Lecture 3
Introduction to Parameters / Math

- A class provides a blueprint for instances of that class
- Instances send each other messages
- Instances respond to a message via a method
- Format of messages is <instance>.<method>();
  - e.g., samBot.moveForward(3);
- Sometimes an instance wants to send a message to itself, using a method defined in its own class: this.<method>();
- this means "me, myself" AND the method is defined in this class
  - Choreographer tells dancer: dancer3.pirouette(2);
  - Dancer tells herself: this.pirouette(2);
- Note: we've not yet learned how to create new instances of any class
This Lecture:

- Mathematical functions in Java
- Defining more complicated methods with inputs and outputs
- The constructor
- Creating instances of a class
- Understanding Java flow of control

Defining Methods

- We know how to define simple methods
- Today, we will define more complicated methods that have both inputs and outputs
- Along the way, we will learn the basics of manipulating numbers in Java

BookstoreAccountant

- We will define a BookstoreAccountant class that models an employee in a bookstore, calculating certain costs:
  - finding the price of a purchase,
  - calculating change needed, etc.
- Each of the accountant’s methods will have inputs (numbers) and a single output (number)
First, we’ll talk about numbers and mathematical expressions in Java.

Integers
- An integer is a whole number, positive or negative, including 0.
- Depending on size (number of digits) of the integer, you can use one of four numerical base types (primitive Java data types): byte, short, int, and long, in increasing order of number of bits of precision.
- Bit: binary digit, 0 or 1.

<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>-9,223,372,036,854,775,808 ($2^{63}$)</td>
<td>9,223,372,036,854,775,807 ($2^{63} - 1$)</td>
</tr>
</tbody>
</table>

In CS15, almost always use int – good range and we’re not as memory-starved as we used to be so don’t need byte.
Floating Point Numbers

- Sometimes, need rational and irrational numbers, i.e., numbers with decimal points
- How to represent pi = 3.14159...?
- **Floating point numbers**
  - 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 x 10^5
- Two numerical base types in Java represent floating point numbers: `float` and `double`
Operators and Math Expressions (2/2)

- Example expressions:
  - $4 + 5 \rightarrow 9$
  - $3.33 \times 3 \rightarrow 9.99$
  - $11 \% 4 \rightarrow 3$
  - $3.0 / 2.0 \rightarrow 1.50$
  - $3 / 2 \rightarrow 1$

- What does each of these expressions evaluate to? why???

Be careful with integer division!

- When dividing two integer types, result is "rounded down" to an int after remainder is dropped
  - $3 / 2 \rightarrow 1$
- If either number involved is floating point, result is floating point: allows greater "precision," i.e., fractional portion.
  - $3.0 / 2 \rightarrow 1.50$
  - $3.0 / 2.0 \rightarrow 1.50$

- Java follows the same evaluation rules that you learned in math class years ago – PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction)
  - $2 + 4 \times 3 - 7 \rightarrow 7$
  - $(2 + 3) + (11 / 12) \rightarrow 5$
  - $3 + (2 - (6 / 3)) \rightarrow 3$
TopHat Question

What does x evaluate to?

```java
int x = (((5 / 2) * 3) + 5);
```

A. 12.5  
B. 11  
C. 13  
D. 10  
E. 12

BookstoreAccountant

- BookstoreAccountants should be able to find the price of a set of books.
- When we tell a BookstoreAccountant to calculate a price, we want it to perform the calculation 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.

```
public class Robot {
    public void turnRight() {
        // code that turns robot right
    }
    public void moveForward(int numberOfSteps) {
        // code that moves robot forward
    }
    public void turnLeft() {
        this.turnRight();
        this.turnRight();
        this.turnRight();
    }
}
```

- The return type of a method is the kind of data it gives back to whomever called it.
- So far, we have only 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 does not return anything.”

Return Type (1/2)
A silly example:

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

- If we want a method to return something, replace `void` with the type of thing we want to return.
- If method should return an integer, specify `int` return type.
- When return type is not `void`, we have promised to end the method with a `return` statement.

Return statements always take the form:

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

Let's write a silly method for `BookstoreAccountant` called `priceTenDollarBook()` that finds the cost of a $10 book.

