Lecture 7
Static Methods, Constants, and Making Decisions

Outline
• Review: numbers in Java and arithmetic operations
• Static methods and static variables
• Constants – values that never change
• Decision making: boolean algebra, if-else statements and the switch statement
• Method overloading – defining multiple methods of the same name

Review: Numbers in Java
• Integers represented with base type int
• Floating point numbers (decimals) represented with base type float (32 bits) or double (64 bits)
## Review: Basic Arithmetic Operators

<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>

## Basic Arithmetic Operators: Shorthand

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>add and reassign</td>
<td><code>a += 5;</code></td>
<td><code>a = a + 5;</code></td>
</tr>
<tr>
<td>-=</td>
<td>subtract and reassign</td>
<td><code>a -= 5;</code></td>
<td><code>a = a - 5;</code></td>
</tr>
<tr>
<td>*=</td>
<td>multiply and reassign</td>
<td><code>a *= 5;</code></td>
<td><code>a = a * 5;</code></td>
</tr>
<tr>
<td>/=</td>
<td>divide and reassign</td>
<td><code>a /= 5;</code></td>
<td><code>a = a / 5;</code></td>
</tr>
<tr>
<td>%=</td>
<td>take remainder and reassign</td>
<td><code>a %= 5;</code></td>
<td><code>a = a % 5;</code></td>
</tr>
</tbody>
</table>

## Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>negate</td>
<td><code>b = -b;</code></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td><code>b++;</code></td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td><code>b--;</code></td>
</tr>
</tbody>
</table>
Increment and Decrement Operators

- `++` and `--` can be applied before (prefix) or after (postfix) the operand.
  - `i++` and `++i` will both increment variable `i` and assigns.
  - `i++` assigns, then increments.
  - `++i` increments, then assigns.

**Postfix example:**

```java
int i = 10;
int j = i++; // j becomes 10, i becomes 11
```

**Prefix example:**

```java
int i = 10;
int j = ++i; // i becomes 11, j becomes 11
```

---

**Java.lang.Math**

- Extremely useful "utility" class, part of core Java libraries.
- Provides methods for basic numeric operations:
  - `abs(double a)`: absolute value
  - `pow(double a, double b)`: exponential
  - `log(double a), log10(double a)`: natural and base 10 logarithm
  - `sqrt(double a)`: square root
  - `cos(double a), sin(double a)...`: trigonometric functions
  - `random()`: returns random number from 0.0(inclusive) to 1.0(exclusive)
- For more check out: [http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html)

---

**Static Methods**

- All of `java.lang.Math`'s methods are declared `static`.
- Example: the method that returns the absolute value of an integer is declared below:
  ```java
  public static int abs(int a) {...}
  ```
- A `static method` belongs to a class, rather than an instance of the class:
  - It cannot access instance variables, whose values may differ from instance to instance.
  - But can have local variables.
Calling a **static** Method

- **static** methods are invoked on the class, not on an instance:
  ```java
  int absoluteValue = Math.abs(-7);
  ```
- That means we can use all of `Math`’s **static** methods without ever instantiating it

*Note*: You won’t need to write any **static** methods of your own in CS15, but you’ll be using `Math`’s **static** methods in future assignments.

---

**TopHat Question**

Object `myCountdownClock` is an instance of the `Countdown` class. Which is the correct way to call this static method:
```java
public static int minutesToFive(){...}
```

A. `int minutesLeft = Instance.minutesToFive();`
B. `int minutesLeft = Countdown.minutesToFive(static);`
C. `int minutesLeft = CountdownInstance.minutesToFive(static);`
D. `int minutesLeft = Countdown.minutesToFive();`
E. `int minutesLeft = myCountdownClock.minutesToFive();`

---

**static** Variables

- Progression in scope:
  - **local** variables are known in a single method
  - **instance** variables are known to all methods of a class
  - **static** instance variables are known to all instances of a class
- Each instance of a class has the same instance variables but typically with different values for those properties
- If instead you want all instances of a class to share the same value for a variable, declare it **static** — this is not very common
- Each time any instance changes the value of a **static** variable, all instances have access to that new value
**static Variables: Simple Example**

- `paperSales` starts out with a value of 0
- Each time a new instance of `PaperSale` is created, `paperSales` is incremented by 1
- Get current value at any point by calling: `PaperSale.getNumPaperSales();`
- **static** methods can use **static** variables — but not instance variables

```java
public class PaperSale {
    private static int _paperSales = 0;

    public PaperSale() {
        _paperSales++;
    }

    public static int getNumPaperSales() {
        return _paperSales;
    }
}
```

