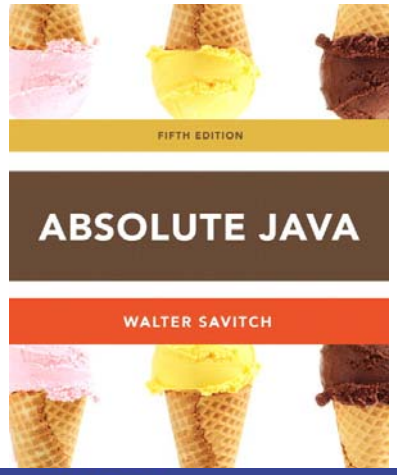


Chapter 4

Defining Classes I

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

Introduction

- Classes are the most important language feature that make *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java

Class Definitions

- You already know how to use classes and the objects created from them, and how to invoke their methods
  - For example, you have already been using the predefined `String` and `Scanner` classes
- Now you will learn how to define your own classes and their methods, and how to create your own objects from them

A Class Is a Type

- A class is a special kind of programmer-defined type, and variables can be declared of a class type
- A value of a class type is called an object or *an instance of the class*
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
- A class determines the types of data that an object can contain, as well as the actions it can perform

Primitive Type Values vs. Class Type Values

- A primitive type value is a single piece of data
- A class type value or object can have multiple pieces of data, as well as actions called *methods*
  - All objects of a class have the same methods
  - All objects of a class have the same pieces of data (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value

The Contents of a Class Definition

- A class definition specifies the data items and methods that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called *fields* or *instance variables*
- Instance variable declarations and method definitions can be placed in any order within the class definition

The **new** Operator

- An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

- The `new` operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

- These can be combined as follows:

```
ClassName classVar = new ClassName();
```


Instance Variables and Methods

- Instance variables can be defined as in the following two examples
  - Note the `public` modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body:*

```
public void myMethod()   <-- Heading
{
    code to perform some action      } Body
    and/or compute a value           }
}
```

- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

File Names and Locations

- Reminder: a Java file must be given the same name as the class it contains with an added **`.java`** at the end
  - For example, a class named **`MyClass`** must be in a file named **`MyClass.java`**
- For now, your program and all the classes it uses should be in the same directory or folder

More About Methods

- There are two kinds of methods:
  - Methods that compute and return a value
  - Methods that perform an action
    - This type of method does not return a value, and is called a **`void`** method
- Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

More About Methods

- A method that returns a value must specify the type of that value in its heading:

```
public typeReturned methodName(paramList)
```

- A **`void`** method uses the keyword **`void`** in its heading to show that it does not return a value :

```
public void methodName(paramList)
```

`main` is a `void` Method

- A program in Java is just a class that has a **`main`** method
- When you give a command to run a Java program, the run-time system invokes the method **`main`**
- Note that **`main`** is a **`void`** method, as indicated by its heading:

```
public static void main(String[] args)
```


`return` Statements

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
    declarations
    statements
}
```

Body

`return` Statements

- The body of a method that returns a value must also contain one or more **`return`** statements
  - A **`return`** statement specifies the value returned and ends the method invocation:

```
return Expression;
```

  - **`Expression`** can be any expression that evaluates to something of the type returned listed in the method heading

`return` Statements

- A **`void`** method need not contain a **`return`** statement, unless there is a situation that requires the method to end before all its code is executed
- In this context, since it does not return a value, a **`return`** statement is used without an expression:

```
return;
```


Method Definitions

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the **`typeReturned`** can be used:

```
typeReturned tRVariable;
tRVariable = objectName.methodName();
```

- An invocation of a **`void`** method is simply a statement:

```
objectName.methodName();
```


Any Method Can Be Used As a **`void`** Method

- A method that returns a value can also perform an action
- If you want the action performed, but do not need the returned value, you can invoke the method as if it were a **`void`** method, and the returned value will be discarded:

```
objectName.returnedValueMethod();
```


Local Variables

- A variable declared within a method definition is called a *local variable*
  - All variables declared in the **`main`** method are local variables
  - All method parameters are local variables
- If two methods each have a local variable of the same name, they are still two entirely different variables

Global Variables

- Some programming languages include another kind of variable called a *global* variable
- The Java language does **not** have global variables

Blocks

- A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, **{}**
- A variable declared within a block is local to that block, and cannot be used outside the block
- Once a variable has been declared within a block, its name cannot be used for anything else within the same method definition

Declaring Variables in a `for` Statement

- You can declare one or more variables within the initialization portion of a **`for`** statement
- A variable so declared will be local to the **`for`** loop, and cannot be used outside of the loop
- If you need to use such a variable outside of a loop, then declare it outside the loop

Parameters of a Primitive Type

- The methods seen so far have had no parameters, indicated by an empty set of parentheses in the method heading
- Some methods need to receive additional data via a list of *parameters* in order to perform their work
  - These *parameters* are also called *formal parameters*

Parameters of a Primitive Type

- A parameter list provides a description of the data required by a method
  - It indicates the number and types of data pieces needed, the order in which they must be given, and the local name for these pieces as used in the method