- It will return the value "10" to whoever called it.
- "10" is an integer – it matches the return type, `int`!

What does it mean for a method to "return a value to whomever calls it"?

Another object can call `priceTenDollarBook()` on a `BookstoreAccountant` from somewhere else in our program and use the result.

For example, consider a `Bookstore` class that has an accountant named `myAccountant`.

We will demonstrate how the `Bookstore` can call the method and use the result.
Assume a Bookstore instance has created an instance of BookstoreAccount named myAccountant.

```java
myAccountant.priceTenDollarBook();
```

- We started by just calling `priceTenDollarBook()`.
- This is fine, it will return 10, but we are not doing anything with that result!
- Let's use the returned value by printing it to the terminal.

Aside: `System.out.println`

- `System.out.println()` is an awesome tool for testing and debugging your code — learn to use it!
- Helps the user see what is happening in your code by printing out values to the terminal as it executes.
- **NOT** equivalent to `return`, meaning other methods cannot see/use what is printed.
- If `Bookstore` program is not behaving properly, can test whether `priceTenDollarBook()` is the problem by printing its return value to verify that it is "10" (yes, obvious in this trivial case, but not in general!)

In a new method, `manageBooks()`, print result.

- "Printing" in this case means displaying a value to the user of the program.
- To print to terminal, we use `System.out.println(<expression to print>)`.
- `println()` method prints out value of expression you provide within the parentheses.

```java
public class BookstoreAccountant {
    /* Some code elided */
    public int priceTenDollarBook() {
        return 10;
    }
}
```
We have provided the expression `this.priceTenDollarBook()` to be printed to the console.

This information given 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.

```
public class BookstoreAccountant {
    /* Some code elided */
    public int priceTenDollarBook() {
        return 10;
    }
    public void manageBooks() {
        System.out.println(
            this.priceTenDollarBook());
    }
}
```

When this line of code is evaluated:
- `println()` is called with argument of `this.priceTenDollarBook()`.
- `priceTenDollarBook()` is called on this instance of the `BookstoreAccountant`, returning 10.
- `println()` gets 10 as an argument, 10 is printed to terminal.

Now your accountant can get the price of a ten-dollar book – but that's completely obvious.

For a functional bookstore, we'd need a separate method for each possible book price!

Instead, how about a generic method that finds the price of any number of copies of a book, given its price?

```
public class BookstoreAccountant {
    /* Some code elided */
    public int priceTenDollarBook() {
        return 10;
    }
    public void manageBooks() {
        System.out.println(
            this.priceTenDollarBook());
    }
    public int priceBooks(int numCps, int price) {
        // let's fill this in!
    }
}
```
Method answers the question: given a number of copies and a price per copy, how much do all of the copies cost together?

To put this in algebraic terms, we want a method that will correspond to the function:

\[ f(x, y) = x \times y \]

"x" represents the number of copies; "y" is the price per copy

```java
class BookstoreAccountant {
    public int priceTenDollarBook() {
        return 10;
    }
    public int priceBooks(int numCps, int price) {
        // let's fill this in!
    }
}
```

Mathematical function:

\[ f(x, y) = x \times y \]

Equivalent Java method:

```java
public int priceBooks(int numCps, int price) {
    return (numCps * price);
}
```

Method takes in two integers from caller and gives appropriate answers depending on those integers

When defining a method, extra pieces of information that the method needs to take in (specified inside the parentheses of the declaration) are called parameters

`priceBooks()` is declared to take in two parameters, "numCps" and "price" – these, like variable names, are arbitrary, i.e., your choice
Outline

- Mathematical functions in Java
- Defining more complicated methods with inputs and outputs
- The constructor
- Creating instances of a class
- Understanding Java flow of control

Parameters (1/3)

- General form of 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 lists of type-name pairs
  - For each parameter, specify type (for example, `int` or `double`), and then name ("x", "y", "banana"... whatever you want!)

