Reminder: Lecture slides

- Lecture slides are posted online and are available before class
- Use these during class and outside of class!
- If you can, download the lecture before class so you can take notes on the slides

Lecture 3
Introduction to Parameters / Math

This Lecture:

1. Mathematical functions in Java
2. Defining more complicated methods with inputs and outputs
3. The constructor
4. Creating instances of a class

Defining Methods

- We know how to define simple methods
- Today, we’ll define more complicated methods that have both inputs and outputs
- Along the way, you’ll learn the basics of manipulating numbers in Java
We will define a class that models a basic Calculator.

Each of the Calculator’s methods will have inputs (numbers) and an output (numeric answer).

First, we'll talk about numbers and mathematical expressions in Java.

An integer is a whole number, positive or negative, including 0.

Depending on size of the integer, we can use one of four numerical base types (primitive Java data types): byte, short, int, and long, in order of number of bits of precision.

Bit: binary digit, 0 or 1

Integers

<table>
<thead>
<tr>
<th>Base Type</th>
<th>Size</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>128 (-2^7)</td>
<td>127 (2^7 - 1)</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>-32,768 (-2^15)</td>
<td>32,767 (2^15 - 1)</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2,147,483,648 (-2^31)</td>
<td>2,147,483,647 (2^31 - 1)</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>-9,223,372,036,854 (-2^63)</td>
<td>9,223,372,036,857 (2^63 - 1)</td>
</tr>
</tbody>
</table>

In CS15, you will almost always use int – good range and we’re not as memory-starved as we used to be.
Floating Point Numbers

- Sometimes, need more precision than integers can provide
- How to represent \( \pi = 3.14159\ldots \)?
- We use floating point numbers for a more precise representation
  - called “floating point” because decimal point can “float” - no fixed number of digits before and after it -- historical nomenclature
  - used for representing numbers in “scientific notation”, with decimal point and exponent, e.g., \( 4.3 \times 10^5 \)
- Two numerical base types in Java represent floating point numbers: float and double

<table>
<thead>
<tr>
<th>Base Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
</tr>
</tbody>
</table>

Feel free to use both in CS15. Use of double is more common in modern Java code, as we are not as memory-starved as we used to be.

Operators and Math Expressions (1/2)

- Example expressions:
  - \( 4 + 5 \)
  - \( 3.33 \times 3 \)
  - \( 10 \% 4 \)
  - \( 3.0 / 2.0 \)
  - \( 3 / 2 \)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
</tr>
</tbody>
</table>

Operators and Math Expressions (2/2)

- Example expressions:
  - \( 4 + 5 \rightarrow 9 \)
  - \( 3.33 \times 3 \rightarrow 9.99 \)
  - \( 10 \% 4 \rightarrow 2 \)
  - \( 3.0 / 2.0 \rightarrow 1.50 \)
  - \( 3 / 2 \rightarrow 1 \)

- What does each of these expressions evaluate to?
- why???
Be careful with integer division!

- When dividing two integers, result is truncated to an integer
  - $3 / 2 \rightarrow 1$
  - $3.0 / 2 \rightarrow 1.50$
- $3 / 2$ evaluates to 1
- If either number involved is floating point, result is also floating point: allows greater precision
  - $3.0 / 2.0 \rightarrow 1.50$

Evaluating Math Expressions

- Java follows same evaluation rules that you learned in math class years ago - PEMDAS
  - $2 + 4 * 3 - 7 \rightarrow 7$
- Evaluation takes place left to right, except:
  - expressions in parentheses evaluated first, starting at the innermost level
  - $2 + 3 + (11 / 12) \rightarrow 5$
  - operators evaluated in order of precedence/priority (* has priority over +)
  - $3 + (2 - (6 / 3)) \rightarrow 3$

Calculator

- Calculators should be able to add, subtract, multiply, and divide
- When we tell a Calculator to add two numbers, want it to add them and then tell us the answer
- To do this, we need to learn how to write a method that returns a value - in this case, a number

Return Type

- The return type of a method is the kind of data it gives back to whoever called it
- So far, we’ve seen return type void
- A method with a return type of void doesn’t give back anything when it’s done executing
- void just means “this method doesn’t return anything.”
Return Type

- If we want a method to return something, we replace `void` with the type of thing we want to return.
- If method should return an integer, specify `int` as the return type.
- When return type is not `void`, we’ve promised to end the method with a `return` statement.
- Any code following the `return` statement will not be executed.