**Constants**

- **Constants** are used to represent values which never change (e.g. Pi, speed of light, etc.) — very common!
- Keywords used when defining a constant:
  - `public`: value should be available for use by anyone (unlike `private` instance variables and local variables)
  - `static`: all instances of the class share one value
  - `final`: value cannot be reassigned
  - Naming convention for **constants** is **all caps** with underscores between words: `LIGHT_SPEED`

**Constants: Example (1/2)**

- Useful to bundle a bunch of constants for your application in a "utility" class (like `Math`), with useful methods using those constants; both constants and methods will be then declared **static**

```java
public abstract class Physics {
    // speed of light (units: hundred million m/s)
    public static final double LIGHT_SPEED = 2.998;

    // constructor elided
    public static double getDistanceTraveled(double numSeconds) {
        return (LIGHT_SPEED * numSeconds);
    }
}
```
Constants: Example (2/2)

- Always use constants when possible
  - Literal numbers, except for 0 and 1, should rarely appear in your code
  - Makes code readable, easy to maintain
- Also called *symbolic* constants—should have descriptive names
- If many classes use same constants, make separate utility class, like `Physics`
- A constants utility class should never be instantiated, so it should be declared `abstract`

```java
public abstract class Physics {
    // speed of light (Units: hundred million m/s)
    public static final double LIGHT_SPEED = 2.998;
    // we can add more constants if we want
}
```

We can access this constant from another class in our program like this:
`Physics.LIGHT_SPEED` (another use of dot notation)

Example:
`spaceShip.setSpeed(Physics.LIGHT_SPEED)`

TopHat Question

Which of the following constants is defined correctly?

A. `public static final double PAPER_WEIGHT = 20;`
B. `public static double final PAPER_WEIGHT = 20;`
C. `private static final double PAPER_WEIGHT = 20;`
D. `public static final double PAPER_WEIGHT;`

Workout (1/6)

- Dwight Schrute decides to try Andy's *super calf workout*—let's model it!
- Depending on his `WEIGHT` and time of his workout, he will gain a certain amount of calf muscle
- Our Head TAs calculated that his effort is the `WEIGHT` times his workout time
- Muscle gained equals one tenth of the square root of his effort
Workout (2/6)

- **WorkoutConstants** class keeps track of important constant values in our calculation

```java
public abstract class WorkoutConstants {
    // Weight
    public static final double START_WEIGHT = 150;

    // Don't want him to look like this:
    public static final double MAX_WEIGHT = 200;
}
```

Workout (3/6)

- Dwight keeps track of instance variable `_weight`
- `_weight` initialized in constructor to starting weight defined in `WorkoutConstants`

```java
import java.lang.Math;
public class Dwight {
    private double _weight;
    public Dwight() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
}
```

Workout (4/6)

- Dwight’s `gainMuscle` method changes his weight according to the amount of time he works out

```java
import java.lang.Math;
public class Dwight {
    private double _weight;
    public Dwight() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```
Workout (5/6)

- First, effort is computed
- Second, muscleGained is calculated according to the formula
- Math.sqrt is a static method from java.lang.Math that computes the square root of a value
- Increment the weight with the muscle gained

```java
import java.lang.Math;
public class Dwight {
    private double _weight;
    public Dwight() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```

Workout (6/6)

- Now fill in `calfWorkout` method
- Dwight will only work out if weight is not already above maximum WEIGHT
- How can we check if condition is met?
- Introducing... boolean's and if's!

```java
import java.lang.Math;
public class Dwight {
    private double _weight;
    public Dwight() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
    public void calfWorkout() {
        // code to workout!
    }
}
```

**booleans**

- British logician George Boole (1815-1864) wanted to improve on Aristotelian (formal) logic, e.g., modus ponens, rule of inference:
  - "All men are mortal, Socrates is a man, therefore..."
- boolean (named after is Boole) is simplest Java base type
- A boolean variable can have value true or false
- Example initialization:
  ```java
  boolean foo = true;
  boolean bar = false;
  ```

The terms foo, bar, etc. are often used as placeholder names in computer programming or computer-related documentation: derives from FUBAR, WWII slang
Relational Operators

• Can compare numerical expressions with relational operators.
• Full expression evaluates to a boolean: either true or false.
• Examples:
  - boolean b1 = (3 > 2);
  - boolean b2 = (5 == 5);
  - boolean b3 = (x <= 6);
• b1 and b2 are true, b3 is false.