- In basic algebra, we only deal with numbers and freely mix their types. In programming, we use many different types, not just numbers, but also class names, and must tell Java explicitly what we intend
  - Java is a "strictly typed" language, i.e., it makes sure the user of a method passes the right number of parameters of the specified type, in the right order— if not, compiler error! In short, the compiler checks for a strict one-to-one correspondence

Parameters (2/3)

- Dummy name of each parameter is completely up to you, but...
  - Java naming restriction: needs to start with a letter
  - Refer to CS15 style guide for naming conventions

- It is the name by which you will refer to the parameter throughout method

- Note again that each parameter is a pair: type and name

The following methods are completely equivalent:

```java
public int priceBooks(int numCps, int price) {
    return (numCps * price);
}
```

```java
public int priceBooks(int bookNum, int pr) {
    return (bookNum * pr);
}
```

```java
public int priceBooks(int a, int b) {
    return (a * b);
}
```
Remember the Robot class from last lecture?
Its `moveForward` method took in one 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!
}
```

We Want Human-readable Code

- 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 "numCps" and "price".
- Try to avoid single-letter names for anything that is not strictly mathematical; be more descriptive.

Accountant (1/2)

- Give the BookstoreAccountant class more functionality by defining more methods.
- Methods to calculate change needed or how many books a customer can afford.
- Each method will take in parameters, perform operations on them, and return an answer.
- We choose arbitrary but helpful parameter names.

```java
public class BookstoreAccountant {
    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
    // calculate a customer's change
    public int calcChange(int amtPaid, int price) {
        return (amtPaid - price);
    }
    // calculate max # of books (same price) u can buy
    public int calcMaxBks(int price, int myMoney) {
        return (myMoney / price);
    }
}
```
Accountant (2/2)

- `calcMaxBks` takes in price of a book (`price`) and an amount of money you have to spend (`myMoney`), tells you how many books you can buy.
- `calcMaxBks` works because when we divide 2 ints, Java rounds the result down to an int.
- Java always rounds down.
- $25 / $10 per book = 2 books

```java
public class BookstoreAccountant {
    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
    // calculates a customer's change
    public int calcChange(int amtPaid, int price) {
        return (amtPaid - price);
    }
    // calculates max # of books customer can buy
    public int calcMaxBks(int price, int myMoney) {
        return (myMoney / price);
    }
}
```

TopHat Question: Declaring Methods

We want a new method `getSalePrice` that returns an integer and takes in two parameters, one integer that represents the original price of a purchase and one integer that represents the percent discount offered. Which method declaration is correct?

- A. `public void getSalePrice() {` // code elided
- B. `public int getSalePrice(int price, int discount) {` // code elided
- C. `public int getSalePrice(price, discount) {` // code elided
- D. `public void getSalePrice(int price, int discount) {` // code elided

Calling (i.e., using) Methods with Parameters (1/3)

- Now that we defined `priceBooks()`, `calcChange()`, and `calcMaxBks()` methods, we can call them on any `BookstoreAccountant` instance.
- When we call `calcChange()` method, we must tell it the amount paid for the books and how much the books cost.
- How do we call a method that takes in parameters?
Calling Methods with Parameters (2/3)

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

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

Calling Methods with Parameters (3/3)

- When we call a method, we pass it any extra piece of information it needs as an argument within parentheses.
- When we call `moveForward` we must supply one `int` as argument:
  - `samBot.moveForward()` is NOT correct.
- Do NOT specify type of argument when calling a method:
  - `samBot.moveForward(4);` is NOT correct.

Arguments vs. Parameters

- In defining a method, the `parameter` is a dummy name picked by the author used by a method to refer to a piece of information passed into it, e.g. "x" and "y" in the function \( f(x, y) = x + y \).
- In calling a method, an `argument` is the actual value passed in, e.g. 2 and 3 in \( f(2, 3) \Rightarrow 5 \).
Calling Methods That Have Parameters (1/9)

- When we call `samBot.moveForward(3)`, we are passing 3 as an argument.
- When `moveForward()` executes, its parameter is assigned the value of argument that was passed in.
- Thus `moveForward()` here executes with `numberOfSteps = 3`.

- When calling a method that takes in parameters, must provide a valid argument for each parameter.
  - analogy: When each district selects 2 tributes to compete in the Hunger Games, they must be one male and one female, and from that district.
  - Means that number and type of arguments must match number and type of parameters.
  - One-to-one correspondence: same number of arguments, given in the same order, of the same matching type.

Calling Methods That Have Parameters (2/9)

- Each of our accountant's methods takes in two `int`s, which it refers to by different names (also called identifiers).
- Whenever we call these methods, must provide two `int`s – first, desired value for first parameter, then desired value for second.

Calling Methods That Have Parameters (3/9)

- public class BookstoreAccountant {
  public int priceBooks(int numCPs, int price) {
    return numCPs * price;
  }
  // calculates a customer's change
  public int calcChange(int amtPaid, int price) {
    return amtPaid - price;
  }
  // calculates max # of books you can buy
  public int calcMaxBks(int bookPr, int myMoney) {
    return myMoney / bookPr;
  }
}
Calling Methods That Have Parameters (4/9)

- Let’s go back to our instance of BookstoreAccountant named myAccountant.
- When we call a method on myAccountant, we provide a comma-separated list of arguments (in this case, ints) in parentheses.
- Those arguments are values we want the method to use for the first and second parameters when it runs.

