Apply to Coordinate The Artemis Project

- Looking for a fun and rewarding summer job in computer science?
- Want to make a difference in the community while teaching what you love?
- Apply to coordinate the Artemis Project, a 5 week Computer Science Camp for local 9th grade girls!

artemis@cs.brown.edu
Lecture 8

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>a += 5;</td>
<td>a = a + 5;</td>
</tr>
<tr>
<td>-=</td>
<td>subtract and reassign</td>
<td>a -= 5;</td>
<td>a = a - 5;</td>
</tr>
<tr>
<td>*=</td>
<td>multiply and reassign</td>
<td>A *= 5;</td>
<td>a = a * 5;</td>
</tr>
<tr>
<td>/=</td>
<td>divide and reassign</td>
<td>A /= 5;</td>
<td>a = a / 5;</td>
</tr>
<tr>
<td>%=</td>
<td>take remainder and reassign</td>
<td>a %= 5;</td>
<td>a = a % 5;</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>b = -b; // negates b</td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>b++; // equivalent to: b = b + 1;</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td>b--; // equivalent to: b = b - 1;</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`
  - `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 class, part of core Java libraries
- Provides methods for basic numeric operations
  - absolute value: `abs(double a)`
  - exponential: `pow(double a, double b)`
  - natural and base 10 logarithm: `log(double a)`, `log10(double a)`
  - square root: `sqrt(double a)`
  - trigonometric functions: `cos(double a)`, `sin(double a)`...
  - random number generation: `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
- For example: `public static int abs(int a) {...}` returns the absolute value of an integer.
- A static method belongs to a class, rather than instance of the class
  - cannot access instance variables, whose values may differ from instance to instance
  - and 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 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
Clicker Question

Which is the correct way to call the static method `numberOfFish()` on the Ocean class?

A. `OceanInstance.numberOfFish();`
B. `Ocean.numberOfFish(static);`
C. `OceanInstance.numberOfFish(static);`
D. `Ocean.numberOfFish();`
**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 *different* copies of the class’s instance variables to allow different values of those properties

- If we want all instances of a class to share the *same* value for a variable, declare it *static*

- Each time any instance changes the value of a static variable, all instances have access to that new value
**static Variables: Simple Example**

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

```java
public class Example {
    private static int __numberOfInstances = 0;

    public Example() {
        __numberOfInstances++;
    }

    public static int getNumInstances() {
        return __numberOfInstances;
    }
}
```
Constants

- Constants are used to represent values which never change (e.g. Pi, speed of light, etc.)

- Keywords used when defining a constant:
  - `static`: all instances of the class share one value
  - `final`: value cannot be reassigned
  - `public`: value should be available for use by anyone

- Naming convention for constants is all caps with underscores between words: `LIGHT_SPEED`
Constants: Example (1/2)

```java
public class Physics {

    // speed of light (Units: hundred million m/s)
    public static final double LIGHT_SPEED = 2.998;

    // constructor elided

    public double getDistanceTraveled(double numSeconds) {
        return (LIGHT_SPEED * numSeconds);
    }
}
```
• Always use constants when possible
  o literal numbers, except for 0 and 1, should rarely appear in your code
  o makes code readable, easy to maintain

• If many classes use same constants, make separate utility class, like `PhysicsConstants`

• A constants utility class should never be instantiated, so it should be declared `abstract`

```java
public abstract class PhysicsConstants {
    //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:
```
PhysicsConstants.LIGHT_SPEED
```

(another use of dot notation!)
Workout (1/6)

- Theodor Seuss Geisel (Dr. Seuss’s real name!) 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
    static final double START_WEIGHT = 1;

    // Don’t want him to look like this:
    static final double MAX_WEIGHT = 10;
}
```
Workout (3/6)

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

```java
import java.lang.Math;
public class Seuss extends Writer {

    private double _weight;

    public Seuss() {
        _weight = WorkoutConstants.START_WEIGHT;
    }

    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```
Workout (4/6)

- **Seuss's `gainMuscle` method** changes his weight according to the amount of time he works out

```java
import java.lang.Math;
public class Seuss extends Writer {

    private double _weight;

    public Seuss() {
        _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 Seuss extends Writer {
    private double _weight;

    public Seuss() {
        _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 *workout* method