Operator | Meaning
---|---
== | is equal to
!= | is not equal to
> | is greater than
< | is less than
>= | is greater than or equal to
<= | is less than or equal to

Comparing References

• Can use == and != to see if two references point to the same instance, or not.
• What three values are printed to the console in this example?
  1. false: d1 and d2 are not equal
  2. true: d1 and d2 refer to the same instance
  3. true: d1 != d2 is false, so foo is true (no false)

```java
public class DogPark {
    // constructor elided
    public void compareReferences() {
        Dog d1 = new Dog();
        Dog d2 = new Dog();
        boolean foo = (d1 == d2);
        System.out.println(foo);
        d2 = d1;
        foo = (d1 == d2);
        System.out.println(foo);
        boolean foo != (d2 != d1);
        System.out.println(foo);
    }
}
```

TopHat Question

Which of the following will print false?

```java
public class TestClass {
    // constructor elided
    public void compareReferences() {
        Student s1 = new Student();
        Student s2 = new Student();
        boolean student1Exists = (s1 != null);
        System.out.println(student1Exists);
        A. s2 = s1;
        System.out.println(s2 != s1);
        B. s2 = s1;
        System.out.println(s2 == s1);
        C. s2 = null;
        System.out.println(s2 == null);
```
if Statements

- if statements allow us to make decisions based on value of a boolean expression
- Syntax:
  
  ```java
  if (boolean expression) {
    // code to be executed if expression is true
  }
  ```
- If boolean expression is true, code in body of if statement is executed. If false, code in body skipped
- Either way, Java compiler continues on with rest of method

if Statement: Flow Chart

- if (myBoolean) {
  // code to execute if myBoolean is true
}
- int y = 9;
  // more code elided
  if (y > 7) {
    // code to execute if y is greater than 7
  }

if Statements: Examples
Logical Operators: And, Or, Not (1/2)

- Logical operators `&` ("and") and `||` ("or") can be used to combine two boolean expressions
  - `<expression a> && <expression b>` evaluates to true only if both expressions are true
  - `<expression a> || <expression b>` evaluates to true if at least one expression is true
- Logical operator `!` ("not") negates a boolean expression
- Logical operator `^` ("exclusive or") returns true if either `a` or `b` is true but not both

### Truth Table

| A  | B  | A && B | A || B | A^B | !A |
|----|----|--------|--------|-----|----|
| false | false | false | false | false | true |
| false | true  | false | true  | true | true |
| true | false | false | true  | true | false |
| true | true  | true  | true  | false | false |

TopHat Question

Which `if` clause statement will run if Jim does not prank Dwight and Stanley is doing a crossword puzzle? (The variables below are of type `boolean`)

A. `if(!jimPranksDwight && !isStanleyPuzzling){...}`
B. `if(!jimPranksDwight && isStanleyPuzzling){...}`
C. `if(jimPranksDwight && !isStanleyPuzzling){...}`
D. `if(jimPranksDwight && isStanleyPuzzling){...}`
if Statements: More Examples

- Should always take one of two forms:
  - if (<boolean expression>)
  - if (!<boolean expression>)

- Never do this (inefficient):
  - if (<boolean expression> == true)
  - if (<boolean expression> == false)

- Be careful! It's easy to mistakenly use = (assignment operator) instead of == (comparator)

```java
if (!myBoolean) {
    // code to execute if myBoolean is false
}
```

```java
int x = 4;
if (x == 4) {
    // code to execute if x == 4
}
```

```java
if (myBoolean == false) {
    // code to execute if myBoolean is false
    // code is inefficient
}
```

if-else (1/2)

- If want to do two different things depending on whether the boolean expression is true or false, we can use an else clause

- Syntax:
  ```java
  if (<boolean expression>) {
      // code executed if expression is true
  } else {
      // code executed if expression is false
  }
  ```

if-else (2/2)

- Can use if-else to fill in the calfWorkout method

- If Dwight's WEIGHT is not greater than the maximum WEIGHT when the method is called, he gains muscle
- Otherwise, he stops and works on building a new invention!
- Does this code limit the final calf weight to MAX_WEIGHT?

```java
import java.lang.Math;
public class Dwight {
    private double _weight;
    // constructor elided
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
    public void calfWorkout() {
        if (_weight < WorkoutConstants.MAX_WEIGHT) {
            this.gainMuscle(60.0); // workout for 60 minutes!
        } else {
            // this method defined elsewhere in the code
            this.stopAndBuildInator();
        }
    }
}
```
An Object-Oriented Mindset to Leadership

Dhruv ’14 is a Technical Program Manager at Google. He leads global cross-functional programs at YouTube and previously worked at Waymo (Google’s self-driving cars) and Android.