```java
/* somewhere else in our code. */
myAccountant.priceBooks(2, 16);
myAccountant.calcChange(16, 12);
myAccountant.calcMaxBks(6, 33);
```

Calling Methods That Have Parameters (5/9)

- Note: `calcChange(8, 4)` isn't `calcChange(4, 8)` – order matters!
- `calcChange(8, 4) → 4`
- `calcChange(4, 8) → -4`

```java
/* in the BookstoreAccountant class. */
public int calcChange(int amtPaid, int price) {
    return amtPaid - price;
}
```

Calling Methods That Have Parameters (6/9)

- Java does 'parameter passing' by:
  - first checking that one-to-one correspondence is honored (this includes type checking),
  - then substituting arguments for parameters,
  - and finally executing the method body using the arguments.

```java
/* somewhere else in our code (e.g., the Bookstore class) */
myAccountant.priceBooks(2, 16);
```

```java
/* in the BookstoreAccountant class. */
public int priceBooks(int numCps, int price) {
    return (numCps * price);
}
```
Calling Methods That Have Parameters (7/9)

/* somewhere else in our code (e.g., the Bookstore class) */

myAccountant.priceBooks(2, 16);

- Java does 'parameter passing' by:
  - first checking that one-to-one correspondence is honored (this includes type checking!),
  - then substituting arguments for parameters,
  - and finally executing the method body using the arguments.

/* in the BookstoreAccountant class... */

public int priceBooks(int numCps, int price) {
    return (numCps * price);
}

Calling Methods That Have Parameters (8/9)

/* somewhere else in our code (e.g., the Bookstore class) */

myAccountant.priceBooks(2, 16);

- Java does 'parameter passing' by:
  - first checking that one-to-one correspondence is honored (this includes type checking!),
  - then substituting arguments for parameters,
  - and finally executing the method body using the arguments.

/* in the BookstoreAccountant class... */

public int priceBooks(int numCps, int price) {
    return (numCps * price);
}

32 is returned

Calling Methods That Have Parameters (9/9)

/* somewhere else in our code (e.g., the Bookstore class) */

System.out.println(myAccountant.priceBooks(2, 16));

- 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 32 printed out!
TopHat Question
Which of the following contains arguments that satisfy the parameters of the method `calcChange()` below in the `BookstoreAccountant` class?

A. `myAccountant.calcChange(20, 14.50)`
B. `myAccountant.calcChange(10)`
C. `myAccountant.calcChange(20, 10)`
D. None of the above

```java
// calculates a customer's change
public int calcChange(int amtPaid, int price) {
    return amtPaid - price;
}
```

But where did `myAccountant` come from?!!

- We know how to send messages to an instance of a class by calling methods
- So far, we have called methods on `samBot`, an instance of `Robot`, and `myAccountant`, an instance of `BookstoreAccountant`...
- Where did we get these objects from? How did we make an instance of `BookstoreAccountant`?
- Next: how to use a class as a blueprint to actually build instances!

Outline
- Mathematical functions in Java
- Defining more complicated methods with inputs and outputs
- The constructor
- Creating instances of a class
- Understanding Java flow of control
Constructors (1/3)

- Bookstore Accountants can priceBooks(), calcChange(), and calcMaxBks()
- Can call any of these methods on any instance of BookstoreAccountant
- But how did these instances get created in the first place?
- Define a special kind of method in the BookstoreAccountant class: a constructor
- Note: every class must have a constructor