A silly example:

```java
public int giveMeTwo() {
    return 2;
}
```

Return statements always take the form:

```
return <something of specified return type>;
```

Calculator (1/6)

```
public class Calculator {
    /* Some code elided */

    public int giveMeTwo() {
        return 2;
    }
    /* This is a return statement. */

    public int addFiveAndSeven() {
        return 5 + 7;
    }
}
```

- Let’s write another silly method for Calculator called `addFiveAndSeven` that adds two particular numbers and returns the result.
- It will return the value of “5+7” to whoever called it.
- We’ll generalize this silly example soon…

Calculator (2/6)

- What does it mean for method to “return a value to whoever calls it?”
- Another object can call `addFiveAndSeven` on a Calculator from somewhere else in our program and use the result.
- For example, consider a MathStudent class that has a Calculator named `myCalculator`.
- We’ll demonstrate how the MathStudent can call the method and use the result.
/* Somewhere in the MathStudent class lives an instance of the Calculator class named myCalculator */
myCalculator.addFiveAndSeven();

- We start by just calling the method
- This is fine, but we're not doing anything with the result!
- Let's use the returned value by printing it to the console

// myCalculator.addFiveAndSeven();
System.out.println(myCalculator.addFiveAndSeven());

public class Calculator {
    /* Some code elided */
    public int addFiveAndSeven() {
        return 5 + 7;
    }
}

- We've provided the expression myCalculator.addFiveAndSeven() to be printed to the console
- This information we give to the println method is called an argument; more on this in a few slides
- Putting one method call inside another is called nesting of method calls; more examples later

- When this line of code is evaluated:
  - First, addFiveAndSeven is called on myCalculator, returning the number 12
  - Next, println is called on System.out, and 12 is printed to the console
Aside: `System.out.println`

- `System.out.println` is an awesome tool for testing and debugging your code – learn to use it!
- Lets you see what's happening in your code by printing out values as it executes.
- If `Calculator` program isn't behaving properly, can test whether method `addFiveAndSeven` is the problem by printing its return value to verify that it's "12" (yes, obvious in this trivial case, but not in general!)

Calculator: a generic add method

- Now `Calculator` can add two specific numbers – but that's not too useful.
- For a functional `Calculator`, we'd have to define new `add` method for every distinct pair of numbers.

```java
public class Calculator {
    public int addFiveAndSeven() {
        return 5 + 7;
    }
    public int addTwoAndThree() {
        return 2 + 3;
    }
    public int addSixAndEight() {
        return 6 + 8;
    }
    // ...
}
```

Can do much better – let's design a generic method that will add any two numbers we want.

- Want a method that corresponds to algebraic function:
  \[ f(x, y) = x + y \]

```java
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}
```

Mathematical function: \[ f(x, y) = x + y \]  
Equivalent Java method:

```java
public int add(int x, int y) {
    return x + y;
}
```
Parameters (1/5)

- General form of a method you are defining that takes in parameters:
  ```java
  <visibility> <returnType> <methodName>(<type1> <name1>, <type2> <name2>...) {
    <body of method>
  }
  ```

- Parameters are specified as comma-separated list
  - For each parameter, specify type (e.g., `int`, `double`), and then `name` (e.g., `x`, `y`, `banana`, whatever you want)

- In algebra, don't specify type because context makes clear what kind of number we want. In programming use many different types, and must tell Java explicitly what we intend
  - Java is a "strictly typed" language, i.e., makes sure the user of a method passes the right number of parameters of the specified type in the right order – (not) compiler error!

Parameters (2/5)

- Name of each parameter is almost completely up to you
  - Naming restriction needs to start with a letter
  - Refer to CS15 style guide for naming conventions

- It's the name by which you'll refer to the parameter throughout the method

- The following methods are completely equivalent:
Parameters (3/5)

- Name of each parameter is almost completely up to you
  - Naming restrictions: no alphanumeric
  - Refer to CS15 style guide for naming conventions
- It’s the name by which you’ll refer to the parameter throughout the method
- The following methods are completely equivalent:

```java
public int add(int first, int second) {
    return first + second;
}
```

Parameters (4/5)

- Name of each parameter is almost completely up to you
  - Naming restrictions: no alphanumeric
  - Refer to CS15 style guide for naming conventions
- It’s the name by which you’ll refer to the parameter throughout the method
- The following methods are completely equivalent:

```java
public int add(int bert, int ernie) {
    return bert + ernie;
}
```

Parameters (5/5)

- Remember Robot class from last lecture?
- Its moveForward method took as a parameter— an int named numberOfSteps
- Follows same parameter format: type, then name

```java
/* within Robot class definition */
public void moveForward(int numberOfSteps) {
    // code that moves the robot
    // forward goes here!
}
```

With great power comes great responsibility...

