Module 2

Basic Concepts:
Properties of a Good Program

Adapted from Absolute Java, Rose Williams, *Binghamton University*

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

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

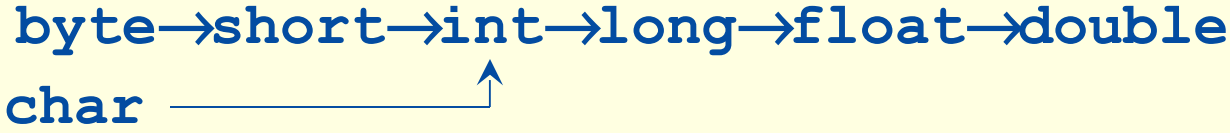
```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

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:



Parameters of a Primitive Type

- A parameter 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 (1/5)

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 (2/5)

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 (3/5)

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 (4/5)

Display 4.6 A Formal Parameter Used as a Local Variable

```
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 (5/5)

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.
```

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

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.

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

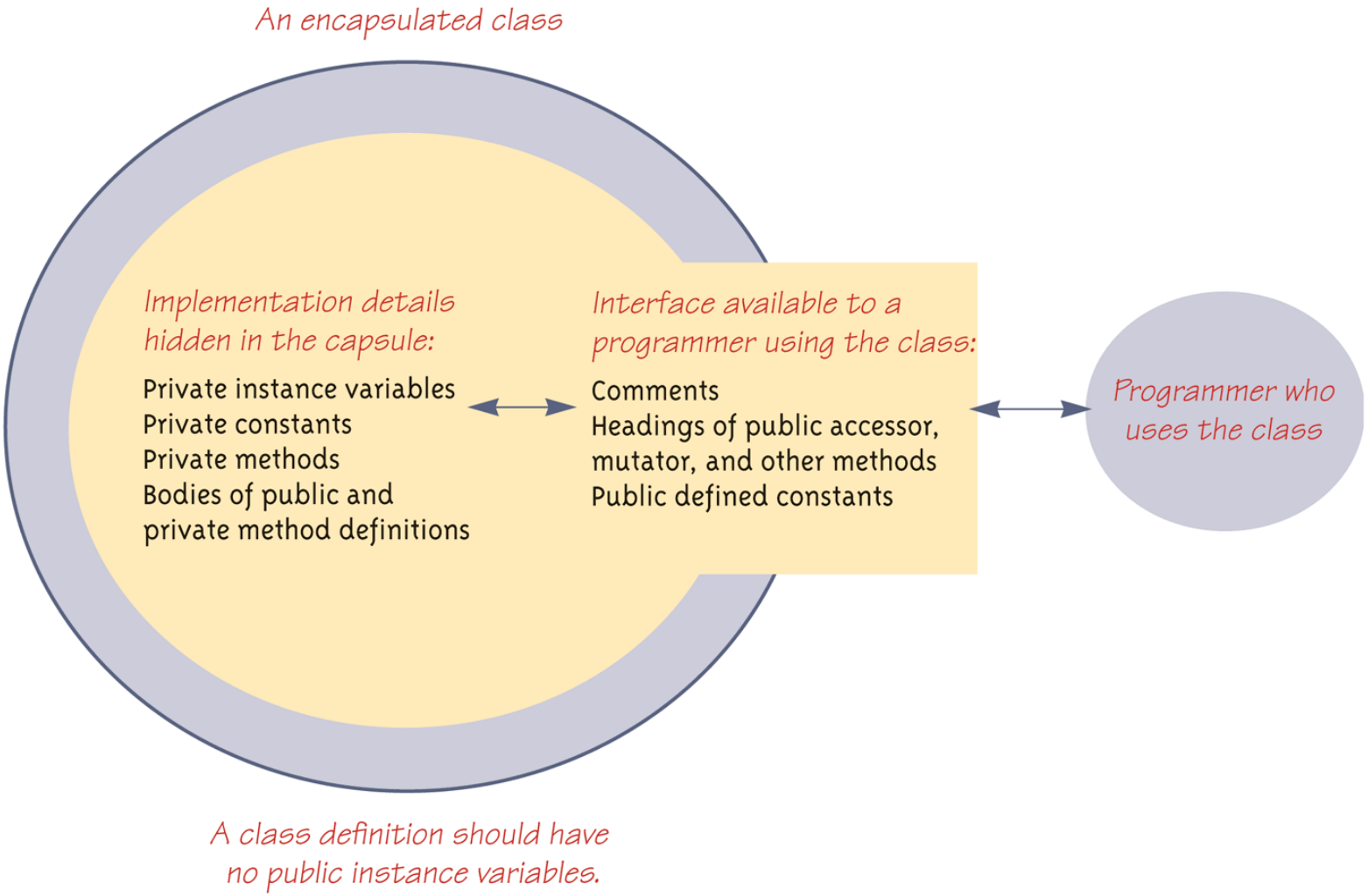
`public` and `private` Modifiers

- 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

```
this.firstPage = InChapter.firstPage;
```

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

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
  - The signature does **not** include the type returned
  - Java does not permit methods with the same signature and different return types in the same class

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

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

Importing Packages and Classes

- Libraries in Java are called *packages*
  - A package is a collection of classes that is stored in a manner that makes it easily accessible to any program
  - In order to use a class that belongs to a package, the class must be brought into a program using an *import* statement
  - Classes found in the package **`java.lang`** are imported automatically into every Java program

```
import java.util.StringTokenizer;
 // import StringTokenizer class only
import java.util.*;
 //import all the classes in package java.util
```