```java
public class BookstoreAccountant {
    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
    public int calcChange(int amtPaid, int price) {
        return (amtPaid - price);
    }
    public int calcMaxBks(int price, int myMoney) {
        return (myMoney / price);
    }
}
```

Constructors (2/3)

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

```java
public class BookstoreAccountant {
    public BookstoreAccountant() {
        // this is the constructor!
    }
    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
    public int calcChange(int amtPaid, int price) {
        return (amtPaid - price);
    }
    public int calcMaxBks(int price, int myMoney) {
        return (myMoney / price);
    }
}
```

Constructors (3/3)

- Constructors are special methods: used to create an instance stored in an assigned memory location
- When we create an instance with the constructor (example in a few slides!), it provides a reference to the location in memory, which is "returned"
- We never specify a return value in its declaration
- Constructor for BookstoreAccountant does not take any parameters (notice empty parentheses),
  - constructors can, and often do, take in parameters — stay tuned for next lecture

```java
public class BookstoreAccountant {
    public BookstoreAccountant() {
        // this is the constructor!
    }
    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
    public int calcChange(int amtPaid, int price) {
        return (amtPaid - price);
    }
    public int calcMaxBks(int price, int myMoney) {
        return (myMoney / price);
    }
}
```
TopHat Question
Which of the following is **not** true of constructors?
A. Constructors are methods
B. Constructors always have the same name as their class
C. Constructors should specify a return value
D. Constructors can take in parameters

Outline
- Mathematical functions in Java
- Defining more complicated methods with inputs and outputs
- The constructor
- Creating instances of a class
- Understanding Java flow of control

Creating Instances of Classes (1/2)
- Now that the `BookstoreAccountant` class has a constructor, we can create instances of it!
- Here is how we create a `BookstoreAccountant` in Java:
  ```java
  new BookstoreAccountant();
  ```
- This means "use the `BookstoreAccountant` class as a blueprint to create a new `BookstoreAccountant` instance"
- `BookstoreAccountant()` is a call to `BookstoreAccountant`'s constructor, so any code in constructor will be executed as soon as you create a `BookstoreAccountant`
Creating Instances of Classes (2/2)

- We refer to "creating" an instance as **instantiating** it.
- When we say:
  ```java
  new BookstoreAccountant();
  ```
  ... We're creating an instance of the `BookstoreAccountant` class, a.k.a. **instantiating** a new `BookstoreAccountant`.
- 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 `BookstoreAccountant`!

Aside: Another Example of Nesting (1/2)

- Our `calcChange()` method takes in two `int`s – the amount the customer paid, and price of the purchase.
- Our `priceBooks()` method finds the price of the purchase.
- What if we want to use result of `priceBooks()` as an argument to `calcChange()`?
  - Say we have got 3 copies of an $11 book. We also have $40 in cash to pay with. `priceBooks()` will tell us that purchase costs $33. We want to use this as "price" parameter for `calcChange()`.
  - How do we do this? **Nesting**!
  ```java
  myAccountant.priceBooks(3, 11)
  ```
  returns "33"
- We want to pass this number into `calcChange()`.

Aside: Another Example of Nesting (2/2)

- `myAccountant.priceBooks(3, 11)` returns "33"
- We can nest `myAccountant's priceBooks()` method within `myAccountant's calcChange()` method:
  ```java
  myAccountant.calcChange(40, myAccountant.priceBooks(3, 11));
  ```
  returns 33
- And `calcChange()` returns 7! Always, evaluate inner parentheses first.
TopHat Question
You have an instance of `BookstoreAccountant`, `accountant`, with the methods given from before.

What is the proper way to calculate the change you will have if you pay with a $50 bill for 5 books at a cost of $8 each?

A. `accountant.priceBooks(5, 8);`
B. `accountant.priceBooks(8, 5);`
C. `accountant.calcChange(50, accountant.priceBooks(5, 8));`
D. `accountant.calcChange(accountant.priceBooks(5, 8));`

Important Techniques Covered So Far

- 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
- Up next: Flow of Control

Outline

- Mathematical functions in Java
- Defining more complicated methods with inputs and outputs
- The constructor
- Creating instances of a class
- Understanding Java flow of control
What Is Flow of Control?

- We’ve already seen lots of examples of Java code in lecture.
- But how does all of this code actually get executed, and in what order?
- **Flow of control** or **control flow** is the order in which individual statements in a program (lines of code) are executed.
- Understanding flow of control is essential for hand simulation and debugging.

Overview: How Programs Are Executed

- Code in Java is executed sequentially, line by line.
- Think of an arrow “pointing” to the current line of code.
- Where does execution start?
  - In Java, the first line of code executed is in a method called the main method.

The Main Method

- Every Java program begins at first line of code in main method and ends after last line of code in main is executed – you will see this shortly.
- You will see this method in every project or lab sheet, typically in App.java (the App class).
  - By CS15 convention, we start our programs in App.
- Program starts when you run file that contains main method.
- Every other part of application is invoked from main.
Method Calls and Constructors

- When a method is called, execution steps into the method
  - next line to execute will be first line of method definition
- Entire method is executed sequentially
  - when end is reached (when method returns), execution returns to line following the method call

Example: Baking Cookies

- Some of your TAs are trying to bake cookies for a grading meeting
  - they’ve decided to make mystery flavored cookies, to surprise the HTAs
- Let’s write a program that will have a baker make a batch of cookies!

The makeCookies() Method

- First, let’s define a method to make cookies, in the Baker class
  - public void makeCookies()

- What are the steps of making cookies?
  - mix this
  - combine dry ingredients in another bowl, and mix
  - combine wet and dry ingredient bowls
  - form balls of dough
  - bake for 10 minutes
  - sometime before baking, preheat oven to 400°F

- Order is not fixed, but some steps must be done before others
- Let’s write methods for these steps and call them in order in makeCookies()
Defining the Baker Class

- First, here are more methods of the Baker class — method definitions are elided. Method definitions can occur in any order in the class.