- Try to come up with descriptive names for parameters that make their purpose clear to anyone reading your code
- Robot’s moveForward method calls its parameter “numberOfSteps”, not “x” or “thingy”
- We used “x” and “y” for simple, generic add method, but for anything more complicated, avoid single-letter names
Let's give `Calculator` class more functionality by defining `subtract`, `multiply`, and `divide` methods!

```java
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
    // Your code for subtract goes here!
    // …
    // Your code for multiply goes here!
    // …
    // Your code for divide goes here!
    // …
}
```

We arbitrarily chose "x" and "y" for names of each method's parameters.

Now that we've defined `add`, `subtract`, `multiply`, and `divide` methods, can call them on any `Calculator`.

Want to call `add` method and tell it which two numbers to add.

How do we call a method that takes in parameters?
Calling Methods with Parameters

- You already know how to call a method that takes in one parameter!
- Remember `moveForward`?

```java
public void moveForward(int numberOfSteps) {
    // code that moves the robot
    // forward goes here!
}
```

When we call a method, we pass it any extra piece of information it needs as an argument within the parentheses.

```java
public class RobotMover {
    /* additional code elided */
    public void moveRobot(Robot samBot) {
        samBot.moveForward(4);
        samBot.turnRight();
        samBot.moveForward(1);
        samBot.turnRight();
        samBot.moveForward(3);
    }
}
```

Arguments vs. Parameters

// within the Robot class
public void moveForward(int numberOfSteps) {
    // code that moves the robot
    // forward goes here!
}

- In defining a method, the parameter is the name by which a method refers to a piece of information passed to it – e.g. “x” and “y” in the function f(x, y) = x + y – is a “dummy name” determined by the definer.
- In calling a method, an argument is the actual value passed in – e.g. 2 and 3 in `add(2, 3)`

Calling Methods That Have Parameters

- When we call `samBot.moveForward(3)`, we're passing the number 3 as an argument.
  // in some other class...
  samBot.moveForward(3);

- When `moveForward` executes, its parameter is assigned the value of the argument that was passed in.

- That means `moveForward` here executes with `numberOfSteps = 3`.

```java
public class RobotMover {
    public void moveForward(int numberOfSteps) {
        // code that moves the robot
        // forward goes here!
    }
}
```
Calling Methods That Have Parameters

- When calling a method that takes in parameters, must provide a valid argument for each parameter.
- This means that the number and type of arguments must match the number and type of parameters: 1-to-1 correspondence.
- Order matters! The first argument you provide will correspond to the first parameter, the second to the second, etc.

Calling Methods That Have Parameters

- Each of Calculator's methods takes in two ints, which it refers to as x and y (no need or ability for user to know those names, also called identifiers).
- Whenever we call these methods, must provide two ints—first our desired value for first parameter, then desired value for second.

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
    public int subtract(int x, int y) {
        return x - y;
    }
    public int multiply(int x, int y) {
        return x * y;
    }
    public int divide(int x, int y) {
        return x / y;
    }
}
```

/* somewhere else in our code… */
myCalculator.add(2, 3);
myCalculator.multiply(6, 2);
myCalculator.divide(8, 4);
myCalculator.subtract(5, 3);

/* somewhere else in our code… */
myCalculator.add(2, 3);

Java does "parameter passing" by:
- First checking that the one-to-one correspondence is honored.
- Then substituting arguments for parameters.
- And finally executing the method body using the arguments.
Calling Methods That Have Parameters

```java
public int add(int x, int y) {
    return x + y;
}
```

- Java does "parameter passing" by:
  - first checking that the one-to-one correspondence is honored,
  - then substituting arguments for parameters,
  - and finally executing the method body using the arguments

Calling Methods That Have Parameters

