Lecture 8
Math and Making Decisions
Outline

• Abstract methods and classes
• Arithmetic operations – java.lang.Math
• Static methods and static variables
• Constants – values that never change
• Decision making: boolean algebra, if-else statements and the switch statement
Recall: Inheritance Example

- Remember the Race program from the last lecture
  - our transportation options include Convertible, CS15Mobile, and Van
  - each of these extend Car and inherit methods from it
  - we can represent this in an Inheritance/Interface Diagram
Outline

- Abstract methods and classes
- Arithmetic operations – java.lang.Math
- Static methods and static variables
- Constants – values that never change
- Decision making: boolean algebra, if-else statements and the switch statement
abstract Methods and Classes (1/6)

- What if we wanted to seat all of the passengers in the car?
- **CS15Mobile, Convertible,** and **Van** all have different numbers of seats
  - they will all have different implementations of the same method
We declare a method \textit{abstract} in a \textit{superclass} when the \textit{subclasses} can't really re-use any implementation the \textit{superclass} might provide – no code-reuse.

In this case, we know that all \texttt{Cars} should \texttt{loadPassengers}, but each \textit{subclass} will \texttt{loadPassengers} very differently.

\textit{abstract} method is declared in \textit{superclass}, but not defined – it is up to \textit{subclasses} farther down hierarchy to provide their own implementations.

Thus \textit{superclass} specifies a contractual obligation to its \textit{subclasses} – just like an interface does to its implementors.
abstract Methods and Classes (3/6)

- Here, we've modified `Car` to make it an abstract class: a class with at least one abstract method

- We declare both `Car` and its `loadPassengers` method abstract: if one of a class's methods is abstract, the class itself must also be declared abstract

- An abstract method is only declared by the superclass, not defined – thus use semicolon after declaration instead of curly braces

```java
public abstract class Car {
    private Racer driver;
    public Car(Racer driver) {
        this.driver = driver;
    }
    public abstract void loadPassengers();
}
```
How do you load *Passengers*?

- every *Passenger* must be told to *sit* in a specific *Seat* in a physical *Car*
- *SeatGenerator* has methods that returns a *Seat* in a specific logical position

```java
public class Passenger {
    public Passenger() { //code elided }
    public void sit(Seat st) { //code elided }
}

public class SeatGenerator {
    public SeatGenerator () { //code elided }
    public Seat getShotgun() { //code elided }
    public Seat getBackLeft() { //code elided }
    public Seat getBackCenter() { //code elided }
    public Seat getBackRight() { //code elided }
    public Seat getMiddleLeft() { //code elided }
    public Seat getMiddleRight() { //code elided }
}
```
abstract Methods and Classes (5/6)

- All concrete subclasses of `Car` override by providing a concrete implementation for `Car`'s abstract `loadPassengers()` method
- As usual, method signature and return type must match the one that `Car` declared
abstract Methods and Classes (6/6)

- **abstract** classes **cannot be instantiated!**
  - this makes sense – shouldn’t be able to just instantiate a generic `Car`, since it has no code to `loadPassengers()`
  - instead, provide implementation of `loadPassengers()` in concrete `subclass`, and instantiate `subclass`
- **Subclass** at any level in inheritance hierarchy can make an **abstract** method concrete by providing implementation
  - it’s common to have multiple consecutive levels of abstract classes before reaching a concrete class
- Even though an **abstract** class can’t be instantiated, its constructor must still be invoked via `super()` by a **subclass**
  - because only the superclass knows about (and therefore only it can initialize) its own instance variables
So.. What’s the difference?

- You might be wondering: what’s the difference between **abstract** classes and interfaces?
- **abstract** classes:
  - can define instance variables
  - can define a mix of concrete and **abstract** methods
  - you can only inherit from one class
- Interfaces:
  - cannot define any instance variables/concrete methods
  - has only undefined methods (no instance variables)
  - you can implement multiple interfaces

*Note: Java, like most programming languages, is evolving. In Java 8, interfaces and **abstract** classes are even closer in that you can have concrete methods in interfaces. We will not make use of this in CS15.*
Summary of Inheritance and Polymorphism

- **Inheritance** models very similar classes
  - factor out all similar capabilities into a generic superclass
  - **superclasses** can
    - declare and define methods
    - declare abstract methods
  - **subclasses** can
    - inherit methods from a superclass
    - define their own specialized methods
    - completely/partially override an inherited method

- **Polymorphism** allows programmers to reference instances of a subclass as their superclass