```java
public class Baker {
    public Baker() {
        // constructor code elided for now
    }
    public void makeCookies() {
        // code on next slide
    }
    public void combineWetIngredients() {
        // code to mix eggs, sugar, butter, vanilla
    }
    public void combineDryIngredients() {
        // code to mix flour, salt, baking soda
    }
    public void combineAllIngredients() {
        // code to combine wet and dry ingredients
    }
    public void formDoughBalls(int numBalls) {
        // code to form balls of dough
    }
    public void bake(int cookTime) {
        // code to bake cookies and remove from oven
    }
    public void preheatOven(int temp) {
        // code to preheat oven to a temp
    }
}
```

The makeCookies() Method

```java
public void makeCookies() {
    this.preheatOven(400);
    this.combineWetIngredients();
    this.combineDryIngredients();
    this.combineAllIngredients();
    this.formDoughBalls(24);
    this.bake(10);
}
```

TopHat Question

Using the Baker class from before, is the following method correct for creating cookie dough? Why or why not?

A. Yes, it has all the necessary methods in proper order
B. No, it uses this instead of Baker
C. No, it has the methods in the wrong order
D. No, it is inefficient
Flow of Control Illustrated

- Each of the methods we call in `makeCookies()` has various sub-steps involved
  - `combineWetIngredients()` involves adding sugar, butter, vanilla, eggs, and mixing them together
  - `bake(int cookTime)` involves putting cookies in oven, waiting, taking them out

- In current code, every sub-step of `combineWetIngredients()` is completed before `combineDryIngredients()` is called
  - Execution steps into a called method, executes everything within method
  - Both sets of baking steps must be complete before combining bowls, so these methods are both called before `combineAllIngredients()`
  - Could easily switch order in which those two methods are called

Putting it Together (1/2)

- Now that Bakers have a method to bake cookies, let's put an app together to make them do so
- Java launches our app `App` in its `main` method
- Generally, use `App` class to start our program and have it do nothing else

