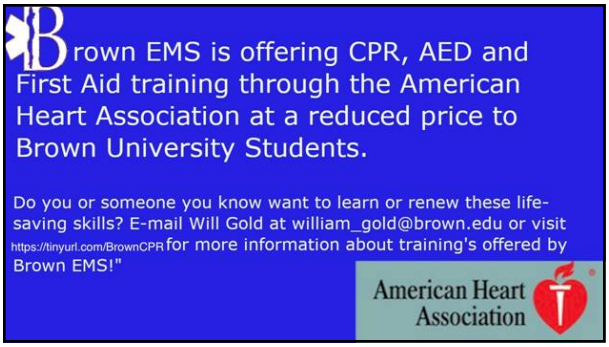
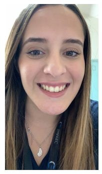

Brown EMS is offering CPR, AED and First Aid training through the American Heart Association at a reduced price to Brown University Students.
Do you or someone you know want to learn or renew these life-saving skills? E-mail Will Gold at william_gold@brown.edu or visit https://tinyurl.com/BrownCPR for more information about training's offered by Brown EMS!"
American Heart Association

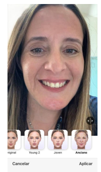

20 Design 19 Fair
September 14th from 12-2pm
at the Brown Design Workshop (BDW)
hosted by Design@Brown and the BDW
Come talk to a few of the groups and organizations from across campus who engage in design, creativity, and innovation (and eat some free pizza too!)
Brown Design Workshop (BDW)
Design@Brown
Discover
Brown STEM + Art (STEAM)
Brown Space Engineering (BSE)
Better World by Design
Design for America (DFA)
College Hill Independent
Brown RISD Game Developers
Brown Bytes
Tink Knit @ Brown
Biomedical Engineering Masters in Design


MOSAIC+
Mosaic+ Mixer
9/13 @4:30-6pm
Location: Front Lawn of Barus&Holley
Email us to join the listserv
mosaic.plus.brown@gmail.com

IT in the News

- New face-morphing app called “FaceApp” uses machine learning to artificially age faces
- Created by a Russia-based company that could use the data they collect for bypassing facial recognition technology
- Users must agree to Terms & Conditions which gives the company **full** and **irrevocable** access to their photo library and user data
- FaceApp's terms of service state that it won't rent or sell client's information to third-parties outside FaceApp (**or the group of companies of which FaceApp is a part**) without client's consent.
  - if the platform is sold to another company, user information and content is “up for grabs”.





Article source:
https://www.npr.org/2019/07/17/742910309/democrats-issue-warnings-against-viral-russia-based-face-morphing-app?utm_medium=social&utm_source=facebook.com&utm_campaign=npr&utm_term=nprnews&fbclid=IwAR3AqkGxEuFbor6FF1eTIX7Pxul3sAuTqvMfFHJwDkSmPiqERXRfR8UYzfM&fbclid=IwAR1iNSY2O41hie9Nas2xicY90y-UAD6yTjxQlTkHgDvSL6C7hje9Non4qbo
Photo source:
https://www.cinetelerevue.be/actus/quest-ce-que-faceapp-la-dangereuse-appli-devenue-virale

IT in the News

- Many celebrities, government personnel, military service members, and more have used this app, thereby putting their data at risk to foreign governments
- Security advocates are expressing concern about how this data could be used in a military or police context considering that Russia actively engages in cyber hostilities.

Article source:
https://www.npr.org/2019/07/17/742910309/democrats-issue-warnings-against-viral-russia-based-face-morphing-app?utm_medium=social&utm_source=facebook.com&utm_campaign=npr&utm_term=nprnews&fbclid=IwAR3AqkGxEuFbor6FF1eTIX7Pxul3sAuTqvMfFHJwDkSmPiqERXRfR8UYzfM&fbclid=IwAR1iNSY2O41hie9Nas2xicY90y-UAD6yTjxQlTkHgDvSL6C7hje9Non4qbo
Photo source:
https://www.thewrap.com/faceapp-perpetual-license-to-photos-russia/

Lecture 3

Introduction to Parameters / Math

Review of Inter-Object Communication

- Instances send each other messages
- Instances respond to a message via a method
- Format of messages is `<receiver>.<method>();`
  - 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 class: `this.<method>();`
- `this` means "me, myself" AND the method is defined in this class
- Example:
  - 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:

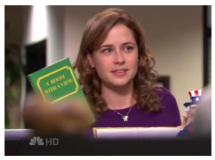
1. Mathematical functions in Java
2. Defining more complicated methods with inputs and outputs
3. The constructor
4. Creating instances of a class
5. Understanding Java flow of control

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, you 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 an output (numeric answer)



Basic Math in Java

$V = l \times w \times h$

- 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

Integers

Base Type	Size	Minimum Value	Maximum Value
`byte`	8 bits	-128 (-2^7)	127 ($2^7 - 1$)
`short`	16 bits	-32,768 (-2^{15})	32,767 ($2^{15} - 1$)
`int`	32 bits	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
`long`	64 bits	-9,223,372,...,808 (-2^{63})	9,223,372,...,807 ($2^{63} - 1$)

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...?
- **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., 4.3×10^{-5}
- Two numerical base types in Java represent floating point numbers: `float` and `double`

Floating Point Numbers

Base Type	Size
`float`	32 bits
`double`	64 bits

Feel free to use both in CS15. Use of `double` is more common in modern Java code

Operators and Math Expressions (1/2)

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	remainder

- Example expressions:

```
4 + 5
3.33 * 3
11 % 4
3.0 / 2.0
3 / 2
```


Operators and Math Expressions (2/2)

- What does each of these expressions evaluate to?