- Inheritance, Interfaces, and Polymorphism take generic programming to the max – more in later lecture
Outline

• Abstract methods and classes
• Arithmetic operations – java.lang.Math
• Static methods and static variables
• Constants – values that never change
• Decision making: boolean algebra, if-else statements and the switch statement
## 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; // negates b</code></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td><code>b++; // equivalent to: b = b + 1;</code></td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td><code>b--; // equivalent to: 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:**
```
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 “utility” 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: [https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)
Outline

• Abstract methods and classes

• Arithmetic operations – java.lang.Math

• Static methods and static variables

• Constants – values that never change

• Decision making: boolean algebra, if-else statements and the switch statement
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
  o `public static int abs(int a) {...}`
• A `static method` belongs to a class, rather than an instance of the class
  o it cannot access instance variables, whose values may differ from instance to instance
    ▪ but can have local variables, e.g., temps
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**

`playerCounter` is an instance of the **Survivor** class. Which is the correct way to call this static method:

```java
public static int playersLeft(){...}
```

A. `int contestants = Instance.playersLeft();`

B. `int contestants = Survivor.playersLeft(static);`

C. `int contestants = SurvivorInstance.playersLeft(static);`

D. `int contestants = Survivor.playersLeft();`

E. `int contestants = playerCounter.playersLeft();`
static Variables

• Progression in scope:
  o local variables are known in a single method
  o instance variables are known to all methods of a class
  o 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

- `players` starts out with a value of 0
- Each time a new instance of `Player` is created, `players` is incremented by 1
- Get current value at any point by calling: `Player.getNumPlayers();`
  - each instance of `Player` will have the same value of `players`
- `static` methods can use `static` and `local` variables – but not `instance` variables

```java
public class Player {
    private static int players = 0;
    public Player () {
        this.players++;
    }
    public static int getNumPlayers() {
        return this.players;
    }
}
```
Outline

• Abstract methods and classes

• Arithmetic operations – java.lang.Math

• Static methods and static variables

• Constants – values that never change

• Decision making: boolean algebra, if-else statements and the switch statement
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
  o literal numbers, except for 0 and 1, should rarely appear in your code
  o makes code readable, easier to alter
• 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 a method in another class in our program like this:

```java
Physics.LIGHT_SPEED
```

(another use of dot notation!)

Example:

```java
spaceShip.setSpeed(Physics.LIGHT_SPEED)
```
TopHat Question

Which of the following constants is defined correctly?

A. public static final int PLAYER_AGE;
B. public static final int PLAYER_AGE = 20;
C. public static int final PLAYER_AGE = 20;
D. private static final int PLAYER_AGE = 20;
Workout (1/6)

- Contestant Boston Rob decides to prepare for Survivor by trying 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 constants in our calculation

```java
public abstract class WorkoutConstants {

    // Weight
    public static final double START_WEIGHT = 150;

    // Don’t want him to gain more muscle than this
    public static final double MAX_WEIGHT = 200;
}
```
Workout (3/6)

- **Rob** keeps track of instance variable **weight**

- **weight** initialized in constructor to starting weight defined in **WorkoutConstants**

```java
import java.lang.Math;

public class Rob {
    private double weight;
    public Rob() {
        this.weight = WorkoutConstants.START_WEIGHT;
    }
}
```
Workout (4/6)

- Rob's `gainMuscle` method changes his weight depending on the amount of time he works out.

```java
import java.lang.Math;

class Rob {
    private double weight;

    Rob() {
        this.weight = WorkoutConstants.START_WEIGHT;
    }

    void gainMuscle(double workoutTime) {
        // code elided
    }
}
```
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 Rob {

    private double weight;

    public Rob() {
        this.weight = WorkoutConstants.START_WEIGHT;
    }

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

- Now fill in `calfWorkout()`
- Rob 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!
  - seen booleans in Pong assignment but let’s formally introduce them

```java
import java.lang.Math;

public class Rob {
    private double weight;

    public Rob() {
        this.weight = WorkoutConstants.START_WEIGHT;
    }

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

    public void calfWorkout() {
        //code to workout!
    }
}
```
Outline

- Abstract methods and classes
- Arithmetic operations – java.lang.Math
- Static methods and static variables
- Constants – values that never change
- Decision making: boolean algebra, if-else statements and the switch statement
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 Boole) is simplest Java base type
- A boolean variable can have value true or false
- Example initialization:

  boolean foo = true;
  boolean bar = false;

The terms foo, bar, etc. are often used as placeholder names in computer programming or computer-related documentation: derived 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: 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 (since foo = !(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 = !(d1 != d2);
        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 sameStudent = (s1 == s2);
        System.out.println(sameStudent);
        s2 = s1;
        sameStudent = (s1 == s2);
        System.out.println(sameStudent);

        boolean student1Exists = (s1 != null);
        System.out.println(student1Exists);
    }
}
```

