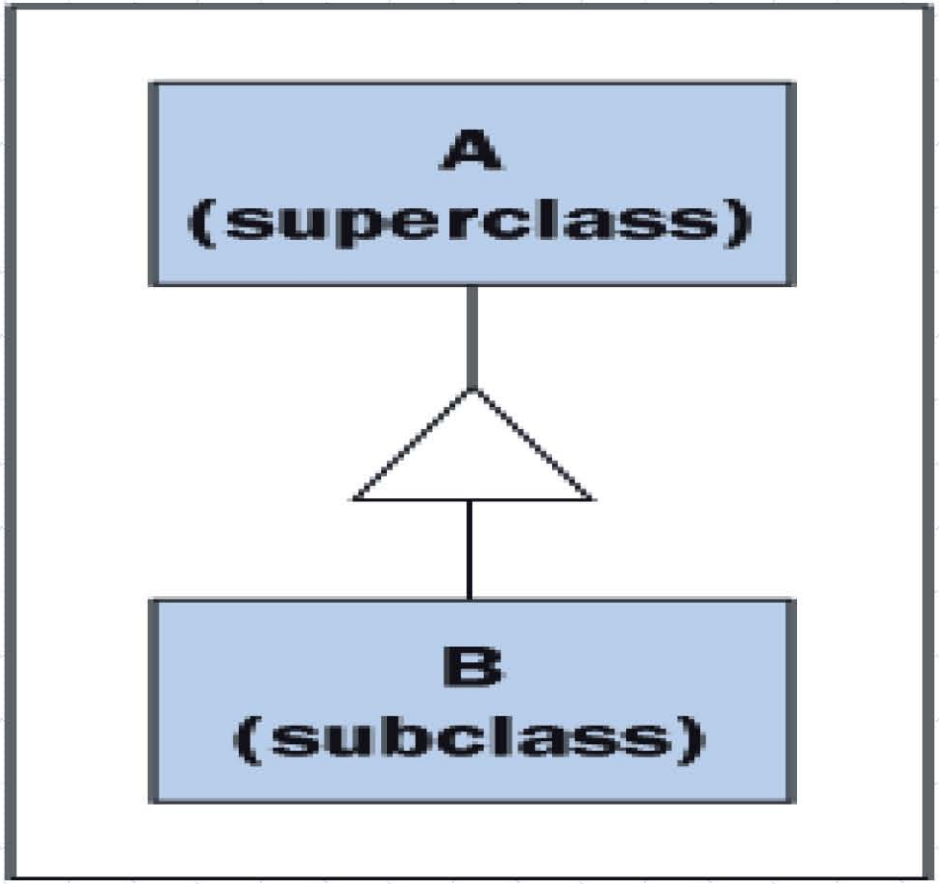
5.2 A Student Class

- Java organizes its classes in a hierarchy. At the base of this hierarchy is a class called Object.
- In the hierarchy, if class A is immediately above another class B, we say that A is the ***superclass*** or ***parent*** of B and B is a ***subclass*** or ***child*** of A.
- Each class, except Object, has exactly one parent and can have any number of children.

  extends <some other class>

5.2 A Student Class

Relationship between superclass and subclass



5.2 A Student Class

- When a new class is created, it is incorporated into the hierarchy by extending an existing class
- The new class's exact placement in the hierarchy is important because a new class inherits the characteristics of its superclass through a process called ***inheritance***.
- If the clause extends <some other class> is omitted from the new class's definition, then by default, the new class is assumed to be a subclass of Objects.

5.2 A Student Class

- Instance variables are nearly always declared to be private.
- This prevents clients from referencing to the instance variables directly.
- Making instance variables private is an important aspect of information hiding.

  private <type> <name>

5.2 A Student Class

- Methods are usually declared to be public, which allows clients to refer to them.

  public <return type> <name of method>

- The clauses private and public are called ***visibility modifiers***.
- Omitting the visibility modifier is equivalent to using public.