- Example expressions:

```
4 + 5 → 9
3.33 * 3 → 9.99
11 % 4 → 3
3.0 / 2.0 → 1.50
3 / 2 → 1
```

why???

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

```
3 / 2 → 1
3.0 / 2 → 1.50
3 / 2.0 → 1.50
3.0 / 2.0 → 1.50
```

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 * 3 - 7 → 7

(2 + 3) + (11 / 12) → 5

3 + (2 - (6 / 3)) → 3
```


19/81

Top Hat Question

What does x evaluate to?

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

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

20/81

BookstoreAccountant

- `BookstoreAccountant`s 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

21/81

Return Type (1/2)

- The **return type** of a method is the kind of data it gives back to whoever 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"

```
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:

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

This is a **return statement**.

Return statements always take the form:

```
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…

```
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 whoever 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 console**

```
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 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 console, we use `System.out.println` (<expression to print>)
- `println` method prints out value of expression you provide within the parentheses

```
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

```
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 on `System.out`, and it needs to use result of `priceTenDollarBook`
  - `priceTenDollarBook` is called on `this`, returning 10
  - `println` gets 10 as an argument, 10 is printed to console

```
public class BookstoreAccountant {

  /* Some code elided */

    public int priceTenDollarBook() {
        return 10;
    }

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

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

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

```
public class BookstoreAccountant {

    public int priceTenDollarBook() {
        return 10;
    }

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


Accountant: A More Generic Price Calculator (3/4)

Mathematical function:

f(x, y) = x * y

name inputs output

Equivalent Java method:

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

name inputs output

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

- 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

```
public class BookstoreAccountant {

  /* Some code elided */

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

}
```

parameters

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:

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

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

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**

```
/* Within Robot class definition */
                              type          name
                               ↘             ↙
 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

```
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 `int`s, Java rounds the result down to an `int`!
  - Java **always rounds down**
- $25 / $10 per book = 2 books

```
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);
    }

}
```


Top Hat Question: Declaring Methods

- We want a new method `calcAvgBks` that returns an integer and takes in two parameters, one integer that represents the price of all books and one integer that represents the number of books. Which method declaration is correct?

A.
```
public void calcAvgBooks() {
        return (price / numBooks);
    }
```

B.
```
public int calcAvgBooks(int price, int numBooks) {
        return (price / numBooks);
    }
```

C.
```
public int calcAvgBooks(price, numBooks) {
        return (price / numBooks);
    }
```

D.
```
public int calcAvgBooks() {
        return (price / numBooks);
    }
```


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`?

```
//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

```
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

Arguments vs. Parameters

parameter

```
// 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. 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 `add(2, 3)`

```
// within the RobotMover class

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

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

```
// 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: When making copies, Pam has to make sure she puts the right sized paper into the printer!
- 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.

Better leave it to Pam….

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

```
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 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, `int`s) in parentheses
- These **arguments** are values we want the method to use for the first and second parameters when it runs

```
/* somewhere else in our code… */

myAccountant.priceBooks(2, 16);
myAccountant.calcChange(18, 12);
myAccountant.calcMaxBks(6, 33);
```

arguments

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

```
/* in the BookstoreAccountant class… */

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:
  - first checking that one-to-one correspondence is honored,
  - 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 (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);
```

```
/* in the BookstoreAccountant class… */

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

32 is returned

- 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 (9/9)

```
/* somewhere else in our code… */

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

```
/* in the BookstoreAccountant class… */

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

- 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!

Top Hat Question

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

A. BookstoreAccountant.calcChange(20, 14.50)
B. BookstoreAccountant.calcChange(10.00, 5.00)
C. BookstoreAccountant.calcChange(20, 10)
D. None of the above

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**

```
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 object 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"

```
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 only once, to create an instance in memory that can be assigned.
- When we create an instance with the constructor using =, it provides a reference to the location in memory, which is "returned"
- And we never have to 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

```
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);
    }

}
```


Top Hat 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

Instantiating Objects (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:

```
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`

Instantiating Objects (2/2)

- We refer to "creating" an object as **instantiating** it
- When we say:

```
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));
```

```
                                returns 33
myAccountant.calcChange(40,             33);
```

- And `calcChange` will return 7!

Top Hat 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)
  - 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

Ignore this parameter for now, we'll discuss it later this semester

```
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 The Office 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

```
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

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


Top Hat Question

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

```
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 substeps 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 substep 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 `Baker`s have a method to bake cookies, let's put an app together to make them do so
- Our app starts in the `main` method, in `App`
  - generally, use `App` class to start our program and nothing else

```
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 constructor!
  - this is not the only way -- stay tuned for next lecture

```
public class App {

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

instantiates a Baker

```
// in Baker class
public Baker() {
    this.makeCookies();
}
```

Following Flow of Control

```
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

- HW1 is due on **Saturday, 9/14 at 11:59PM**
- AndyBot will be released on **Sunday, 9/15**
  - AndyBot is due on **Thursday, 9/19 at 11:59PM**
  - *HW1 and AndyBot must be turned in through the cs0150_handin script* (department machine)
- Questions on homework or course material?
  - sign up for Piazza at https://piazza.com/class/ivxw4tiov2n11c
  - make sure your questions are private!
- Please sign the collaboration policy!
- If you need to email an individual TA: <login>@cs.brown.edu!
  - Logins are on the staff page of the website