• Seuss 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 Seuss extends Writer {
    private double _weight;

    public Seuss() {
        _weight = WorkoutConstants.START_WEIGHT;
    }

    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }

    public void workout() {
        //code to workout!
    }
}
```
boolean's

- boolean (named after British logician George Boole, 1815-1864) is another 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:
  ```java
  boolean b1 = (3 > 2);
  boolean b2 = (5 == 5);
  int x = 8;
  boolean b3 = (x <= 6);
  ```
  - b1 and b2 are true, b3 is false

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
</tbody>
</table>
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`: `s1` and `s2` initially refer to different instances
  2. `true`: `s1` and `s2` refer to the same instance
  3. `true`: `s1` is not `null`

```java
public class TestClass {
    //constructor elided
    public void compareReferences() {
        Student s1 = new Student();
        Student s2 = new Student();

        boolean sameStudent = (s1 == s2);
        System.out.println(sameStudent);

        s2 = s1;
        sameStudent = (s1 == s2);
        System.out.println(sameStudent);

        boolean student1Exists = (s1 != null);
        System.out.println(student1Exists);
    }
}
```
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 continues on with rest of method
if Statement: Flow Chart

Previous Statements

Is condition true?

Yes

Execute if clause

No

Execute rest of method
if Statements: Examples

```java
int x = 6;
if (x == 5) {
    // code to execute if x is 5
}

if (myBoolean) {
    // code to execute if myBoolean is true
}

int y = 9;
//more code elided – y is not reassigned
if (y > 7) {
    // code to execute if y is greater than 7
}
```
Logical Operators: And, Or, Not (1/2)

• Logical operators `&&` (“and”) and `||` (“or”) can be used to combine two boolean expressions
  o `<expression a> && <expression b>` evaluates to true only if both expressions are true
  o `<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.
## Logical Operators: And, Or, Not (2/2)

| 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 |
Clicker Question

Which if statement will run if the Grinch does not steal Christmas and Horton does hear a who?

A. if (!grinch.stole() && horton.hears()){}
B. if (grinch.stole() && !horton.hears()){}
C. if (grinch.stole() && horton.hears()){}
D. if (!grinch.stole() && !horton.hears()){}
if Statements: More Examples

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

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

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

```plaintext
int x = 6;
if (x == 5 || x == 6) {
    // code to execute if x is 5 or 6
}

if (firstBoolean && secondBoolean) {
    // code to execute if both booleans are true
}

if (!myBoolean) {
    // code to execute if myBoolean false
}
```
if-else

- 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

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

• If Seuss’s WEIGHT is not greater than the maximum WEIGHT when the method is called, he gains muscle

• Otherwise, he stops and writes!

• does this code limit the final calf weight to MAX_WEIGHT?

import java.lang.Math;

public class Seuss extends Writer{

    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); //workout for 60 minutes! Phew!
        } else {
            this.stopAndWrite();
        }
    }
}
if-else: Flow Chart

1. Previous Statements
2. Is condition true?
   - Yes: Execute if clause
   - No: Execute else clause
3. Execute rest of method
Complex if-else Statements

• If expression 1 is true, block 1 is executed and blocks 2 and 3 are skipped

• If expression 1 is false and 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

if(cs15Student.hasProject()) {
    if(cs15Student.hasInitiative()) {
        cs15Student.workOnProject();
    } else {
        cs15Student.playMarioKart();
    }
}
Clicker Question

Which print statement will be printed out?

```java
int x = 10;
if (x < 10) {
    if ((x+10)>15) {
        System.out.println("x + 10 is greater than 15");
    } else {
        System.out.println("x is less than 10");
    }
} else if (x <= 15) {
    if ((x+2) > 13) {
        System.out.println("x + 2 is greater than 13");
    } else {
        System.out.println("x is less than or equal to 11");
    }
} else {
    System.out.println("x is greater than 15");
}
```

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 we already know what the full expression will evaluate to after evaluating left-hand side, no need to evaluate right-hand side
  - `&&`: if left side of conditional evaluates to `false`, right side not evaluated
  - `||`: if left side evaluates to `true`, right side not evaluated

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

```c
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

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

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

• If want to do something different for every possible value of a variable, have two options:
  
  o use a lot of `else-ifs`:
    ```
    if (myInteger == 0) {
        // do something...
    } else if (myInteger == 1) {
        // do something else...
    } else if (myInteger == 2) {
        // do something else...
    } else if {
        // etc...
    }
    ```

  o better solution: use a `switch` statement!
switch Statements (2/2)

Syntax:

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

Rules:

• <variable> usually an integer; char and enum also possible
• values have to be mutually exclusive
• If default is not specified, Java 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)
switch Example (1/6)

• Let’s make an Thneed Factory that produces different colored Thneeds using a `switch` statement— a Factory is a fancier kind of constructor that can do arbitrary computation.