5.2 A Student Class

Implementation of the Student Class: Student.java

```
/* Student.java Student.txt
Manage a student's name and three test scores.
*/
public class Student {
   //Instance variables
   //Each student object has a name and three test scores
   private String name;              //Student name
   private int test1;                //Score on test 1
   private int test2;                //Score on test 2
   private int test3;                //Score on test 3

   //Constructor method

   public Student(){     //constructor
   //Initialize a new student's name to the empty string and the test
   //scores to zero.
      name = "";
      test1 = 0;
      test2 = 0;
      test3 = 0;
  }
```

5.2 A Student Class

```
//Other methods

public void setName (String nm){ //modifier
//Set a student's name
   name = nm;
}

public String getName (){  //accessor
//Get a student's name
   return name;
}

public void setScore (int i, int score){    //modifier
//Set test i to score
   if      (i = = 1) test1 = score;
   else if (i = = 2) test2 = score;
   else              test3 = score;
}
```

5.2 A Student Class

```
public int getScore (int i){  //accessor
//Retrieve score i
   if      (i == 1) return test1;
   else if (i == 2) return test2;
   else             return test3;
}

public int getAverage(){   //accessor
//Compute and return the average
   int average;
   average = (int) Math.round((test1 + test2 + test3) / 3.0);
   return average;
}

public int getHighScore(){   //accessor
//Determine and return the highest score
   int highScore;
   highScore = test1;
   if (test2 > highScore) highScore = test2;
   if (test3 > highScore) highScore = test3;
   return highScore;
}
```

5.2 A Student Class

```
    public String toString(){
    //Construct and return a string representation of the student
        String str;
        str = "Name:     " + name  + "\n" +
        // "\n" denotes a newline
              "Test 1:  " + test1 + "\n" +
              "Test 2:  " + test2 + "\n" +
              "Test 3:  " + test3 + "\n" +
              "Average: " + getAverage();
        return str;
    }
}
```

5.2 A Student Class

- In the preceding example all the methods, except the constructor method, have a return type, although the return type may be void, indicating that the method in fact returns nothing.
- In summary when an object receives a message, the object activates the corresponding method. The method then manipulates the object's data as represented by the instance variables.

5.2 A Student Class

Constructors

- The principal purpose of a constructor is to initialize the instance variables of a newly instantiated object.
- Constructors are activated when the keyword new is used and at no other time.
- A constructor is never used to reset instance variables of an existing object.

5.2 A Student Class

- A class template can include more than one constructor, provided each has a unique parameter list; however, all the constructors must have the same name- that is, the name of the class.
- Constructors with empty parameter lists and are called ***default constructors***.
- If a class template contains no constructors then JVM provides a primitive default constructor.
- This constructor initializes numeric variables to zero and object variables to null, thus indicating that the object variables currently reference no objects.

5.2 A Student Class

To illustrate we add several constructors to the student class.

```
// Default constructor -- initialize name to the empty string and
// the test scores to zero.
public Student(){
   name = "";
   test1 = 0;
   test2 = 0;
   test3 = 0;
}

// Parameterized constructor -- initialize the name and test scores
// to the values provided.
public Student(String nm, int t1, int t2, int t3){
   name = nm;
   test1 = t1;
   test2 = t2;
   test3 = t3;
}

// Copy constructor -- initialize the name and test scores
// to match those in the parameter s.
public Student(Student s){
   name = s.name;
   test1 = s.test1;
   test2 = s.test2;
   test3 = s.test3;
}
```

5.2 A Student Class

Chaining Constructors

- When a class includes several constructors, the code for them can be simplified by ***chaining*** them.
- The three constructors in the Student class each do the same thing – initialize the instance variables.
- Simplify the code for the first and third constructors by calling the second constructor.
- To call one constructor from another constructor, we use the notation:

```
this (<parameters>);
```

