Methods and Data (Savitch, Chapter 5)

TOPICS

- Invoking Methods
- Return Values
- Local Variables
- Method Parameters
- Public versus Private
- Static Methods

Methods

- A **method** (a.k.a. function, procedure, routine) is a piece of code that performs a useful action.
  - Up to this point you have put the entire program in a method called *main*.
  - The *main* method is special only in that it provides an entry point when you start a Java program.
- Writing monolithic programs in the *main* method is too complex and hard to understand.
  - Instead programmers decompose programs into many methods, each of which is simple to understand.
  - Each method has its own set of **local** variables, more details on data in a minute

Mysteries Revealed

```
public class Temperature {
   public static void main(String[] args) {
      // your code here
   }
}
```

In our recitations and assignments, you define classes (e.g. P1, R1).

*You also define a method called **main** that takes an array of Strings as its arguments*

Method Types

- When you use a method you "invoke" or "call" it from another method.
- Two kinds of Java methods
  - Perform some action and return a single item
  - Perform some and return nothing
- The method `main` is a `void` method
  - Invoked by the system, not the application
  - Does not return anything

Calling Methods

- Calling a method that returns an item
  - For **void** method, invoke using object name:
    - *object.method();*
  - When method returns a value:
    - *int valueReturned = object.method();*
  - Use return value anywhere any other literal or variable value can be used
  - Return values can be of any data type:
    - Primitive types, objects, collections

Defining `void` Methods

- Consider method to print something, does not require any data:

```
public void printMethod()
   System.out.println("Student Name: ");
   System.out.println("Student EID: ");
}
```

- Method definitions reside in class definition
  - Can be called only on objects of that class

Method Declarations

- `public` methods can be called from inside or outside the class
- `private` methods can be called only from inside the class
- Data type specifies return type, `void` means no return value
- Heading includes parameters in `( )`
- Body enclosed in braces `{ }`

Return Values

- Consider method `getMonthsInYear( )`

```
public int getMonthsinYear()
{
     int monthsInYear = 12;
     return monthsInYear;
}
```

- Heading declares type of return value
- Last statement executed is `return`
- Parameter list is empty

Local Variables

- Variables declared inside a method are called *local* variables
  - May be used only inside the method
  - All variables declared in method **`main`** are local to **`main`**
- Local variables having the same name and declared in different methods are different variables

Local Example

```
public void sin(double angle {
     double value = Math.sin(angle);
     return value;
}

public void cos(double angle {
     double value = Math.cos(angle);
     return value;
}
```


Blocks

- Recall compound statements
  - Enclosed in braces **`{ }`**
- When you declare a variable within a compound statement
  - The compound statement is called a *block*
  - The scope of the variable is from its declaration to the end of the block
- Variable declared outside the block usable both outside and inside the block

Passing Parameters

- Method declaration must include a list of parameters and their types

```
public double sin(double angle) {
     return Math.sin(angle);
}
```

- Empty list means no parameters
- Parameters are separated by commas

Method Parameters

- Note the declaration

```
public int computeInterest(double rate)
```

  – The *formal* parameter is `rate`

- Calling the method

```
double interest = obj.computeInterest(5.9);
```

  – The *actual* parameter is 5.9

Primitive Parameters

- Parameter names are local to the method
- When method invoked
  – Each parameter initialized to value in corresponding actual parameter
  – Primitive actual parameter cannot be altered by invocation of the method
- Automatic type conversion performed

```
byte -> short -> int ->
      long -> float -> double
```


Method Examples

- `public double sin(double angle)`
- `public double cos(double angle)`
- `public char charAt(int index)`
- `public int indexOf(char c)`
- `public int minimum(int i, int j)`
- `public String toLower(String s)`
- `public int[] getArray()`

`public` and `private`

- `public`
  – Can access the class, method, or data by name outside defining class
- `private`
  – Can access the class, method, or data by name only inside defining class
- Classes generally specified as `public`
- Instance variables usually are `private`
- Methods can be `public` or `private`

Pre-condition Comment

- Precondition comment
  - States conditions that must be true before method is invoked
- Example

```
double squareRoot(double value) {
   // Parameter value must be > 0
   …
}
```


Post-condition Comment

- Postcondition comment
  - Tells what will be true after method executed
- Example

```
double squareRoot(double value) {
   …
   // Return value is sqrt(value)
}
```


Accessors and Mutators

- When instance variables are private must provide methods to access values stored there
  - Typically named **`getSomeValue`**
  - Referred to as an accessor method
- Must also provide methods to change the values of the private instance variable
  - Typically named **`setSomeValue`**
  - Referred to as a mutator method

Methods Calling Methods

- A method body may call any other method
- If the method is within the same class
  - Need not use prefix of receiving object
  - ***`this`*** keyword is assumed, represents the class we're currently in