A. System.out.println(sameStudent);

B. System.out.println(sameStudent);

C. 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 compiler 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
}
```

Not executed

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

```java
int y = 9;
// more code elided - y is not reassigned
if (y > 7) {
    // code to execute if y is greater than 7
}
```

Executed
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)

- To represent the values a logical operator may take, a **truth table** is used

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A &amp;&amp; B</td>
<td>A</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------</td>
<td>--------</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
TopHat Question

Which \textbf{if} clause statement will run if Rob is not eliminated and there is a challenge tomorrow? (The variables below are of type \texttt{boolean})

A. \texttt{if(!robEliminated \&\& !challengeTomorrow)\{\ldots\}}

B. \texttt{if(!robEliminated \&\& challengeTomorrow)\{\ldots\}}

C. \texttt{if(robEliminated \&\& !challengeTomorrow)\{\ldots\}}

D. \texttt{if(robEliminated \&\& challengeTomorrow)\{\ldots\}}
if Statements: More Examples

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

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

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

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

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

if (myBoolean == false) {
    // code to execute if myBoolean is false
    // code is inefficient
}
```
If we 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**

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

- Can use if-else to fill in the calfWorkout method
- If Rob’s weight is less than the maximum weight when the method is called, he gains muscle
- Otherwise, he stops and joins Survivor!
- Does this code limit the final calf weight to MAX_WEIGHT?

```java
import java.lang.Math;

public class Rob {
    private double weight;

    // constructor elided

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

    public void calfWorkout() {
        if (this.weight < WorkoutConstants.MAX_WEIGHT) { // this method defined elsewhere in the code
            this.gainMuscle(60.0); // workout for 60 minutes!
        } else {
            this.joinSurvivor();
        }
    }
}
```
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 defined elsewhere

```java
if (cs15Student.hasProject()) {
    if (cs15Student.hasInitiative()) {
        cs15Student.workOnProject();
    } else {
        cs15Student.doChallenge();
    }
}
```
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");
}
```
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 < 0) && (n++ == 2)) {
    // code to be executed if expression is true
}
```

```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, second not.

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

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

- Use a lot of `else-if` statements:
  ```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
  }
  ```

- 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
        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 `BandanaCreator` class that produces different colored bandanas for contestants using a switch statement.

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

`BandanaCreator` generates random values using `Math`.

Based on random value, creates and returns a bandana of a particular type.

This is an example of the "factory" pattern in object-oriented programming: it is a method that has more complicated logic than a simple assignment statement for each instance variable.
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 is a way of changing the type of an object to another specified type. Casting from a `double` to `int` truncates your `double`!
**switch** Example (3/6)

- We initialize `myBandana` to `null`, and `switch` on the random value we've generated

```java
// imports elided - Math and Color
public class BandanaCreator{
  // constructor elided
  public Bandana generateBandana() {
    int rand_int = (int) (Math.random() * 10);
    Bandana myBandana = null;
    switch (rand_int) {
      
    } }
}
```
switch Example (4/6)

• **Bandana** takes in an instance of `javafx.scene.paint.Color` as a parameter of its constructor (needs to know what color it is)

• Once you import `javafx.scene.paint.Color`, you only need to say, for example, `Color.ORANGE` to name a color of type `Color`

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

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

```java
public class BandanaCreator{
    // constructor elided
    public Bandana generateBandana() {
        // imports elided - Math and Color
        int rand_int = (int) (Math.random() * 10);
        Bandana myBandana = null;
        switch (rand_int) {
            case 0: case 1:
                myBandana = new Bandana(Color.ORANGE);
                break;
        }
    }
}
```
switch Example (5/6)

- If our random value is 2, 3, or 4, we instantiate a red `Bandana` and assign it to `myBandana`.
- `Color.RED` is another constant of type `Color` – check out Javadocs 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 Bandanas, respectively.
- Our **default** case (if random value is 8 or 9) creates a brown Bandana.
- Last, we return `myBandana`, which was initialized in this `switch` with a color depending on the value of `rand_int`.

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

Which of the following switch statements is correct?

- In the constructor for Island, the parameter is a character string.

A.
```java
int rand = (int) (Math.random() * 10);
Island island = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        island = new Island("Maui");
        break;
    case 4: case 5: case 6: case 7:
        island = new Island("Bali");
        break;
    default:
        island = new Island("Fiji");
        break;
}
```

B.
```java
int rand = (int) (Math.random() * 10);
Island island = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        island = new Island("Maui");
        break;
    case 4: case 5: case 6: case 7:
        island = new Island("Bali");
        break;
    default:
        island = new Island("Fiji");
        break;
}
```

C.
```java
Place place = Place.random();
Island island = null;