• Thneed colors chosen by weighted distribution (more red, orange, brown and fewer green, yellow, blue).

• Factory generates random value using `Math`.

• Based on random value, creates and returns a Thneed of a particular color.

```java
// imports elided-- Math and Color
public class ThneedFactory{
    // constructor elided
    public Thneed getThneed() {
        // code elided
    }
}
```
To generate a random value, we use static method `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 from a double to int truncates your int!
We initialize our Thneed to `null`, and `switch` on the random value we’ve generated.
• **Thneed** takes in an instance of `javafx.scene.paint.Color` as a parameter of its constructor (needs to know what color it is)

• If random value turns out to be 0 or 1, instantiate an orange **Thneed** and assign it to `thneed`

• `Color.ORANGE` is a constant of type `Color`

• `break` breaks us out of `switch` statement

```java
public class ThneedFactory {
    // constructor elided
    public Thneed getThneed() {
        int rand = (int) (Math.random() * 10);
        Thneed thneed = null;
        switch (rand) {
            case 0: case 1:
                thneed = new Thneed(Color.ORANGE);
                break;
        }
    }
}
```
If our random value is 2, 3, or 4, we instantiate a red `Thneed` and assign it to `thneed`

- `Color.RED` is another constant of type `Color`—check out `javadoc` for `javafx.scene.paint.Color`!
switch Example (6/6)

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

```java
// imports elided-- Math and Color
public class ThneedFactory{
    // constructor elided
    public Thneed getThneed() {
        int rand = (int) (Math.random() * 10);
        Thneed thneed = null;
        switch (rand) {
            case 0: case 1:
                thneed = new Thneed(Color.ORANGE);
                break;
            case 2: case 3: case 4:
                thneed= new Thneed(Color.RED);
                break;
            // cases 5, 6, and 7 elided.
            // they are green, blue, yellow.
            default: //create a brown Thneed
                thneed= new Thneed(Color.rgb(150, 100,0));
                break;
        }
        return thneed;
    }
}
```
Method Overloading (1/4)

- 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)

```java
/* this is an approximation to what Math’s three max methods look like */

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
    }
}
```
Clicker Question

What is a valid way to have overloaded methods?

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 and/or return types.
D. Two methods that are the same, except one contains an error
Method Overloading (2/4)

• 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 `ints`, one for `floats`, one for `doubles`
Method Overloading (3/4)

• When you call an overloaded method, Java infers which method you mean based on types and number of arguments provided.

• A class cannot contain two methods with identical signatures but different return types—this yields a compiler error.

```java
/* this is an approximation of what Math’s three max methods look like */

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 (4/4)

• 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 Java can differentiate

• Using same signatures in different classes is OK because Java 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 alphanumerical characters, including space!
- Example:

```java
public class Cook {
    private String _dessert, _entree;

    public Cook() {
        _dessert = "Green Eggs";
        _entree = "Ham";
    }

    public Cook(String dessert) {
        _dessert = dessert;
        _entree = "Ham";
    }

    public Cook(String dessert, String entree) {
        _dessert = dessert;
        _entree = entree;
    }
}
```

`String s = "CS15 Rocks!";` System.out.println(s) would print out `CS15 Rocks!` in the console.

When instance variables are of the same type, they can be declared with a comma separation.
Method Overloading: Example

• An overloaded method can call other overloaded methods

```java
public class CatInTheHat{

    public CatInTheHat(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
• **booleans**
• Making decisions with **if**, **if-else**, **switch**
• Method overloading
• Method signatures: (name; number and types of parameters)
Announcements

• TASafeHouse goes out today!
  o Design discussions next week M-W
  o Mini-assignment due Monday at 2 PM
    ▪ Sign up for your design discussion by Sunday
  o Review sessions on inheritance, polymorphism

• Working from Home help session from 5-7pm on Sunday, October 2 in MacMillan 117