Dhruv was a CS15 HTA. He developed DoodleJump, an upcoming project, and SignMeUp, the app CS dept uses for TA hour signups.

Dhruv shadowed 9/24’s lecture. Despite no longer being an engineer, he uses the principles of OOP to successfully lead projects at Google.

### Encapsulation

I think of Google’s cross-functional teams as organizational objects. I work with them to define what properties they own (PRDs, UX mocks, engine designs) and what functions they have (build/test/ship a feature, collect user feedback, etc.).

### Polymorphism

When a team is given a business problem to solve, I get different responses based on the kind of problem being tackled – sometimes, a finished product, and other times, data confirming why something shouldn’t be done.

### Interfaces

I clarify expectations between teams using interfaces. I define the methods of engagement, but it’s up to the teams to implement them. The organization aligns around these interfaces/contracts.

### Classes vs. Instances

Instead of tackling one instance of a problem at a time, I factor out commonalities to solve an entire class of problems at once. This prevents future instances of the problem from recurring.

OOP is a way to model the world around you. Java is merely the digital representation of that model. What you’re learning in CS15 is incredibly powerful, regardless of whether you choose to have a career in technology!

---

Responsible CS (1/2)

GPT-2 supposedly generates human-like text

Full model considered “too dangerous,” not released yet

Smaller models released to select groups

Harvard researchers working on fake text detection technology specifically trained on GPT-2

Sources:

Responsible CS (1/2)

Consequences

Researchers can use the model to better classify which articles are real vs. fake. Other parties can use the technologies to generate fake news that can confuse the public and influence their decisions.

Was OpenAI's decision to release its model ethical?

Important Reminders and Announcements

- Please carefully read all relevant documents before reaching out to a TA via email or on hours.
  - Read the missive (and all docs under "Required Reading" on the website).
  - "Read the TA Hours Policy!"
  - Read the handout thoroughly before visiting hours.
    - This helps the TAs out a lot and saves a lot of time!
  - Visit conceptual hours if you don't understand something conceptually from a project; this is the best place to go!
- TopHat Questions: check them in advance and be prepared to answer on TopHat website or App.
  - Slides are on the website prior to the start of class.

if-else: Flow Chart
Complex if-else Statements

- If `<boolean expression 1>` is true, block 1 is executed and blocks 2 and 3 are skipped.
- If `<boolean expression 1>` is false and `<boolean expression 2>` is true, block 2 is executed and blocks 1 and 3 are skipped.
- If both expressions are false, block 3 is executed and blocks 1 and 2 are skipped.

```java
if (<boolean expression 1>) {
    // block 1
} else if (<boolean expression 2>) {
    // block 2
} else {
    // block 3
}
```

Nested if Statements

// variables and methods made up

```java
if (cs15Student.hasProject()) {
    if (cs15Student.hasInitiative()) {
        cs15Student.workOnProject();
    } else {
        cs15Student.playBasketball();
    }
}
```

TopHat Question

Which print statement will be printed out?

```java
int x = 10;
if (x > 10) {
    if ((x+10)>15) {
        System.out.println("case A");
    } else {
        System.out.println("case B");
    }
} else if (x <= 15) {
    if ((x+2) > 13) {
        System.out.println("case C");
    } else {
        System.out.println("case D");
    }
} else {
    System.out.println("case E");
}
```

A → B → C → D → E →
Short-Circuiting (1/2)

- What is the value of \( n \) after the code to the right has executed?
- \( n \) is still 1!
- Why?

```java
int n = 1;
if ((n < 0) && (n++ == 2)) {
    // code to be executed if expression is true
}
System.out.println(n);
```

Short-Circuiting (2/2)

