• Opportunity to gain practical experience in videogame programming, development, and design
• No previous experience is required or expected!
• Interested students should sign up for the email list (through Facebook/Twitter) to join the first meeting on Zoom, January 30th at 1pm EST
**BROWN STUDENT FORMULA RACING**

**DESIGN, BUILD, COMPETE.**

**DESIGN:** Fall Semester
- Learn Solidworks, MATLAB, Ricardo, ANSYS, and more!
- Work on a team to ideate, prototype, and build parts

**BUILD:** Spring Semester
- Assemble the car, from chassis to electronics, to body
- Learn manufacturing tools and DFM principles

**COMPETE:** End of Spring
- Test, debug, train drivers, push the car to its limits
- In Detroit, race against teams from around the world

**Signup:** [https://tinyurl.com/brownfsae](https://tinyurl.com/brownfsae)
**Email:** fsae@brown.edu

New member intro meeting: Saturday 1/30 5-6 pm EST
Link: [https://brown.zoom.us/j/97444235437](https://brown.zoom.us/j/97444235437)
Lecture 3
Introduction to Parameters / Math
Review of Inter-Object Communication

• 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 `<receiver>..<method>()`
  o e.g., `samBot.moveForward(3)`;
• Typically, sender and receiver are instances of different classes
• Sometimes sender 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
  o Choreographer tells dancer: `dancer3.pirouette(2)`;
  o Dancer tells herself: `this.pirouette(2)`;
  o Note: we’ve not yet learned how to create new instances of any class

Note: Object is used loosely for both class and instance. We try to minimize our use of this overloaded term.
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**
5. **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)
Basic Math in Java

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

\[ V = l \times w \times h \]
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
## 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,...,808 (-2^{63})</td>
<td>9,223,372,...,807 (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 rational and irrational numbers, i.e., numbers with decimal points

● How to represent \( \pi = 3.14159\ldots \)?

● Floating point numbers
  o called “floating point” because decimal point can “float” – no fixed number of digits before and after it – historical nomenclature
  o 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: \texttt{float} and \texttt{double}
## Floating Point Numbers

<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.
## Operators and Math Expressions (1/2)

- **Example expressions:**
  - $4 + 5$
  - $3.33 * 3$
  - $11 \% 4$
  - $3.0 / 2.0$
  - $3 / 2$

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

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

- When dividing two integer types, result is “rounded down” to an int after remainder is dropped
- 3 / 2 evaluates to 1
- If either number involved is floating point, result is floating point: allows greater “precision,” i.e., fractional portion.
  - 10 / 3 → 3
  - 10 / 3.0 → 3.3333... (more precise)
  - called mixed-mode arithmetic

\[
\begin{align*}
3 / 2 & \rightarrow 1 \\
3.0 / 2 & \rightarrow 1.50 \\
3 / 2.0 & \rightarrow 1.50 \\
3.0 / 2.0 & \rightarrow 1.50
\end{align*}
\]
Evaluating Math Expressions

- Java follows the same evaluation rules that you learned in math class years ago – PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction)

- Evaluation takes place left to right, except:
  - expressions in parentheses evaluated first, starting at the innermost level
  - operators evaluated in order of precedence/priority (* has priority over +)

\[
2 + 4 \times 3 - 7 \rightarrow 7
\]

\[
(2 + 3) + (11 / 12) \rightarrow 5
\]

\[
3 + (2 - (6 / 3)) \rightarrow 3
\]
Lecture Question

What does x evaluate to?

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
Return Type (1/2)

- 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”

```java
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();
    }
}
```
Return Type (2/2)

- 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`.
  - Any code following the return statement will not be executed.

A silly example:

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

This is a return statement.

Return statements always take the form:

```java
return <something of specified return type>;
```
Accountant (1/6)

- 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
- We will generalize this example soon…

```java
public class BookstoreAccountant {

    /* Some code elided */

    public int priceTenDollarBook() {
        return 10;
    }

    // "10" is an integer – it matches the return type, int!
}
```
Accountant (2/6)

- 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.
Accountant (3/6)

/* Somewhere in the Bookstore class lives an instance of the BookstoreAccountant class named myAccountant */

myAccountant.priceTenDollarBook();

● We start by just calling the method
● This is fine, but we are not doing anything with the result!
● Let’s use the returned value by printing it to the terminal

```java
public class BookstoreAccountant {
    /* Some code elided */
    
    public int priceTenDollarBook() {
        return 10;
    }
}
```
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!)
Accountant (4/6)

• 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;
    }

    public void manageBooks() {
        System.out.println(
            this.priceTenDollarBook());
    }
}
```
Accountant (5/6)

- 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.