5.2 A Student Class

- Thus, the code for the constructors becomes (P. 167) :

```
// Default constructor -- initialize name to the empty string and
// the test scores to zero.
public Student()
{
   this("", 0, 0, 0);
}


 // Parameterized  constructor -- initialize the name and test scores
// to the values provided.
public Student(String nm, int t1, int t2, int t3){
   name = nm;
   test1 = t1;
   test2 = t2;
   test3 = t3;
}

// Copy constructor -- initialize the name and test scores
// to match those in the parameter s.
public Student(Student s){
   this(s.name, s.test1, s.test2, s.test3);
}
```

5.3 Editing, Compiling, and Testing the Student Class

- To use the Student class, we must save it in a file called ***Student.java*** and compile it by typing the following in a terminal window:

```
javac Student.java
```

- If there are no compile-time errors, the compiler creates the byte code file ***Student.class***

5.3 Editing, Compiling, and Testing the Student Class

- Once the Student class is compiled, applications can declare and manipulate student objects provided that:
  - The code for the application and ***Student.class*** are in the same directory or
  - The ***Student.class*** is part of a package

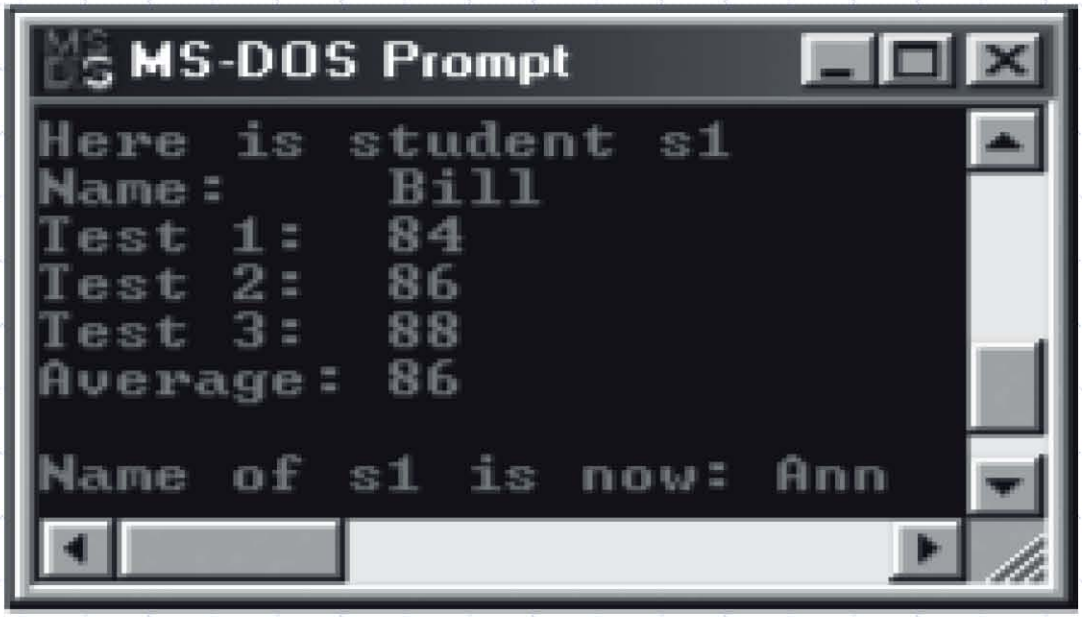
5.3 Editing, Compiling, and Testing the Student Class

The following is a small program that uses and tests the student class: TestStudent.java

```
public class TestStudent{     //TestStudent.txt

   public static void main (String[] args){
      Student s1, s2;
      String str;
      int i;

      s1 = new Student();      // Instantiate a student object
      s1.setName ("Bill");     // Set the student's name to "Bill"
      s1.setScore (1,84);      // Set the score on test 1 to 84
      s1.setScore (2,86);      //               on test 2 to 86
      s1.setScore (3,88);      //               on test 3 to 88
      System.out.println("\nHere is student s1\n" + s1);

      s2 = s1;                 // s1 and s2 now refer to the same object
      s2.setName ("Ann");      // Set the name through s2
      System.out.println ("\nName of s1 is now: " + s1.getName());
   }
}
```

5.3 Editing, Compiling, and Testing the Student Class

- Figure 5-5 shows the results of running such a program.



