Chapter 4 – Part 1

Classes, Objects, and Methods

- Class and Method Definitions
- Parameter passing

Classes

- *Class*—the *abstract* definition for new objects
  - » similar to a template, design, or outline for constructing new, unique objects
  - » objects, properly referred to as "instances of a class"
- a class specifies the kind the attributes and methods objects of that class contain,
  - » objects of a class have similar data items (attributes) but with different values
  - » objects of a class have the same actions (methods), but the outcomes are unique to each object

Class as an Outline

Class Definition

Instantiations of the class def.

Class Name: Automobile
Data:
amount of fuel _____
speed _____
license plate _____
Methods (actions):
increaseSpeed:
How: Press on gas pedal.
stop:
How: Press on brake pedal.

First Instantiation:
Object name: patsCar
amount of fuel: 10 **gallons**
speed: 55 **miles per hour**
license plate: "135 XJK"

Second Instantiation:
Object name: suesCar
amount of fuel: 14 **gallons**
speed: 0 **miles per hour**
license plate: "SUES CAR"

Third Instantiation:
Object name: ronsCar
amount of fuel: 2 **gallons**
speed: 75 **miles per hour**
license plate: "351 WLF"

Objects

- objects are *named instances* of a class
  - » similar in concept to variables, except the class is the object's type
  - » also, objects are *reference variables* that indirectly reference the data stored by the "object"
- objects have both data and actions:
  - » data are called attributes, actions are called methods
  - » both attributes and methods are referred to as *members of the object* (belonging to the object)
    - – attributes are also called *instance variables*
    - – attributes can be *primitives vars.,* or *objects* themselves

Example: `String` Class

- **String** is a class
  - » it stores a sequence of individual characters
  - » its has a number of useful methods,
    - – **.length**(), **.charAt**(), **.indexOf**()
  - » each String object has its own data, example:

```
String s1="two words", s2="three nice words";

System.out.println( s1.length() );
System.out.println( s2.length() );
```


Class Definition Files

- each Java *class definition* should be a separate file
- common file rules apply:
  - » filename is **exactly** the same as the class
  - » the files ends in **.java** and must be compiled (**.class**)
    - – **.class** files are not related to class definitions
  - » file is in the same folder as the program using the class
- some class definitions are *static*
  - » *static classes* usually do not have instance declarations
  - » variables, constants, and methods are used directly
  - » for example: **Math** and **SavitchIn**
  - » *static classes are used to define utility classes*

Instantiating Objects

- Syntax:
  `class_Name instance_Name = new class_Name();`
  - » note the keyword **new**
- ex: the textbook defines a class named **SpeciesFirstTry**

```
//instantiate a SpeciesFirstTry object
SpeciesFirstTry speciesOfTheMonth =
                    new SpeciesFirstTry();
```

  - » this class has a *public instance variable* named **.name** that is accessed via the "dot operator"

```
SpeciesOfTheMonth.name = "Garfield";
```


The *Program* Class

- every Java program you have written begins with `public class Program_Class_Name`
  - » meaning *your program is actually a class definition!*
  - » after the program is compiled to byte code as a **.class** file, the Java interpreter (virtual machine) runs the program by first creating an *instance of the program,*
    - – an *executable object* of your *program class*
- the implication of all this is that more than one instance of your program can exist at one time, each instance with the *same instructions but different data*
- methods included in a *program class* (or *static class*) must also include the **static** keyboard

Defining Methods in a Class

- rather than consider on all types of class definitions, we will focus on *program classes*
  - – the same ideas will be applied to all class definitions
- in a program, methods are used to,
  - – segment, organise, or simplify code by "hiding" sections of code and referring to them with a single name
  - – encapsulate code that is repeated at multiple locations in the program (to reduce the overall program length)
  - – create useful routines that can be used like a servant for other sections of the program
  - – define the actions of a class, without burdening another program with "how" the action is performed

Methods and their Return Statements

- **methods** can be declared anywhere in a class, but
  - » placed after public member variable declarations, and after the **main()** method definition
- the purpose of any method is to accomplish an action and **return** a result
  - » methods are stoped, and a value returned, through the **return** statement; example,

```
// square_area() – return the area of a square
public int square_area(int length)
{
   int areacalc = length*length;  //calc area
   return (areacalc);             //return area
} // end of square_area()
```

method header

method body

method end

void Methods and ***null*-Returns**

- a method that returns no value, because
  - » it is used strictly for output (i.e., printing); or
  - » it returns data through a parameter (*discussed later*)
- a method that returns no value is called **void**, with this keyword used in the method header
  - – **void** methods can use a null-return (a return with no value) to end & exit the method

```
// display_gui() – output string to GUI dialog
public void display_gui(String outstr)
{
   JOptionPane.showMessageDialog(null,outstr);
   return;            // end, ** optional **
} // end of display_gui()
```