```
public double myMethod(int p1, int p2, double p3)
```

Parameters of a Primitive Type

- When a method is invoked, the appropriate values must be passed to the method in the form of *arguments*
  - Arguments are also called *actual parameters*
- The number and order of the arguments must exactly match that of the parameter list
- The type of each argument must be compatible with the type of the corresponding parameter

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```


Parameters of a Primitive Type

- In the preceding example, the value of each argument (not the variable name) is plugged into the corresponding method parameter
  - This method of plugging in arguments for formal parameters is known as the *call-by-value mechanism*

Parameters of a Primitive Type

- If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion
  - In the preceding example, the `int` value of argument `c` would be cast to a `double`
  - A primitive argument can be automatically type cast from any of the following types, to any of the types that appear to its right:

    `byte`→`short`→`int`→`long`→`float`→`double`

    `char` ↗

Parameters of a Primitive Type

- A parameters is often thought of as a blank or placeholder that is filled in by the value of its corresponding argument
- However, a parameter is more than that: it is actually a local variable
- When a method is invoked, the value of its argument is computed, and the corresponding parameter (i.e., local variable) is initialized to this value
- Even if the value of a formal parameter is changed within a method (i.e., it is used as a local variable) the value of the argument cannot be changed

A Formal Parameter Used as a Local Variable (Part 1 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `Bill.java`.

```
1  import java.util.Scanner;

2  public class Bill
3  {
4      public static double RATE = 150.00; //Dollars per quarter hour

5      private int hours;
6      private int minutes;
7      private double fee;
```

(continued)

A Formal Parameter Used as a Local Variable (Part 2 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

```
8      public void inputTimeWorked()
9      {
10         System.out.println("Enter number of full hours worked");
11         System.out.println("followed by number of minutes:");
12         Scanner keyboard = new Scanner(System.in);
13         hours = keyboard.nextInt();
14         minutes = keyboard.nextInt();
15     }

16     public double computeFee(int hoursWorked, int minutesWorked)
17     {
18         minutesWorked = hoursWorked*60 + minutesWorked;
19         int quarterHours = minutesWorked/15; //Any remaining fraction of a
20                                              // quarter hour is not charged for.
21         return quarterHours*RATE;
22     }

23     public void updateFee()
24     {
25         fee = computeFee(hours, minutes);
26     }
```

computeFee uses the parameter `minutesWorked` *as a local variable.*

Although `minutes` *is plugged in for* `minutesWorked` *and* `minutesWorked` *is changed, the value of* `minutes` *is not changed.*

(continued)

A Formal Parameter Used as a Local Variable (Part 3 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

```
27     public void outputBill()
28     {
29         System.out.println("Time worked: ");
30         System.out.println(hours + " hours and " + minutes + " minutes");
31         System.out.println("Rate: $" + RATE + " per quarter hour.");
32         System.out.println("Amount due: $" + fee);
33     }
34 }
```

(continued)

A Formal Parameter Used as a Local Variable (Part 4 of 5)

Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `BillingDialog.java`.

```
1  public class BillingDialog
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Welcome to the law offices of");
6          System.out.println("Dewey, Cheatham, and Howe.");
7          Bill yourBill = new Bill();
8          yourBill.inputTimeWorked();
9          yourBill.updateFee();
10         yourBill.outputBill();
11         System.out.println("We have placed a lien on your house.");
12         System.out.println("It has been our pleasure to serve you.");
13     }
14 }
```

(continued)

A Formal Parameter Used as a Local Variable (Part 5 of 5)

Display 4.6 **A Formal Parameter Used as a Local Variable**

SAMPLE DIALOGUE

```
Welcome to the law offices of
Dewey, Cheatham, and Howe.
Enter number of full hours worked
followed by number of minutes:
3 48
Time worked:
2 hours and 48 minutes
Rate: $150.0 per quarter hour.
Amount due: $2250.0
We have placed a lien on your house.
It has been our pleasure to serve you.
```


Pitfall: Use of the Terms "Parameter" and "Argument"

- Do not be surprised to find that people often use the terms parameter and argument interchangeably
- When you see these terms, you may have to determine their exact meaning from context

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`

The `this` Parameter

- `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
```

local — instance

The `this` Parameter

- The **`this`** parameter is a kind of hidden parameter
- Even though it does not appear on the parameter list of a method, it is still a parameter
- When a method is invoked, the calling object is automatically plugged in for **`this`**

Methods That Return a Boolean Value

- An invocation of a method that returns a value of type **`boolean`** returns either **`true`** or **`false`**
- Therefore, it is common practice to use an invocation of such a method to control statements and loops where a Boolean expression is expected
  - **`if-else`** statements, **`while`** loops, etc.