5.3 Editing, Compiling, and Testing the Student Class

Finding the Locations of Run-time Errors

- The messages indicate that
  - An attempt was made to divide by zero in the Student class's getAverage method (line 50)
  - Which had been called from the Student class's toString method (line 64)
  - Which had been called by some methods we did not write.
  - Which, finally, had been called from the TestStudent class's main method (line 13)

5.3 Editing, Compiling, and Testing the Student Class

Following are the lines of code mentioned:

```
Student getAverage line 50 :
average = (int) Math.round ((test1 + test2 + test3)
              / average);
Student toString line 64 :
"Average: " + getAverage();
TestStudent main line 13 :
System.out.println ("\nHere is student s1\n" + s1);
```

5.3 Editing, Compiling, and Testing the Student Class

- We can now unravel the error.
  - In line 13 of main, the concatenation (+) of s1 makes an implicit call s1.toString().
  - In line 64 of toString, the getAverage method is called.
  - In line 50 of getAverage, a division by zero occurs.

Case Study 1

StudentApp.java

StudentApp.txt

5.4 The Structure and Behavior of Methods

The Structure of a Method Definition

- Methods generally have the following form:

```
<visibility modifier> <return type> <method name> (<parameter list>)
{

    <implementing code>
}
```

5.4 The Structure and Behavior of Methods

Return Statements

- If a method has a return type, its implementing code must have at least one return statement that returns a value of that type.
- There can be more than one return statement in a method; however, the first one executed ends the method.
- A return statement in a void method quits the method and returns nothing.

5.4 The Structure and Behavior of Methods

- The following is an example of a method that has two return statements but executes just one of them:

```
boolean odd(int i)
{
    if (i % 2 == 0)
        return false;
    else
        return true;
}
```