```java
myCalculator.add(2, 3);
```

Calling Methods That Have Parameters

```java
myCalculator.add(2, 3);
```

- If we want to check the result returned from our method call, use `System.out.println` to print it to the console.
- We'll see the number 5 printed out!

DEMO

```
 System.out.println(myCalculator.add(2, 3));
```
Where did all these instances come from?

- We know how to send messages to an instance of a class by calling methods
- So far, we've called methods on `samBot`, an instance of `Robot`, and `myCalculator`, an instance of `Calculator`
- Where did those classes come from?
- Next: how to use a class as a blueprint to actually build instances!

Constructors (1/3)

- Calculators can add, subtract, multiply, and divide
- Can call any of these methods on any instance of `Calculator`
- But how did these instances get created in the first place?
- Define a special kind of method in the `Calculator` class: a constructor
- Note: every object must have a constructor

```java
public class Calculator {

    // this is the constructor!
    public Constructor() {
        // multiply and divide elided
    }

    public int add(int x, int y) {
        return x + y;
    }

    public int subtract(int x, int y) {
        return x - y;
    }
}
```

Constructors (2/3)

- A constructor is a special kind of method that is called whenever an object is to be "born", i.e., created – see shortly how it is called
- Constructor's name is always the same as name of class
- If the class is called "Calculator", its constructor needs to be called "Calculator". If the class is called "Dog", its constructor had better be called "Dog"

```java
public class Calculator {

    // this is the constructor!
    public Constructor() {
        // multiply and divide elided
    }

    public int add(int x, int y) {
        return x + y;
    }

    public int subtract(int x, int y) {
        return x - y;
    }
}
```

Constructors (3/3)

- Constructors are special methods: used only once, to create the instance
- And we never specify a return value in its declaration
- Constructor for `Calculator` does not take in any parameters (notice empty parentheses)
- Constructors can, and often do, take in parameters – stay tuned for next lecture
- Note: constructors always have a return value, and it is implicitly `this`
Instantiating Objects (1/3)

- Now that the Calculator class has a constructor, we can create instances of Calculator!
- Here’s how we create a Calculator in Java:
  ```java
  new Calculator();
  ```
- This means “use the Calculator class as a blueprint to create a new Calculator instance”
- Calculator() is a call to Calculator’s constructor, so any code in the constructor will be executed as soon as you create a Calculator.

Instantiating Objects (2/3)

- We refer to “creating” an object as instantiating it
- When we say: `new Calculator();`
- We’re creating an instance of the Calculator class, a.k.a. instantiating a new Calculator
- Where exactly does this code get executed?
- Stay tuned for the next lecture to see how this constructor is used by another instance to create a new Calculator!

Aside: Nesting

- Remember our samBot’s `moveForward` method?
  ```java
  public void moveForward(int numberOfSteps) {
      // code that moves robot forward
  }
  ```
- When we call `samBot.moveForward`, we need to tell her how many steps to move forward by:
  ```java
  samBot.moveForward(5);
  ```
- What if we want to calculate a number using our `myCalculator`, and then tell `samBot` to move forward by that much?

Aside: Nesting

- `myCalculator.add(int x, int y) returns an int`
- `samBot.moveForward` takes in an `int`
- We can nest `myCalculator's add(int x, int y)` method within `samBot's moveForward method`:
  ```java
  samBot.moveForward(myCalculator.add(5,7));
  ```
  ```java
  returns 12
  ```
  ```java
  samBot.moveForward(12);
  ```
- And `samBot` will move forward 12 steps!
**Important Points**

- Defining methods that take in **parameters** as input
- Defining methods that **return** something as an output
- Defining a **constructor** for a class
- Creating an instance of a class with the **new** keyword

**Announcements**

- Homework 1 and Andybot are released today on the course website
  - Homework 1 is due Sunday at 2PM
  - Andybot is due Tuesday 9/22 at 11:59PM
- Lab 0 is due at the end of next lab period (Thursday 9/23 at 11:59PM)
  - If you haven’t gotten checked off by a TA, come to TA hours before the deadline
  - You can add the lab calendar to your Google calendar
- Questions on homework or course logistics?
  - Sign up to Piazza at [http://piazza.com/brown/fall2015/cs15](http://piazza.com/brown/fall2015/cs15)
  - Submit your questions privately. TA hours are posted on the course website