The methods `equals` and `toString`

- Java expects certain methods, such as **`equals`** and **`toString`**, to be in all, or almost all, classes
- The purpose of **`equals`**, a **`boolean`** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **`==`** to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the **`toString`** method is to return a **`String`** value that represents the data in the object

```
public String toString()
```


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`

A Couple of Important Acronyms: API and ADT

- The API or *application programming interface* for a class is a description of how to use the class
  - A programmer need only read the API in order to use a well designed class
- An ADT or *abstract data type* is a data type that is written using good information-hiding techniques

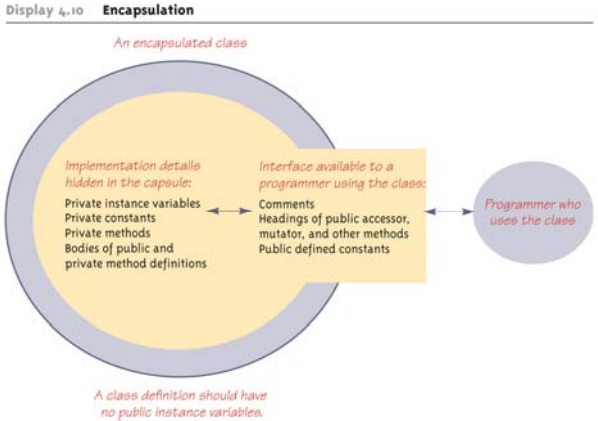
`public` and `private` Modifiers

- The modifier `public` means that there are no restrictions on where an instance variable or method can be used
- The modifier `private` means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make **all** instance variables `private`
- Most methods are `public`, and thus provide controlled access to the object
- Usually, methods are `private` only if used as helping methods for other methods in the class

Accessor and Mutator Methods

- *Accessor* methods 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 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`

Encapsulation



A Class Has Access to Private Members of All Objects of the Class

- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object

Mutator Methods Can Return a Boolean Value

- Some mutator methods issue an error message and end the program whenever they are given values that aren't sensible
- An alternative approach is to have the mutator test the values, but to never have it end the program
- Instead, have it return a boolean value, and have the calling program handle the cases where the changes do not make sense

Preconditions and Postconditions

- The *precondition* of a method states what is assumed to be true when the method is called
- The *postcondition* of a method states what will be true after the method is executed, as long as the precondition holds
- It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment

Overloading

- *Overloading* is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different *signatures*
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

Overloading and Automatic Type Conversion

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

Pitfall: You Can Not Overload Based on the Type Returned

- The signature of a method only includes the method name and its parameter types
  - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class

You Can Not Overload Operators in Java

- Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire

Constructors

- A *constructor* is a special kind of method that is designed to initialize the instance variables for an object:

```
public ClassName(anyParameters){code}
```

  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**
  - Constructors are typically overloaded

Constructors

- A constructor is called when an object of the class is created using **`new`**

```
ClassName objectName = new ClassName(anyArgs);
```

  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **`new`** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **`new`**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead

You Can Invoke Another Method in a Constructor

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

A Constructor Has a `this` Parameter

- Like any ordinary method, every constructor has a **`this`** parameter
- The **`this`** parameter can be used explicitly, but is more often understood to be there than written down
- The first action taken by a constructor is to automatically create an object with instance variables
- Then within the definition of a constructor, the **`this`** parameter refers to the object created by the constructor

Include a No-Argument Constructor

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, be sure to provide your own no-argument constructor as well

Default Variable Initializations

- Instance variables are automatically initialized in Java
  - **`boolean`** types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to **`null`**
- However, it is a better practice to explicitly initialize instance variables in a constructor
- Note: Local variables are not automatically initialized

The `StringTokenizer` Class

- The **`StringTokenizer`** class is used to recover the words or *tokens* in a multi-word **`String`**
  - You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
  - In order to use the **`StringTokenizer`** class, be sure to include the following at the start of the file:

```
import java.util.StringTokenizer;
```


Some Methods in the `StringTokenizer` Class (Part 1 of 2)

Display 4.17 Some Methods in the Class `StringTokenizer`

The class `StringTokenizer` is in the `java.util` package.

`public StringTokenizer(String theString)`

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

`public StringTokenizer(String theString, String delimiters)`

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

`public boolean hasMoreTokens()`

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

(continued)

Some Methods in the `StringTokenizer` Class (Part 2 of 2)

Display 4.17 Some Methods in the Class `StringTokenizer`

`public String nextToken()`

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[5]

`public String nextToken(String delimiters)`

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.
(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)[5]

`public int countTokens()`

Returns the number of tokens remaining to be returned by `nextToken`.