5.4 The Structure and Behavior of Methods

Formal and Actual Parameters

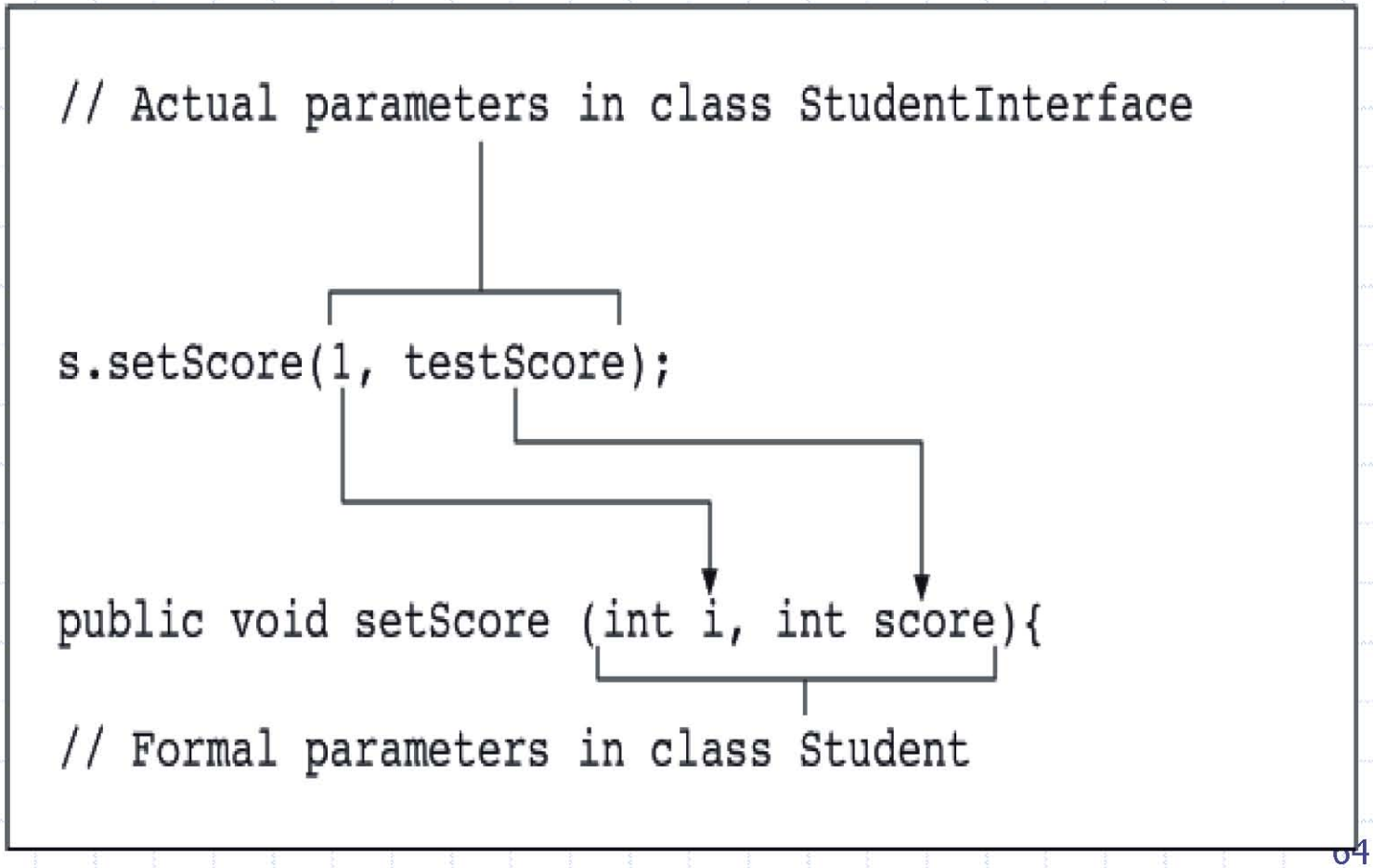
- Parameters listed in a method's definition are called ***formal parameters***. Values passed to a method when it is invoked are called ***arguments*** or ***actual parameters***.

```
// Client code

Student s = new Student();
Scanner reader = new Scanner(System.in);
System.out.print("Enter a test score:");
int testScore = reader.nextInt();
s.setScore(1, testScore);
```

```
// Server code

public void setScore (int i, int score){
   if      (i == 1) test1 = score;
   else if (i == 2) test2 = score;
   else             test3 = score;
}
```

5.4 The Structure and Behavior of Methods

Parameter passing



5.4 The Structure and Behavior of Methods

- When a method has a multiple parameters, the caller must provide the right number and types of values.
- The actual parameters must match the formal parameters in position and type.
- The rules for matching the types of a formal and an actual parameter are similar to those for assignment statements.

5.4 The Structure and Behavior of Methods

- The actual parameter's type must be either the same as or less inclusive than the type of the corresponding formal parameter.
- For example, the method Math.sqrt, which has a single formal parameter of type double, can receive either a double or an int as an actual parameter from the caller.

5.4 The Structure and Behavior of Methods

Parameters and Instance Variables

- The purpose of a parameter is to pass information to a method.
- The purpose of an instance variable is to maintain information in an object.
- These roles are clearly shown in the method setScore in figure 5-8.
- This method receives the score in the formal parameter score.
- This value is then transferred to one of the instance variables test1, test2, test3.

5.4 The Structure and Behavior of Methods

Local Variables

- Occasionally, it is convenient to have temporary working storage for data in a method.
- The programmer can declare ***local variables*** for this purpose.
- The following example declares a variable average, assigns it the result of computing the average of the integer instance variables, and returns its value.