The `main` Method

- for a *program class* to solve a problem (rather than simply provide a definition for objects) is must contain a specially named method: **main()**
- when the Java interpreter attempts to execute a program class, it begins with the **main()** method
  - – if this method is missing, an exception error is displayed

```
public class Sample_Prog
{   ...declare member variables…
   public static void main(String[] args)
   {
      …statements that define the main method…
   } // end of main()
   ...define other methods for the class
} //end of class Sample_Prog
```

The `main` Method

- the **main()** method,
  - » returns no value and is therefore a **void**-method
    - – although a **return;** can be used to end the method
  - » has a single parameter: **(String args[ ]),** which is data provided to the method from the OS
    - – the **main()**'s header is a standard—**do not change**
  - » is sometimes included in "non-executable" class definitions (such as **String, SavitchIn,** and **Math**) so that the class can be tested during creation
    - – the **main()** would only have a sequence of tests

Locals, Globals, and Scope!

- "scope" – the description of *where* and *when* identifiers (variables, classes, objects, and methods) can be seen by statements
- "global" – any identifier declared *outside* of all methods and compound statements
  - – a "global" can be "seen" by the entire class
- "local" – any identifier declared *within* a method or compound statement block,
  - – a "local" can be "seen" by only its block, and blocks within that block
  - – locals do not exist (inaccessible) outside the block

Passing Values to a Method: Parameters

- methods are provided *data* in two (2) ways:
  - » accessing the value of a global-variable or –object
    - – not recommended, since the same variable is always used this limits the flexibility of the method
  - » by providing the method data via its **parameter list**
    - – recommended, since it makes the method flexible and useful, as well as allowing for use outside the class
- methods can only return a data value through the **return** statement
  - » methods can also return data through **pass-by-reference** parameters (*more on this later*)

Pass-by-Value: *Primitive* Types as Parameters

- rules for parameters,
  - » if defined as a *primitive type* the parameter is **passed-by-value**—a "copy" of the data is passed
  - » if defined as a *class type* the parameter is **passed-by-reference**—the "reference address" is passed
    - – *reference parameters can be used as* **method output***!*
  - » in calling a method, the *type* of the arguments must be the same as the *type* of the method parameters
    - – implicit and explicit casting rules apply
  - » parameters are initialised with the argument's value
  - » parameters are *local* to the method (*scope is enforced*)
  - » variables as arguments are not changed by the method

Pass-by-Value Example

```
//definition of method to double an integer
public static int doubleValue(int numberIn)
{
   return (2 * numberIn);
}// end of doubleValue()
```

```
//method call (somewhere in main() )
...
int next = SavitchIn.readLineInt();
System.out.println
     ("Twice next = " + doubleValue(next) );
```

- What is the *parameter* in the method definition?
  - » `numberIn`
- What is the *argument* in the method invocation (call)?
  - » `next`

Pass-by-Reference: *Class* Types as Parameters

- unlike variable parameters that act as "pass-by-value", *class parameters* copy the argument's ***address*** (not the value) to the parameter,
  - » remember: objects of *class* are reference variables
  - » the parameter stores (points) to the same address as the argument object
- the outcome is that within the method,
  actions taken on the parameter is actually performed on the original argument!
- this implication,
  - » the argument value is not protected for class types!
  - » the argument/parameter can be used for input and output
  - » extra memory is not used to duplicate the entire object

Pass-by-Reference

- **pass-by-reference** will be analysed in more depth when discussing "abstract data types" ADTs
- for the moment, just understand that there are two methods,
  - **pass-by-value**: pass the data
  - **pass-by-reference**: pass the address of the data
- although they are of the String class, String objects are special (they understand assignment (**=**) and work like a primitive type)
  - » pass-by-reference does not work as expected with Strings

```
//definition to capture String data from user
public static String getString()
{
   String inputvar= SavitchIn.readLine();
   return (inputvar);
}
```


Method Examples
in-class discussion

- Create a method that calculates the area of a triangle based on the *width* and *height* of the shape,
  - » start with the method call in **main()**
  - » the method should use **double** values
- Create a method that mathematically rounds a double value to exactly 3 decimal places
  - » the method has a single parameter and returns a single value
- Create a method that accepts 10 phrases (strings) from the user and displays them as a single output
  - » both input and output must be done via a GUI
  - » the method call accepts no arguments and returns no value