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>
<th>Example</th>
<th>Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>a += 5;</td>
<td>a = a + 5;</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>a -= 5;</td>
<td>a = a - 5;</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>a *= 5;</td>
<td>a = a * 5;</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>a /= 5;</td>
<td>a = a / 5;</td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
<td>a %= 5;</td>
<td>a = a % 5;</td>
</tr>
</tbody>
</table>

Basic Arithmetic Operators: Shorthand
### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>negate</td>
<td><code>b = -b;</code> // negates b</td>
</tr>
<tr>
<td><code>++</code></td>
<td>increment</td>
<td><code>b++;</code> // equivalent to: <code>b = b + 1;</code></td>
</tr>
<tr>
<td><code>--</code></td>
<td>decrement</td>
<td><code>b--;</code> // equivalent to: <code>b = b - 1;</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`
  - `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
  - `absoluteValue.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 CS10, 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

**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);
    }
}
```

**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
- 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

**static Variables: Simple Example**

```java
public class Example {
    private static int _numberOfInstances = 0;
    public Example() {
        _numberOfInstances++;
    }
    public static int getNumInstances() {
        return _numberOfInstances;
    }
}
```

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

```java
public 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)**

- 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 the time of his workout
- 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;
    // code to workout!
    public void workout(){
        _weight += workoutTime_* effort;
    }
}
```

Workout (3/6)

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

```java
public class Seuss extends WorkoutConstants{
    //workout
    private double _weight;
    public Seuss(double weight){
        _weight = 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 = workoutTime_* 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 = workoutTime_* 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 = workoutTime_* START_WEIGHT;
    }
    public void gainMuscle(double workoutTime){
        double effort = workoutTime_* _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```

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;
  ```
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);
  - 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 {
    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 Statements: Examples

```java
int x = 4;
if (x == 5) {
    // code to execute if x is 5
}
if (symbolism) {
    // code to execute if symbolism is true
}
int y = 9;
// 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
  - <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.
Logical Operators: And, Or, Not (2/2)

| A   | B   | A && B | A || B | A\(^\oplus\)B | !A |
|-----|-----|--------|--------|-------------|----|
| false | false | false   | false   | false       | true |
| false | true  | false   | true    | false       | true |
| true  | false | true    | true    | false       | true |
| 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 ==

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: Flow Chart

```java
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 = (1.0 / 10) * Math.sqrt(effort + _weight);
    _weight += muscleGained;
  }
  public void calfWorkout() {
    if (_weight < WorkoutConstants.MAX_WEIGHT) {
      this.gainMuscle(60); // workout for 60 minutes!
    } else {
      this.stopAndWrite();
    }
  }
}
```
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();
    }
} else {
    // block 3
}

Clicker Question

Which print statement will be printed out?

A ->
```
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");
    }
}
```

B ->
```
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");
    }
}
```

C ->
```
else {
    System.out.println("x is greater than 15");
}
```

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

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

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

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

```
int n = 1;
if (false && (x++ == 23)) {
    // code to be executed if expression is true
}
System.out.println(n);
```

“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 && (x++ == 23)) {
    // code to be executed if expression is true
}
System.out.println(n);
```

```java
int x = 1;
if ((x++ == 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:
  - 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 {  // etc...
    }
  - best 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
}
```

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 — skipping to end of switch statement (if you forget `break`, the code in next case will execute)

switch Example (1/6)

- Let’s make an `ThneedFactory` that produces different colored `Thneeds` using a switch statement—`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
public class ThneedFactory {
  // constructor elided
  public Thneed getThneed() {
    // imports elided -- Math and Color
    int rand = (int) (Math.random() * 10);
    Thneed thneed = null;
    switch (rand) {
      case 0: case 1:
        thneed = new Thneed(Color.ORANGE);
        break;
    }
  }
}
```

switch Example (2/6)

- 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!

```java
public class ThneedFactory {
  // constructor elided
  public Thneed getThneed() {
    // imports elided -- Math and Color
    int rand = (int) (Math.random() * 10);
    Thneed thneed = null;
    switch (rand) {
      case 0: case 1:
        thneed = new Thneed(Color.ORANGE);
        break;
    }
  }
}
```

switch Example (3/6)

- We initialize our `Thneed` to null, and switch on the random value we’ve generated

```java
public class ThneedFactory {
  // constructor elided
  public Thneed getThneed() {
    // imports elided -- Math and Color
    int rand = (int) (Math.random() * 10);
    Thneed thneed = null;
    switch (rand) {
      case 0: case 1:
        thneed = new Thneed(Color.ORANGE);
        break;
    }
  }
}
```

switch Example (4/6)

- `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() {
    // imports elided -- Math and Color
    int rand = (int) (Math.random() * 10);
    Thneed thneed = null;
    switch (rand) {
      case 0: case 1:
        thneed = new Thneed(Color.ORANGE);
        break;
    }
  }
}
```
Method Overloading (1/4)

```
public class Math {

    // return max of two doubles
    public static double max(double a, double b) {
        // return one of two doubles
        return a > b ? a : b;
    }

    // return max of two floats
    public static float max(float a, float b) {
        // return one of two floats
        return a > b ? a : b;
    }

    // return max of two ints
    public static int max(int a, int b) {
        // return one of two ints
        return a > b ? a : b;
    }
}
```

Method Overloading (2/4)

```
public class Math {

    // return max of two doubles
    public static double max(double a, double b) {
        // return one of two doubles
        return a > b ? a : b;
    }

    // return max of two floats
    public static float max(float a, float b) {
        // return one of two floats
        return a > b ? a : b;
    }

    // return max of two ints
    public static int max(int a, int b) {
        // return one of two ints
        return a > b ? a : b;
    }
}
```

Method Overloading (3/4)

```
public class Math {

    // return max of two doubles
    public static double max(double a, double b) {
        // return one of two doubles
        return a > b ? a : b;
    }

    // return max of two floats
    public static float max(float a, float b) {
        // return one of two floats
        return a > b ? a : b;
    }

    // return max of two ints
    public static int max(int a, int b) {
        // return one of two ints
        return a > b ? a : b;
    }
}
```

Method Overloading (4/4)

```
public class Math {

    // return max of two doubles
    public static double max(double a, double b) {
        // return one of two doubles
        return a > b ? a : b;
    }

    // return max of two floats
    public static float max(float a, float b) {
        // return one of two floats
        return a > b ? a : b;
    }

    // return max of two ints
    public static int max(int a, int b) {
        // return one of two ints
        return a > b ? a : b;
    }
}
```

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 (4/4)

- Be careful not to confuse overloading and overriding!
  - Overloading 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: Example

- An overloaded method can call other overloaded methods

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
}

Method Overloading: Constructors

- Even constructors can be overloaded if the constructor has multiple constructors.
- A String(java.lang.String) is a sequence of alphanumeric characters, including space!
- Example:

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

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!
  - Design discussions next week M-W
  - Mini-assignment due Monday at 2 PM
    - Sign up for your design discussion by Sunday
  - Review sessions on inheritance, polymorphism
- SSH too slow? Working from Home help session!
  - 5-7pm on Sunday, October 2 in MacMillan 117