5.4 The Structure and Behavior of Methods

```
public double getAverage()
{
    double average;  //local variable
    average = (test1 + test2 + test3) / 3.0;
    return average;
}
```

- Note that there is no need for the method to receive data from the client, so we do not use a parameter.
- There is no need for the object to remember the average, so we do not use an instance variable for that.

5.4 The Structure and Behavior of Methods

Helper Methods

- Occasionally, a task performed by a method becomes so complex that it helps to break it into subtasks to be solved by several other methods.
- A class can define one or more methods to serve as ***helper methods***.
- Helper methods are usually private, because only methods already defined within the class need to use them.

5.4 The Structure and Behavior of Methods

- For example, it is helpful to define a debug when testing a class.
- This method expects a string and a double as parameters and displays these values in the terminal window.

```
private void debug(String message,  double value)
{
    System.out.println(message + "  " + value);
}
```

5.4 The Structure and Behavior of Methods

- The advantage to this approach is that debugging statements throughout the class can be turned on or off by commenting out a single line of code:

```
private void debug(String message, double value)
{
    //System.out.println(message + " " + value);

}
```

5.5 Scope and Lifetime of Variables

- A class definition consists of two principal parts:
  - a list of instance variables and
  - a list of methods.
- When an object is instantiated, it receives its own complete copy of the instance variables, and when it is sent a message, it activates the corresponding method in its class.
- It is the role of objects to contain data and to respond to messages
- It is the role of classes to provide a template for creating objects and to store the code for methods.

5.5 Scope and Lifetime of Variables

- When a method is executing, it does so on behalf of a particular object, and the method has complete access to the object's instance variables.
- The instance variables form a common pool of variables accessible to all the class's methods, called ***global variables***.
- Variables declared within a method are called ***local variables***.

5.5 Scope and Lifetime of Variables

Scope of Variables

- The scope of a variable is that region of the program within which it can validly appear in lines of code.
- The scope of a parameter or a local variable is restricted to the body of the method that declares it
- The scope of a global or instance variable is all the methods in the defining class.
- The compiler flags as an error any attempt to use variables outside of their scope.

5.5 Scope and Lifetime of Variables

- Following is an example that illustrates the difference between local and global scope:

```
public class ScopeDemo
{
   private int iAmGlobal;

   public void clientMethod (int parm)
   {
       int iAmLocal;
    }
   private int helperMethod (int parm1, int parm2){

      int iAmLocalToo;
      }
 }
```

5.5 Scope and Lifetime of Variables

- Table 5-3 shows where each of the variables and parameters can be used (i.e., its scope):
- Notice that formal parameters are also local in scope, that is, their visibility is limited to the body of the method in which they are declared.

VARIABLE	`helperMethod`	`clientMethod`
iAmGlobal	Yes	Yes
parm	No	Yes
iAmLocal	No	Yes
parm1 and parm2	Yes	No
iAmLocalToo	Yes	No

5.5 Scope and Lifetime of Variables

Block Scope

- Within the code of a method, there can also be nested scopes.
- Variables declared within any compound statement enclosed in { } are said to have ***block scope***.
- They are visible only within the code enclosed by { }.

5.5 Scope and Lifetime of Variables

- For example, consider the following for loop to sum 10 input numbers:

```
int sum = 0;
int number =0;
Scanner reader = new Scanner();
for (int i = 1; i <= 10; i++){
    System.out.print("Enter a  number: ");
    number = reader.nextInt();
    sum += number;
}
System.out.println("The sum is " + sum);
```

5.5 Scope and Lifetime of Variables

Lifetime of Variables

- The lifetime of a variable is the period during which it can be used.
- Local variables and formal parameters exist during a single execution of a method.
  - Each time a method is called, it gets a fresh set of formal parameters and local variables
  - Once the method stops executing, the formal parameters and local variables are no longer accessible.

