Java Methods

Lecture 9
CGS 3416 Spring 2017

February 15, 2017

Java Methods

- In Java, the word **method** refers to the same kind of thing that the word **function** is used for in other languages.
- Specifically, a method is a function that belongs to a class.
- In Java, every function belongs to a class.
- A function is a reusable portion of a program, sometimes called a *procedure* or *subroutine.*
- The properties of a method are:
  - It is like a mini-program (or subprogram) in its own right.
  - Can take in special inputs (arguments).
  - Can produce an answer value (return value).
  - Similar to the idea of a *function* in mathematics.

Why write and use functions?

- Divide-and-conquer
  - Breaking up programs and algorithms into smaller, more manageable pieces
  - This makes for easier writing, testing, and debugging
  - Also easier to break up the work for team development
- Reusability
  - Functions can be called to do their tasks anywhere in a program, as many times as needed
  - Avoids repetition of code in a program
  - Functions can be placed into libraries to be used by more than one "program"
- With methods (functions), there are 2 major points of view
  - **Builder** of the method – responsible for creating the *declaration* and the *definition* of the method (i.e. how it works)
  - **Caller** – somebody (i.e. some portion of code) that uses the method to perform a task

Using Methods

- The user of a method is the caller.
- Use a method by making calls to the method with real data, and getting back real answers.
- Consider a typical function from mathematics:

  `f(x) = 2x + 5`

- In mathematics, the symbol 'x' is a placeholder, and when you run the function for a value, you "plug in" the value in place of x. Consider the following equation, which we then simplify:

```
y = f(10)              // must evaluate f(10)
y = 2 * 10 + 5         // plug in 10 for x
y = 20 + 5
y = 25               // so f(10) results in 25
```

- In programming, we would say that the call f(10) returns the value 25.

Using Methods

- Java methods work in largely the same way. General format of a Java method call:

```
methodName(argumentList)
```

- The argumentList is a comma-separated list of arguments (data being sent into the method). Use the call anywhere that the returned answer would make sense.
- When calling a Java method from another class library, we have to precede the call with the object name or the class name, depending on whether the method is static or not:

```
className.methodName(argumentList)
        // for static methods
objectName.methodName(argumentList)
        // for instance methods
```

- If a method is a member of the same class from which it is called

Example

There is a pre-defined Math class method called sqrt, which takes one input value (of type double) and returns its square root. Sample calls:

```
double x = 9.0, y = 16.0, z;

z = Math.sqrt(36.0); //returns 6.0 (stored in z)
z = Math.sqrt(x); //returns 3.0 (stored in z)
z = Math.sqrt(x + y); //returns 5.0(stored in z)

System.out.print(Math.sqrt(100.0));
     //returns 10.0, which gets printed
System.out.print(Math.sqrt(49));
     //due to automatic type conversion rules

System.out.print( Math.sqrt(Math.sqrt(625.0)));
```

A special use of import for static methods

- There is a special use of the keyword `static` for use in import statements.
- On an `import` statement, a programmer can import the static methods of a class, so that the class name and dot-operator does not have to be used in subsequent calls in the file.
- For example, suppose we do this statement in our file:

```
import static java.lang.Math.sqrt;
```

- The above would mean that anywhere in the file we call the sqrt method, it's specifically the one from the Math class.
- In this case, we would not need to use the `Math.` syntax before each call.

A special use of import for static methods

- To import all static methods from a class this way, use the * wildcard character.
- For example:

```
import static java.lang.Math.*;
    // import all static methods from Math
```

- It's best to use this sparingly.
- If a code file is using multiple libraries, it can get confusing what class different method calls are coming from, especially if multiple classes have similarly named methods.

Building Methods

- The **builder** of a method (a programmer) is responsible for the **prototype** (or signature) of a method, as well as the **definition** (i.e. how it works)
- The structure of a method:

```
modifier(s) returnType methodName(parameter list)
          // this is the signature
{
     // method body (i.e.  what it does,
            how it works) -- the definition
}
```

Building Methods

The pieces:

- **methodName** - identifier chosen by the builder.
- **parameter list** - a comma-separated list of the parameters that the method will receive.
  - This is data passed IN to the method by the caller.
  - The parameter list indicates the types, order, and number of parameters.
- **returnType** - the data type of the value returned by the method. A method that returns no value should have return type void.
- **modifier(s)** - optional labels that can specify certain properties or restrictions on the method.
  - For now, we will use the modifier static on our methods.
- **method body** - code statements that make up the definition of the method and describe what the method does, how it works.

Returning values

- To return a value (from the body of a method with a non-void return type), use the keyword **return**, followed by an expression that matches the expected return type:

```
return expression;
```

- A return statement will force immediate exit from the method, and it will return the value of the expression to the caller.
- A method with a non-void return type needs to return an appropriate value.