```java
public class App {
    public static void main(String[] args) {
    }
}
```

Putting it Together (2/2)

- First, we need a Baker
- Calling new Baker() will execute Baker's constructor
- How do we get our Baker to bake cookies?
  - Call the `makeCookies()` method from its constructor
  - This is not the only way — stay tuned for next lecture

```java
public class App {
    public static void main(String[] args) {
        new Baker();
    }
}
```
Following Flow of Control

```java
public class App {
    public static void main(String[] args) {
        new Baker();
    }
}

public class Baker {
    public Baker() {
        makeCookies();
    }
    public void makeCookies() {
        preheatOven(400);
        combineWetIngredients();
        combineDryIngredients();
        combineAllIngredients();
        formDoughBalls(24);
        bake(10);
    }
    public void preheatOven(int temp) {
        // code to preheat oven to a temp
    }
    public void combineWetIngredients() {
        // code to mix eggs, sugar, butter, vanilla
    }
    public void combineDryIngredients() {
        // code to mix flour, salt, baking soda
    }
    public void combineAllIngredients() {
        // code to combine wet and dry ingredients
    }
    public void formDoughBalls(int numBalls) {
        // code to form balls of dough
    }
    public void bake(int cookTime) {
        // code to bake cookies and remove from oven
    }
}
```

Modifying Flow of Control

- In Java, various control flow statements modify sequence of execution
  - these cause some lines of code to be executed multiple times, or skipped over entirely

- We'll learn more about these statements in Making Decisions and Loops lectures later on

Important Concepts Covered

- Numbers represented as integers (e.g., `int` type) or floating-point (e.g., `double` type)
- Defining methods that take in parameters as input
- Defining methods that return something as an output
- Using `System.out.println` to test and debug code
- Defining a constructor for a class
- Creating an instance of a class with the `new` keyword
- Following Java's sequential flow of control
Announcements (1/2)

• Get lab0 checked off by Saturday
  o if you’re having issues with IntelliJ setup or running code or want to get lab checked off come to Conceptual Hours!

• Rattytouille due Saturday, 9/16 @ 11:59pm

• Code-Along to cover Java syntax
  o hands-on opportunity to code along with a TA in a group
  o Tomorrow and Sunday at 7pm in Macmillan 117!
  o check Ed post / email for all the specific dates and times

Announcements (2/2)

• Fill out Mentorship form by tonight at 11:59; mandatory for all freshmen, fill out during lab/section (or using the link on Ed)
• Permanent Lab/Section Swap form up on Ed.
• Temporary Swaps will be dealt with by emailing your lab/section TAs and the TAs of the lab/section you are switching into, at least the Monday of the week.

Socially Responsible Computing: Intro to AI
What is Artificial Intelligence? (approximately!)

**Artificial Intelligence**
The ability of a machine to perform intelligent tasks (predicting outcomes, classifying inputs, learning, planning, perception, robotics…)

**Machine Learning**
The ability of a machine to "learn" by gaining takeaways from data using statistical/mathematical methods (pattern recognition, image discrimination, query analysis)

**Deep Learning**
A subset of ML, based on a simplified model of the human brain (artificial neural networks)

Current Final Project: Othello, uses mini-max algorithm!
History of AI

1997: Deep Blue beats Gary Kasparov
2006: University of Toronto develops Deep Learning
2011: IBM's Watson wins Jeopardy
2016: Go software based on Deep Learning beats world's champions

Increased explosive growth due to GAI

More on large language models next lecture!
Text-to-image generator developed by OpenAI
- Recognizes concepts, attributes and styles

Projected Generative AI Revenue Growth

![Projected Generative AI Revenue Growth](image-source: CB Insights)

AI Startups in Different Market Sectors

![Al Startups in Different Market Sectors](image-source: CB Insights)
Positive Use Cases of AI

Concerning Use Cases

Courses at Brown:
- CSCI 1410: Artificial Intelligence
- CSCI 1420: Machine Learning
- CSCI 1430: Computer Vision
- CSCI 1460: Computational Linguistics
- CSCI 1470: Deep Learning
- CSCI 1951R: Intro to Robotics

How can I get involved?

Quadcopter from CSCI 1951R: Intro to Robotics