```java
public class BookstoreAccountant {
    /* Some code elided */

    public int priceTenDollarBook() {
        return 10;
    }

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

    public int priceTenDollarBook() {
        return 10;
    }

    public void manageBooks() {
        System.out.println(
            this.priceTenDollarBook());
    }
}

accountant (6/6)

• 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
Accountant: A More Generic Price Calculator (1/4)

- 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?
  - useful when the bookstore needs to order new books

```java
public class BookstoreAccountant {
    public int priceTenDollarBook() {
        return 10;
    }

    public int priceBooks(int numCps, int price) {
        // let's fill this in!
    }

    // cost of the purchase
    // number of copies you're buying
    // price per copy
}
```
Accountant: A More Generic Price Calculator (2/4)

- 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
public 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

```java
public class BookstoreAccountant {
    /* Some code elided */

    public int priceBooks(int numCps, int price) {
        return (numCps * price);
    }
}
```
Parameters (1/3)

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

- Parameters are specified as comma-separated list
  
  - for each parameter, specify type (for example, int or double), and then name (“x”, “y”, “banana”... whatever you want!)

- In basic algebra, we do not specify type because context makes clear what kind of number we want. In programming, we use many different types 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 one-to-one correspondence
Parameters (2/3)

- Name of each parameter is **almost** completely up to you
  - 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

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);
}
```
Parameters (3/3)

- Remember `Robot` class from last lecture?
- Its `moveForward` method took in 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 “numCps” and “price”

- Try to avoid single-letter names for anything that is not strictly mathematical; be more descriptive
Accountant (1/2)

- Let’s give `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 a price per 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
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);
    }
}
```
Lecture 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 int getSalePrice() { // code elided }`
Calling (i.e., using) Methods with Parameters (1/3)

● Now that we have defined `priceBooks`, `calcChange`, and `calcMaxBks` methods, we can call them on any `BookstoreAccountant`.

● 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
class Robot {
    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.

```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);
    }
}
```

- When we call `moveForward` we must supply one `int` as argument.

- Do not specify type of argument when calling a method.
  - `samBot.moveForward(int 4);` is NOT correct.
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 into it, e.g. “x” and “y” in the function \( f(x, y) = x + y \). It is a “dummy name” determined by definer.

● In calling a method, an argument is the actual value passed in, e.g. 2 and 3 in \( \text{add}(2, 3) \).

// within the RobotMover class

public void moveRobot(Robot samBot) {
    samBot.moveForward(4);
    samBot.turnRight();
    samBot.moveForward(1);
    samBot.turnRight();
    samBot.moveForward(3);
}

argument

parameter
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.

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

```java
// in some other class...
samBot.moveForward(3);

// in the Robot class...
public void moveForward(int numberOfSteps) {
    // code that moves the robot
    // forward goes here!
}
```
Calling Methods That Have Parameters (2/9)

- When calling a method that takes in parameters, must provide a valid argument for each parameter
  - loose analogy: To get to Love Island, Huey must fill his boat’s fuel tank with the right type of fuel for it to work
- Means that number and type of arguments must match number and type of parameters: one-to-one correspondence
- Order matters! The first argument you provide will correspond to the first parameter, second to second, etc.
Calling Methods That Have Parameters (3/9)

- 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 our desired value for first parameter, then desired value for second

```java
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 say we have an 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
- These 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(18, 12);
myAccountant.calcMaxBks(6, 33);
```
Calling Methods That Have Parameters (5/9)

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

```java
public int calcChange(int amtPaid, int price) {
    return amtPaid - price;
}
```
Calling Methods That Have Parameters (6/9)

/* somewhere else in our code... */

myAccountant.priceBooks(2, 16);

• Java does “parameter passing” by:
  o first checking that one-to-one correspondence is honored,
  o then substituting arguments for parameters,
  o 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 (7/9)

/* somewhere else in our code... */

myAccountant.priceBooks(2, 16);

/* in the BookstoreAccountant class... */

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

- Java does “parameter passing” by:
  - first checking that 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 (8/9)

/* somewhere else in our code... */

myAccountant.priceBooks(2, 16);

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

/* in the BookstoreAccountant class... */

public int priceBooks(2, 16) {
    return (2 * 16);
}

32 is returned
Calling Methods That Have Parameters (9/9)

/* somewhere else in our code… */

```
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!

/* in the BookstoreAccountant class… */

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

Which of the following contains arguments that satisfy the parameters of the method `calcChange` in the `BookstoreAccountant` class?

A. `myAccountant.calcChange(20, 14.50)`
B. `myAccountant.calcChange(10.00, 5.00)`
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!
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 in 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!
        // constructor code elided
    }

    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);
    }

}
Lecture 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
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: 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!**
Aside: Nesting (2/2)

- `myAccountant.priceBooks(3, 11)` returns “33”
  - we want to pass this number into `calcChange`
- We can **nest** `myAccountant`'s `priceBooks` method within `myAccountant`'s `calcChange` method:

```
myAccountant.calcChange(40, myAccountant.priceBooks(3, 11));
```

Aside: Nesting (2/2)

- And `calcChange` returns 7! Always, evaluate inner parentheses first