- Beware of short-circuiting!
- If Java already knows what the full expression will evaluate to after evaluating left argument, no need to evaluate right argument
  - \&\&: if left argument of conditional evaluates to false, right argument not evaluated
  - ||: if left argument evaluates to true, right argument not evaluated

```java
int n = 1;
if ((n == 1) || (n == 2)) {
    // code to be executed if expression is true
}
```

“Side-effect”ing

- Updating a variable inside a conditional is not good coding style; it makes code confusing and hard to read
- Keep in mind short-circuiting if you ever call a method that might have a "side effect" inside a conditional – here the first if will leave \( n \) incremented

```java
int n = 1;
if ((n++ == 2) && false) {
    // code to be executed if expression is true
}
System.out.println(n);
```
If want to do something different for every possible value of an integer variable, have two options:

1. use a lot of else-if:
   ```java
   if (myInteger == 0) {
       // do something...
   } else if (myInteger == 1) {
       // do something else...
   } else if (myInteger == 2) {
       // do something else...
   } else if (myInteger == 3) {
       // etc…
   }...
   else {
       // last case
   }
   ```

2. better solution: use a switch statement!

### Switch Statements (2/2)

**Syntax:**

```java
switch (<variable>) {
    case <value>:
        // do something
        break;
    case <other value>:
        // do something else
        break;
    default:
        // take default action
        break;
}
```

**Rules:**

- `<variable>` usually an integer; char and enum (discussed later) also possible
- `values` have to be mutually exclusive
- If `default` is not specified, Java compiler will not do anything for unspecified values
- `break` indicates the end of a case — skips to end of switch statement (if you forget break, the code in next case will execute)

Let’s make a `PaperCompany` that produces a different colored paper using a switch statement.

The sheet is chosen by weighted distribution (more orange, red, blue, and fewer brown, green, yellow).

`PaperCompany` generates random value using `Math`.

Based on random value, creates and returns a sheet of a particular type.
• To generate a random value, we use static method `Math.random()` from `java.lang.Math`.
• `random` returns a `double` between 0.0 (inclusive) and 1.0 (exclusive).
• This line returns a random `int` 0-9 by multiplying the value returned by `random` by 10 and casting the result to an `int`.
• Casting is a way of changing the type of an object to another specified type. Casting from a `double` to `int` truncates your `double`!

• We initialize our `Sheet` to `null`, and `switch` on the random value we’ve generated.

• Sheet takes in an instance of `javafx.scene.paint.Color` as a parameter of its constructor (needs to know what color it’s)
• Once you import `javafx.scene.paint.Color`, you only need to say, for example, `Color.RED` to name a color.
• If random value turns out to be 0 or 1, instantiate an orange `Sheet` and assign it to sheet.
• `Color.ORANGE` is a constant of type `Color`.
• `break` breaks us out of `switch` statement.
**switch Example (5/6)**

```java
public class PaperCompany{
    public Sheet generatePaper() {
        int rand_int = (int) (Math.random() * 10);
        Sheet sheet = null;
        switch (rand_int) {
            case 0: case 1:
                sheet = new Sheet(Color.ORANGE);
                break;
            case 2: case 3: case 4:
                sheet = new Sheet(Color.RED);
                break;
            // cases 5, 6, and 7 elided.
            // they are green, blue, yellow.
            default:
                sheet = new Sheet(Color.BROWN);
                break;
        }
        return sheet;
    }
}
```

- If our random value is 2, 3, or 4, we instantiate a red Sheet and assign it to `sheet`.
- `Color.RED` is another constant of type `Color` – check out javadocs for `javafx.scene.paint.Color`!

**switch Example (6/6)**

```java
public class PaperCompany{
    public Sheet generatePaper() {
        int rand_int = (int) (Math.random() * 10);
        Sheet sheet = null;
        switch (rand_int) {
            case 0: case 1: case 2: case 3:
                sheet = new Sheet(Color.ORANGE);
                break;
            case 4: case 5: case 6: case 7:
                sheet = new Sheet(Color.RED);
                break;
            // cases 5, 6, and 7 elided.
            // they are green, blue, yellow.
            default:
                sheet = new Sheet(Color.BROWN);
                break;
        }
        return sheet;
    }
}
```

- We skipped over the cases for values of 5, 6, and 7; assume they create green, blue, and yellow sheets, respectively.
- Our `default` case (if random value is 8 or 9) creates a brown sheet.
- Last, we return `sheet`, which was initialized with a color depending on the value of `rand_int`.