Class Data

- Local data resides in a method, class or instance data resides in the class
- Data defined in the class can be of two types:
  - Data may belong to the class (and will take the same value for all the objects)
  - Data may belong to the object (and can take different values for each instance)
  - Objects of the former type must be marked as **static** to allow the compiler to differentiate

Class Data: Example

```
public class P1 {
    public int int0;
    private double real0;
    static String str0;
}
```

- *int0* visible outside of class, *real0* is not
- *str0* has an identical value for all instances

Static Methods

- Methods are called with an instantiated object of the type class:
  - The notation is **objectname.method()**
  - You must have a String variable called word to call word.length()
  - The length() method can access data in the instance it is called on
  - Such methods are called instance methods
- Exception: static methods can be called with only the class name, i.e. no instance:
  - The notation is **classname.method()**
  - Not all methods need to access data specific to objects
  - Static declares that a method will not access instance data
  - Static methods may access class data, but not instance data

public static void main

- Remember that magic incantation at the start of your program?
  - **main** is the name of your method
    - The main method is called by the OS at program startup.
  - **void** says that the main function does not return a value
    - What would the OS do with a return value?
  - **static** says that main will not access instance variables
    - Because the OS needs to call it without creating a class instance
  - **public** allows access to main outside the class
    - Again the OS needs this access to start the program

Static methods

- *main* is an example of a static method
- It can only access class variables (**static**)
- It cannot access instance variables. To use instance variables, we will have main create an instance of its class…
- But first, let ‘s see some static methods
  - First we will see static methods that don’t share data
  - Then we will see static methods that can share data

Simple example (main method calling snowService)

```
import java.util.Scanner;
public class SnowRemoval {
    public static void main(String[] args) {
        System.out.println("Enter your address:");
        Scanner keyboard = new Scanner(System.in);
        String address = keyboard.nextLine();
        int delay = snowService(address);
        if (delay==0)
            System.out.println("My driveway is safe now");
        else // assume status is non-negative
            System.out.println("I have to wait for " + delay + " hours");
    }
```


Simple example (snowService method)

```
    public static int snowService(String home){
        System.out.print("Clearing driveway of " + home);
        return 0;
    }
}
```


Communication between calling and called methods

- Method parameters:
  - Method declares a parameter *formal* parameter to state what can be provided by the calling program.

```
public String reverseCase(String s1)
```

  - Indicates the calling program must specify a String

```
public int returnRandom()
```

  - Indicates the calling program specifies no parameters

Communication between calling and called methods

- Method return type and value:
  - Can return void (e.g. nothing)
  - Can return a type (e.g. int, double, char, String, …)
    - If a value is returned, there must be a return statement in the method body
    - There must be a return for each possible path through the code
  - Return type must match assignment in calling program

Communication between calling and called methods

```
public String reverseCase(String s1)
public int returnRandom()
```

- Calling method:
  - Supplies arguments that must match the type of the parameters in the method declaration
  - Uses the return value to do something
  - Return value must match type of variable

```
System.out.print(reverseCase(str));
int random = returnRandom();
```


Caution: Pass by value

- What do you expect this to print?

```
public class PassByValue {
  public static void main(String[] args) {
    int number = 100;
    increment(number);
    System.out.println("Number: " + number;
  }
  public static void increment(int n) {
     n++;
  }
}
```

- The value of the argument is copied, so no change to number!

Incorrect Swapping

```
public class Swapper {
    public static void main(String[] args) {
        String s1 = "Martin";
        String s2 = "Scorcese";
        swap(s1, s2);
        System.out.println("main: After swap, s1=" +s1+ " and s2=" +s2);
    }
    public static void swap(String x, String y) {
        System.out.println("swap: Before swap, x=" +x+ " and y=" +y);
        String temp = x; x = y; y = temp;
        System.out.println("swap: After swap, x=" +x+ " and y=" +y);
    }
}
```

- Nothing gets swapped!

Use methods for subtasks

- The general rule is:
  - Break subtasks into tasks until tasks are trivial
  - Every subtask is a method
  - Some methods (subtasks) may call others

Methods inside a class

- Order of writing methods is arbitrary
  - Generally constructors are written first
- What if two methods need to share data?
  - One subtask reads input and creates a string of words separated by white space
  - Another subtask checks each word in the string one at a time

Solution #1

- Method 1 for subtask 1 returns a value
- Method 2 for subtask 2 uses the value
- Example:

```
public static void main(String[] args) {
    String wordList = readInput();
    processWords(wordList);
}
```


Solution #2

- Use instance variables
  - Define `String wordList;` as an instance variable in the class
  - Any method of a class can access its variables
    - readInput() can create and write the string
    - processWords() can access the same string
  - Of course neither method can be **static** since the wordList is not static!