switch (place) {
    case Maui:
        island = new Island("Maui");
        break;
    case Bali:
        island = new Island("Bali");
        break;
    default:
        island = new Island("Fiji");
        break;
}
```
That’s It!

Important Concepts:

- static methods and static variables
- Constants
- booleans
- Making decisions with if, if-else, switch
Announcements

• TicTacToe late handin due today at 11:59 PM
• FruitNinja (handout and help slides) released tomorrow
  o Early handin: 10/10 (+2 bonus points)
  o On-time handin: 10/12
  o Late handin: 10/14 (-8 for late handin, but 3 late days to use throughout semester)
• Debugging Hours start tomorrow 10/06
  o More information on the course website
• Polymorphism section this week
  o email your section TAs mini-assignment on time
• Course calendar
  o Google Sheets
  o Google calendar
Topics in Socially Responsible Computing

Drones and Robotics
Robotics

• A machine that is controlled by computer(s) and can carry out complex actions (semi-) automatically
  ○ Not all robots are anthropomorphic!

• Types:
  ○ Pre-programmed
  ○ Autonomous
  ○ Humanoid
  ○ Tele-operated
  ○ Augmenting
Unmanned Automatic Vehicles (Drones)

• Deliver packages (to layperson)
• Useful for delivering medicine and supplies
  o Increase equity of medical care
  o Transport across unreliable terrain with minimal risk of contamination
    ▪ Zipline serves close to 2,500 hospitals and health facilities across Rwanda and Ghana during the pandemic
    ▪ Swoop Aero delivers to remote villages in Malawi, DRC, and Vanuatu
• Vaccine distribution for those living in “pharmacy deserts”
Unmanned Automatic Vehicles (Drones)

- **Documentation and activism**
  - 2016-2017 used by protestors at Standing Rock, ND
    - Revealed proximity of pipeline to tribe’s water
    - Recorded police brutality and provided counter-evidence to law enforcement’s false narratives
  - May 2020: photographer George Steinmetz exposed NYC COVID “mass grave”
    - Hart Island: where city buried COVID victims whose bodies were not claimed for private burial
      - Graves dug by Rikers Island jail inmates
    - NYPD confiscated drone + detained George, resulted in legal battle

- **High resolution images of dangerous or difficult to access places**
Unmanned Automatic Vehicles (Drones)

- Surveillance
  - Drones used for public and private surveillance
    - Ex. FBI used drones in 2015 to surveil protestors in Baltimore

- Warfare
  - UAVs have been used by US military since ~2001 to surveil and perform unmanned strikes
    - While use of drones saves lives of US soldiers
      - Distance enables error, dehumanization of targets, "gamification" of warfare
      - Up to 25% of victims are civilians
        - ~25% of civilian victims are children
Robotics @ Brown

Brown Robotics
Robotics, Learning and Autonomy at Brown

• Faculty
  o George Konidaris
  o Michael Littman
  o Stefanie Tellex
    ▪ Flying robots curriculum for high schoolers

• Groups
  o Humanity Centered Robotics Initiative
  o Humans 2 Robots Lab
  o Intelligent Robot Lab
  o RLAB (reinforcement learning and adaptive behavior) group
Back to Facebook

- Facebook products “harm children, stoke division, weaken our democracy, and much more” – former Facebook employee
  - Leaked tens of thousands of pages of internal documents
  - Hope for government regulation
- Internal research found toxic risks of Instagram
  - Threat to teen girls’ mental health
  - Prevalence of drug cartels and human traffickers
- Can follow WSJ investigative series
More reading that may be of interest!

- Are drones suitable tools for delivering medical supplies in developing countries? — GSMA
- This Aerial Photographer Captured Images of the Mass Burials on Hart Island. Then, the New York Police Department Confiscated His Drone — artnet
- Police Drones — New York Times
- FBI releases secret spy plane footage — ACLU
- Drone War — The Bureau Investigates
- Whistleblower to Congress: Facebook products harm children and weaken democracy — NPR
- The Facebook Files — WSJ