5.5 Scope and Lifetime of Variables

- Instance variables last for the lifetime of an object.
  - When an object is instantiated, it gets a complete set of fresh instance variables.
  - These variables are available every time a message is sent to the object, and they, in some sense, serve as the object's memory.
  - When the object stops existing, the instance variables disappear too.

5.5 Scope and Lifetime of Variables

Duplicating Variable Names

- Because the scope of a formal parameter or local variable is restricted to a single method, the same name can be used within several different methods without causing a conflict.
- When the programmer reuses the same local name in different methods, the name refers to a different area of storage in each method.
- In the next example, the names `iAmLocal` and `parm1` are used in two methods in this way:

5.5 Scope and Lifetime of Variables

```
public class ScopeDemo {

    private int iAmGlobal;

    public void clientMethod (int parm1){
        int iAmLocal;
        ...
    }
    private int helperMethod (int parm1,
        int parm2){

        int iAmLocal;
        ...
    }
    ...
}
```

5.5 Scope and Lifetime of Variables

When to Use Instance Variables, Parameters, and Local Variables

- The only reason to use an instance variable is to remember information within an object.
- The only reason to use a parameter is to transmit information to a method.
- The only reason to use a local variable is for temporary working storage within a method.
- A very common mistake is to misuse one kind of variable for another.
- Following are the most common examples of these types of mistakes:

5.5 Scope and Lifetime of Variables

A global variable is used for temporary working storage

- The method runs correctly only the first time.
- The next time the method is called, it adds scores to the sum of the previous call, thus producing a much higher average than expected.

```
private int sum; //global to the class
...
public int getAverage(){
    for (int i = 1; i <= 3; i++)
        sum += getScore(i);
    return (int) Math.round(sum / 3.0);
}
```

5.5 Scope and Lifetime of Variables

A local variable is used to remember information in an object

- This mistake can lead to errors in cases where the programmer uses the same name for a local variable and a global variable.
- In this case, the variable name has been accidentally "localized" by prefixing it with a type name.
- Thus, the value of the parameter nm is transferred to the local variable instead of the instance variable, and the Student object does not remember this change.

```
public void setName (String nm){
//Set a student's name
   String name = nm; //Whoops! we have just declared name local.
}
```

5.5 Scope and Lifetime of Variables

A method accesses data by directly referencing a global variable when it could use a parameter instead.

```
// Server class
public class ServerClass{

   private int x;
   public void m1(){
      ...
      x = 0;  // The real source of the error
   }
   public void m2(){
      int y = 10 / x;  // Exact spot of run-time error
   }
   ...
}
```

5.5 Scope and Lifetime of Variables

```
// Client class

public class ClientClass
{
     public void m3()
{
        m1(); // Misuse of x occurs, but is hidden from
                 //client
        m2(); // Run-time error occurs
    }
    ...
}
```

Summary

- Java class definitions consist of instance variables, constructors, and methods.
- Constructors initialize an object's instance variables when the object is created.
- A default constructor expects no parameters and sets the variables to default values.
- Mutator methods modify an object's instance variables.

Summary (cont.)

- Accessor methods allow clients to observe the values of these variables.
- The visibility modifier `public` makes methods visible to clients.
- `private` encapsulates access.
- Helper methods are called from other methods in a class definition.
  - Usually declared to be private

Summary (cont.)

- Instance variables track the state of an object.
- Local variables are used for temporary working storage within a method.
- Parameters transmit data to a method.
- A formal parameter appears in a method's signature and is referenced in its code.

Summary (cont.)

- Actual parameter is a value passed to a method when it is called.
- Scope of an instance variable is the entire class within which it is declared.
- Scope of a local variable or a parameter is the body of the method where it is declared.

Summary (cont.)

- Lifetime of an instance variable is the same as the lifetime of a particular object.
- Lifetime of a local variable and a parameter is the time during which a particular call of a method is active.