---

**TopHat Question**

Which of the following `switch` statements is correct?

- **A.**
  ```java
  int rand = (int) (Math.random() * 10);
  CompanyBranch companyBranch = null;
  switch (rand) {
          companyBranch = null;
          break;
      default:
          companyBranch = new CompanyBranch("Corporate");
          break;
  }
  ```

- **B.**
  ```java
  int rand = (int) (Math.random() * 10);
  CompanyBranch companyBranch = null;
  switch (rand) {
          companyBranch = null;
          break;
      default:
          companyBranch = new CompanyBranch("Corporate");
          break;
  }
  ```

- **C.**
  ```java
  Place place = Place.random();
  CompanyBranch companyBranch = null;
  switch (place) {
      case Scranton:
          companyBranch = new CompanyBranch("Scranton");
          break;
      case Stamford:
          companyBranch = new CompanyBranch("Stamford");
          break;
      default:
          companyBranch = new CompanyBranch("Corporate");
          break;
  }
  ```
Method Overloading (1/3)

• Can define multiple methods of same name within a class, as long as method signatures are different
• Method signature: name, number and types of parameters and their order
• Signature does NOT include return type
• Two methods with identical signatures but different return types (and different bodies) will yield a compiler error – why?
• Compiler (and you, the reader) can’t distinguish between two methods with the same signature and different return types when an instance calls those methods - method name and argument types passed in are exactly the same! So, signature is just name and parameter list

/* this is an approximation to what Math's three max methods look like */

```java
public class Math {
    // other code elided
    public static int max(int a, int b) {
        // return max of two ints
    }
    public static float max(float a, float b) {
        // return max of two floats
    }
    public static double max(double a, double b) {
        // return max of two doubles
    }
}
```

TopHat Question

Which of the following is true of a class that contains an overloaded method? The class has...

A. Two methods that are absolutely identical
B. Two methods that are the same, except in their return type
C. Two methods that have the same name, but different parameters
D. Two methods that are the same, except one contains an error

Method Overloading (2/3)

• Example: java.lang.Math
• `static` method `max` takes in two numbers and returns the greater of the two
• There are actually three `max` methods – one for `int`s, one for `float`s, one for `double`s
• When you call an overloaded method, the compiler infers which method you mean based on types and number of arguments provided

/* this is an approximation of what Math's three max methods look like */

```java
public class Math {
    // other code elided
    public static int max(int a, int b) {
        // return max of two ints
    }
    public static float max(float a, float b) {
        // return max of two floats
    }
    public static double max(double a, double b) {
        // return max of two doubles
    }
}
```
Method Overloading (3/3)

- Be careful not to confuse overloading and overriding!
  - Overriding an inherited method in a subclass: the signatures must be the same
  - Overloading methods within the same class: names are the same but the rest of the signatures must be different so the compiler can differentiate
- Using same signatures in different classes is OK because the compiler can differentiate by class/type of instance on which the method is called

Method Overloading: Constructors

- Even constructors can be overloaded! Cook class has multiple constructors
- A String (java.lang.String) is a sequence of alphanumeric characters, including space!
- Example:
  ```java
  public class Cook {
    private String _dessert, _entree;
    public Cook() {
      _dessert = "Birthday cake";
      _entree = "Sandwich";
    }
    public Cook(String dessert) {
      _dessert = dessert;
      _entree = "Sandwich";
    }
    public Cook(String dessert, String entree) {
      _dessert = dessert;
      _entree = entree;
    }
  }
  ```
  The above code would print out CS15 Rocks! in the console

Method Overloading: Example

- An overloaded method can call other overloaded methods

```java
public class FriendMakeover{
  public FriendMakeover(Wardrobe wardrobe) {
    Hat hat = wardrobe.getHat();
    this.wearAwesomeOutfit(hat);
  }
  public void wearAwesomeOutfit(Hat hat) {
    Tie tie = hat.getMatchingTie();
    this.wearAwesomeOutfit(hat, tie);
  }
  public void wearAwesomeOutfit(Hat hat, Tie tie) {
    //code to wearAwesomeOutfit elided
  }
  //other methods elided
}
```
That’s It!

Important Concepts:
- static methods and static variables
- Constants
- boolean
- Making decisions with if, if-else, switch
- Method overloading
- Method signatures: (name, number and types of parameters)