Piazza Stats

- 585 total posts
- 792 TA responses
- 5 min average response time!

→ Use Piazza! (But do follow the Piazza rules)

---

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

- **Postfix example:**
  - int i = 10;
  - int j = i++; // j becomes 10, i becomes 11

- **Prefix example:**
  - 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
  o absolute value: abs(double a)
  o exponential: exp(double a)
  o natural and base 10 logarithm: log(double a), log10(double a)
  o square root: sqrt(double a)
  o trigonometric functions: cos(double a), sin(double a)...
• random number generation: random() returns random number from 0.0 to 1.0
  o for more check out: 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
  o cannot access instance variables, whose values may differ from instance to instance

Calling a static Method

• static methods are invoked on the class, not on an instance:
  
  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

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**

```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);
    }
}
```

**Constants: Example**

- Always use constants when possible
  - literal numbers, except for 0 and 1, should rarely appear in your code
  - 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 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
Workout (1/6)

- Hank 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
  - 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)

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

Workout (4/6)

- Hank’s gainMuscle method changes his weight according to the amount of time he works out
**Workout (5/6)**

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

```java
import java.lang.Math;
public class Hank extends DEAofficer {
    private double _weight;
    public Hank() {
        _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
- Hank will only work out if weight is not above maximum WEIGHT
- How can we check if condition is met?
- Introducing... booleans!

```java
import java.lang.Math;
public class Hank extends DEAofficer {
    private double _weight;
    public Hank() {
        _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!
    }
}
```

---

**booleans**

- 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;
  ```

**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);
  boolean 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 {
    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

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;
if (y > 7) {
    // code to execute if y is greater than 7
}
```
Logical Operators: And, Or, Not

- 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.

### 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 `==`

```java
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 is true
}
```

### 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

import java.lang.Math;

public class Hank extends DEAofficer {

    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);
        } else {
            this.stopAndDrinkBeer();
        }
    }
}

if-else: Flow Chart

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

Nested if Statements

// variables and methods made up
if(cs1Student.hasProject()) {
    if(cs1Student.hasInitialive()) {
        cs1Student.workOnProject();}
    else {
        cs1Student.playMarioKart();
    }
}
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 (false && (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

```java
int n = 1;
if (false && (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);
```

switch Statements (1/2)

- If want to do something different for every possible value of a variable, have two options:
  - use a lot of else-ifs:
    ```java
    int n = 1;
    if (myInteger == 0) {
        // code to be executed if
        // expression is true
    }
    else if (myInteger == 1) {
        // code to be executed if
        // expression is true
    }
    else if (myInteger == 2) {
        // code to be executed if
        // expression is true
    }
    else if {
        // etc...
    }
    System.out.println(n);
    ```
  - 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)

public class MmFactory {
    public Mm getMm() {
        // imports elided-- Math and Color
    }
}

• Let's make an M&M Factory that produces different colored M&Ms using a switch statement—Factory is a fancier kind of constructor
• M&M 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 an M&M of a particular color

switch Example (2/6)

public class MmFactory {
    public Mm getMm() {
        int rand = (int)(Math.random() * 10);
        Mm mnm = null;
        switch (rand) {
            // imports elided-- Math and Color
        }
    }
}

• 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

switch Example (3/6)

public class MmFactory {
    public Mm getMm() {
        int rand = (int)(Math.random() * 10);
        Mm mnm = null;
        switch (rand) {
            // imports elided-- Math and Color
        }
    }
}

• We initialize our Mm to null, and switch on the random value we've generated
**switch Example (4/6)**

- MnM takes in an instance of `java.awt.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 MnM and assign it to `mnm`
- `Color.ORANGE` is a constant of type `Color`
- `break` breaks us out of `switch` statement

```java
public class MnMFactory {
    // constructor elided
    public MnM getMnM() {
        int rand = (int) (Math.random() * 10);
        MnM mnm = null;
        switch (rand) {
            case 0: case 1:
                mnm = new MnM(Color.ORANGE);
                break;
        }
    }
}
```

**switch Example (5/6)**

- If our random value is 2, 3, or 4, we instantiate a red MnM and assign it to `mnm`
- `Color.RED` is another constant of type `Color` – check out javadocs for `java.awt.Color`!

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

**switch Example (6/6)**

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

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

**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
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
    }
}
```

- This is an approximation of what Math's three max methods look like */
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 `int`s, one for `float`s, one for `double`s

```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 (3/4)

- 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
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

```java
public class Cook {
    private String _dessert, _entree;
    public Cook() {
        _dessert = "Creeme brullee";
        _entree = "Pizza";
    }
    public Cook(String dessert) {
        _dessert = dessert;
        _entree = "Pizza";
    }
    public Cook(String dessert, String entree) {
        _dessert = dessert;
        _entree = entree;
    }
    System.out.println(s); // would print out "CS15 Rocks!" in the console
}
```

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 = "Creeme brullee";
        _entree = "Pizza";
    }
    public Cook(String dessert) {
        _dessert = dessert;
        _entree = "Pizza";
    }
    public Cook(String dessert, String entree) {
        _dessert = dessert;
        _entree = entree;
    }
    System.out.println(s); // would print out "CS15 Rocks!" in the console
}
```
Method Overloading: Example

• An overloaded method can call other overloaded methods
  ```java
  public class Sharpay {
      public Sharpay(Wardrobe wardrobe) {
          Dress dress = wardrobe.getDress();
          this.wearFabulousOutfit(dress);
      }
      public void wearFabulousOutfit(Dress dress) {
          Heels heels = dress.getMatchingHeels();
          this.wearFabulousOutfit(dress, heels);
      }
      public void wearFabulousOutfit(Dress dress, Heels heels) {
          //code to wearFabulousOutfit 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 early handin is tonight at 11:59PM
    • on-time handin is Thursday at 11:59PM
    • late handin is Saturday 10:00 PM
  • We changed (pushed back!) some of the due dates for the next assignment (Cartoon) -- please see the email we sent out
  • We are now holding you accountable for our TA Hours policies:
    • TAs can only help with the question you entered in SignMeUp (“TASafeHouse question” is too vague)
    • you will be turned away for bug help if you can't show evidence of debugging on your own (Use the tools you learned in Lab 2!)
    • introducing the 'No-Code' line on the whiteboard