Method Examples

Here are two simple methods that do a math calculation and return a result

- ```
  public static int sum(int x, int y, int z)
  // add the 3 parameters and return the result
  {
        int answer;
        answer = x + y + z;
        return answer;
  }
  ```
- ```
  public static double average (double a, double b,
                                double c)
  //add parameters, divide by 3, return the result
  {
        return (a + b + c) / 3.0;
  }
  ```

More Examples

More than one return statement may appear in a function definition, but the first one to execute will force immediate exit from the function.

```
boolean InOrder(int x, int y, int z)
// answers yes/no to the question "are these parameters in
order,
// smallest to largest?" Returns true for yes, false for
no.
{
     if (x <= y && y <= z)
          return true;
     else
          return false;
}
```

Some common mistakes

Examples of ILLEGAL syntax (common mistakes to watch out for):

- ```
  double average(double x, y, z){ }
              // Each parameter must list a type
  ```
- ```
  printData(int x){ }
              // missing return type
  ```
- ```
  int double Task(int x) { }
              // only one return type allowed!
  ```
  ```

Scope of Identifiers

- The scope of an identifier (i.e. variable) is the portion of the code where it is valid and usable
- A variable declared within a block (i.e. a compound statement) of normal executable code has scope **only within that block**.
  - Includes method bodies
  - Includes other blocks nested inside methods (like loops, if-statements, etc)
  - Does not include some special uses of block notation to be seen later (like the declaration of a class – which will have a separate scope issue)
- Variables declared in the formal parameter list of a method definition have scope **only within that method**.
  - These are considered **local variables** to the method.
  - Variables declared completely inside the method body (i.e. the block) are also local variables.

void methods and empty parameter lists

- Parameter lists
  - Mathematical functions must have 1 or more parameters
  - Java methods can have **0 or more** parameters
  - To define a method with no parameters, leave the parentheses empty
  - Same goes for the call. (But parintheses must be present, to identify it as a method call).
- Return types
  - A mathematical function must return exactly 1 answer
  - A Java method can return **0 or 1** return value
  - To declare a method that returns no answer, use void as the return type
  - A void method can still use the keyword return inside, but not with an expression (only by itself). One might do this to force early exit from a method.
  - To CALL a void method, call it by itself – do NOT put it in the middle of any other statement or expression

Sample Prototype

Here are some sample method prototypes:

```
char getALetter()      // no parameters

void printQuotient(int x, int y) // void return type

void killSomeTime()      // both
```

Functions and the compiler

- The compiler will check all method CALLS to make sure they match the expectations (which are described in the method signature)
  - method name must match
  - arguments passed in a call must match expected types and order
  - returned value must not be used illegally
  - static methods can be called through class name, but instance methods only through an object
- Decisions about parameters and returns are based on type-checking.
  - legal automatic type conversions apply when passing arguments into a method, and when checking what is returned against the expected return type

Pass By Value

- Default mode of passing parameters into methods
- Means that the parameter inside the method body is a copy of the original argument that was passed in
- Changes to the local parameter only affect the local copy, not the original argument in the call

```
static int myMethod(int x, int y)
{
      x = x * 2;
      System.out.println("x = " + x);
      y = y * 2;
      System.out.println("y = " + y);
      return x + y;
}
```

Pass by value

- Sample call:

```
int a = 5, b = 8, ans;
ans = myMethod(a, b);
System.out.println("ans = " + ans);
System.out.println("a = " + a);
System.out.println("b = " + b);
```

- Notice that the output of the code is:

```
x = 10
y = 16
ans = 26
a = 5
b = 8
```

Method Overloading

- The term method overloading refers to the fact that it is perfectly legal to have more than one method in the same class **with the same name**, as long as they have different parameter lists.
- The difference can be in the number of parameters, or in the types of parameters.
- Example:

```
int process(double num) { } // method 1
int process(char letter) { } // method 2
int process(int num, int pos) { } //method 3
```

Method Overloading

- Notice that although all three methods above have the same exact name, they each have a different parameter list.
- Some of them differ in the number of parameters (2 parameters vs. 1 parameter), and the first two differ in types (double vs. char).
- The compiler will distinguish which function to invoke based on what is actually passed in when the function is called.

```
x = process(3.45,12); //invokes the third function

x = process('f'); // invokes the second function
```

Ambiguous Invocation

- Because of method overloading and the legality of some automatic type conversions, it is possible to make a call that could match two methods (due to the type conversion issue). This will result in a compiler error.
- Example:

```
double sum(int x, double y);
double sum(double x, int y);
```

- This pair of methods is legal, since the parameter lists differ. But the following is an illegal call, due to ambiguous invocation:

```
System.out.print("The sum is " + sum(3, 4));
```