```
myAccountant.calcChange(40, 33);
```
Lecture 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(accountant.priceBooks(5, 8))`;  
D. `accountant.calcChange(50, 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
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, first line of code executed is in a special 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 stencil, typically in App.java (the `App` class)
  o 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 the line following the method call

```java
public static void main(String[] args) {
    System.out.println("first line");
    System.out.println("last line");
}
```
Example: Baking Cookies

● Some of your TAs are trying to bake cookies for a grading meeting
  ◦ they’ve decided to make Love 15land cookies, the HTAs’ favorite kind

● 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?
  - combine wet ingredients (and sugars) in one bowl
    - 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 400º

- 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
    }
}
// end of Baker class
```
The `makeCookies()` Method

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

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

```java
public class Baker {
    //constructor elided
    public void createDough() {
        this.combineWetIngredients();
        this.combineAllIngredients();
        this.combineDryIngredients();
    }
    //other methods elided
}
```

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

- Java launches our app App in its main method

- Generally, use App class to start our program and have it do nothing else
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();
    }
}

// in Baker class
public Baker() {
    this.makeCookies();
}
```
public class App {
    public static void main(String[] args) {
        new Baker();
    }
}

public class Baker {
    public Baker() {
        this.makeCookies();
    }
    public void makeCookies() {
        this.preheatOven(400);
        this.combineWetIngredients();
        this.combineDryIngredients();
        this.combineAllIngredients();
        this.formDoughBalls(24);
        this.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
    }
} // end of Baker class
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
Announcements

• Hello World is due on **Saturday, 1/30 at 11:59PM**
  o get Lab0 checked off at conceptual hours before beginning Hello World
  o If your IntelliJ setup/running your code doesn’t work, reach out to HTAs for help

• AndyBot will be released on **Sunday, 1/31**
  o AndyBot is due on **Thursday, 2/4 at 11:59PM**
  o *Hello World and AndyBot must be turned in on codePost*

• Permanent Lab/Section Signups out today!
  o these will be your slots for the rest of the semester

• Please sign the collaboration policy!
  o linked in the Hello World handout
IT in the News

ft. Socially Responsible Computing!
Types of COVID-related misinformation, disinformation, and false narratives:

• The virus itself
  • ex: it’s a hoax, children are immune
• Cures / preventative measures
  • ex: drinking bleach
• Anti-vaccine, anti-mask
  • ex. Bill Gates microchip conspiracy
• Feel-good stories
  • ex: dolphins returned to Venice canals! :)

Social Media in 2020: A Year of Misinformation and Disinformation

It was a tough year for social media. But, Kate Starbird says, it can still be a force for positive communication.

December 11, 2020 Wall Street Journal
Negative Effects of Social Media & COVID-19 (2/2)

• Nearly half of Twitter accounts talking about COVID are likely bots
  • Creates confusion and distrust of authority
  • Makes the public less likely to comply with measures like lockdowns and social distancing

• The pandemic significantly increased social media use to cope with isolation
  • Especially Facebook, Zoom, TikTok, Instagram
  • Cost = mass surveillance, loss of privacy
  • Higher level of social media use associated with worse mental health

---

Media Consumption Influenced by COVID-19

How has your usage of the following changed, if at all, since the COVID-19 outbreak started?

<table>
<thead>
<tr>
<th>Activity</th>
<th>% of “Increased Significantly” or “Increased Somewhat”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using social media</td>
<td>66%</td>
</tr>
<tr>
<td>Streaming content from a subscription service</td>
<td>55%</td>
</tr>
<tr>
<td>Streaming content from a free service</td>
<td>68%</td>
</tr>
<tr>
<td>Watching live programming</td>
<td>47%</td>
</tr>
<tr>
<td>Playing games on a mobile device or tablet</td>
<td>45%</td>
</tr>
<tr>
<td>Watching recorded content</td>
<td>37%</td>
</tr>
<tr>
<td>Reading magazines/newspaper</td>
<td>31%</td>
</tr>
<tr>
<td>Playing games on a computer</td>
<td>25%</td>
</tr>
<tr>
<td>Buying or renting a movie/TV show online</td>
<td>22%</td>
</tr>
<tr>
<td>Watching Blu-ray or DVD</td>
<td>22%</td>
</tr>
<tr>
<td>Playing video games on a console</td>
<td>18%</td>
</tr>
</tbody>
</table>

* Numerator Survey 4/14/2020 – Shoppers with confirmed purchases week prior | N=1059
Positive Effects of Social Media and COVID-19

• Social media use can increase connectedness and happiness
  • US, 2016: link between use of social network sites and increased feelings of connectedness
  • Germany, 2019: daily online social network users reported lower social isolation scores
  • Belgium, 2020: social media use led to increased happiness for participants with anxiety
• Social media + tech enables distance learning, remote healthcare, etc.
  • widens digital divide between haves/have-nots