Lecture 8

Math and Making Decisions
Review: Inheritance and Polymorphism Summary

- **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
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
abstract Methods and Classes (2/6)

- We declare a method **abstract** in a **superclass** when the **subclasses** can’t really re-use any implementation the **superclass** might provide – no code-reuse
- In this case, we know that all **Cars** should **loadPassengers**, but each **subclass** will **loadPassengers** very differently
- **abstract** method is declared in **superclass**, but not defined – it is up to **subclasses** farther down hierarchy to provide their own implementations
- Thus **superclass** specifies a contractual obligation to its **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 myDriver) {
        this.driver = myDriver;
    }
    public abstract void loadPassengers();
}
```
abstract Methods and Classes (4/6)

- 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 }
}
```

```java
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.

```java
public class Convertible extends Car{
    @Override
    public void loadPassengers(){
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getShotgun());
    }
}

public class CS15Mobile extends Car{
    @Override
    public void loadPassengers(){
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getShotgun());
        this.passenger2.sit(seatGen.getBackLeft());
        this.passenger3.sit(seatGen.getBackCenter());
    }
}

public class Van extends Car{
    @Override
    public void loadPassengers(){
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getMiddleLeft());
        this.passenger2.sit(seatGen.getMiddleRight());
        this.passenger3.sit(seatGen.getBackLeft());
        //more code elided
    }
}
```
**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
Abstract Methods & Classes

• Abstract classes have 1 or more abstract methods

• An abstract method simply specifies a contractual application for a child class (at any level below parent) to provide a concrete implementation

• A class can NOT be instantiated if it is abstract

• An interface is simply an abstract class with NO code to inherit
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.
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><code>+=</code></td>
<td>add and reassign</td>
<td><code>a += 5;</code></td>
<td><code>a = a + 5;</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td>subtract and reassign</td>
<td><code>a -= 5;</code></td>
<td><code>a = a - 5;</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td>multiply and reassign</td>
<td><code>a *= 5;</code></td>
<td><code>a = a * 5;</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td>divide and reassign</td>
<td><code>a /= 5;</code></td>
<td><code>a = a / 5;</code></td>
</tr>
<tr>
<td><code>%=</code></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>( 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:**
```
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
```
• Extremely useful “utility” class, part of core Java libraries

• Provides methods for basic numeric operations
  o absolute value: `abs(double a)`
  o exponential: `pow(double a, double b)`
  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)...`
  o random number generation: `random()` returns random number from 0.0(inclusive) to 1.0(exclusive)
  o for more check out: 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
tributeCounter is an instance of the HungerGames class. Which is the correct way to call this static method of the HungerGames class:

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

A. `int tributesRemaining = Instance.numAlive();`
B. `int tributesRemaining = HungerGames.numAlive(static);`
C. `int tributesRemaining = HungerGamesInstance.numAlive(static);`
D. `int tributesRemaining = HungerGames.numAlive();`
E. `int tributesRemaining = tributeCounter.numAlive();`
**static Variables**

- Progression in scope:
  - **local** variables are known in a single method
  - **instance** variables are known to all methods of a class
  - **static** instance variables are known to all instances of a class

- Each instance of a class has the same instance variables but typically with **different** values for those properties

- If instead you want all instances of a class to share the **same** value for a variable, declare it **static** – this is not very common (and probably not used in CS15)

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

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

```java
public class Tribute {
    private static int tributes = 0;

    public Tribute () {
        this.tributes++;
    }

    public static int getNumTributes () {
        return this.tributes;
    }
}
```
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:
  o **public**: value should be available for use by anyone (unlike **private** instance variables and local variables)
  o **static**: all instances of the class share one value
  o **final**: value cannot be reassigned
  o 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);
    }
}
```

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.
Always use constants when possible
- literal numbers, except for 0 and 1, should rarely appear in your code
- 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
```

*Example:

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

Which of the following constants is defined correctly?

A. public static final int TRIBUTE_AGE;
B. public static final int TRIBUTE_AGE = 17;
C. public static int final TRIBUTE_AGE = 17;
D. private static final int TRIBUTE_AGE = 17;
Bread Makers (1/6)

- Peeta has entered a competition to see who can sell the most loaves of bread!
  - (don’t take this example too literally)
- Depending on the amount of dough and time to bake it, he will be able to make a certain amount of loaves
- Our Head TAs calculated that his number of loaves made is the amount of dough times his baking time
- Loaves sold equals one half of the square root of his baked loaves
Bread Makers (2/6)

- **BreadMakerConstants** class keeps track of important constants in our calculation

```java
public abstract class BreadMakerConstants {

    // Already sold 10 loaves
    public static final double START_LOAVES = 10;

    // Number of loaves sold to win the competition
    public static final double MAX_LOAVES = 200;

}
```
Bread Makers (3/6)

- **Peeta** keeps track of instance variable `loavesSold`

- `loavesSold` initialized in constructor to `START_LOAVES` defined in `BreadMakerConstants`

```java
import java.lang.Math;

class Peeta {
    private double loavesSold;

    public Peeta() {
        this.loavesSold = BreadMakerConstants.START_LOAVES;
    }
}
```
Peeta’s **bake** method changes his number of loaves sold depending on the amount of dough he has and the time he has to bake.
Bread Makers (5/6)

- First, `loavesMade` is computed
- Second, `anotherLoafSold` 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 total loaves sold

```java
import java.lang.Math;
public class Peeta {
    private double loavesSold;

    public Peeta() {
        this.loavesSold = BreadMakerConstants.START_LOAVES;
    }

    public void bake(double dough, double bakeTime) {
        double loavesMade = dough * bakeTime;
        double anotherLoafSold = (1/2) * Math.sqrt(loavesMade);
        this.loavesSold += anotherLoafSold;
    }
}
```
Bread Makers (6/6)

- Now fill in `sellBread()`
- Peeta will only bake & sell bread until he wins the competition
- 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 Peeta {
    private double loavesSold;

    public Peeta() {
        this.loavesSold = BreadMakerConstants.START_LOAVES;
    }

    public void bake(double dough, double bakeTime) {
        double loavesMade = dough * bakeTime;
        double anotherLoafSold = (1/2) * Math.sqrt(loavesMade);
        this.loavesSold += anotherLoafSold;
    }

    public void sellBread() {
        // decision-making logic that calls bake()!
    }
}
```
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
• British logician George Boole (1815-1864) wanted to improve on Aristotelian (formal) logic, e.g., modus ponens, rule of inference:
  o “All men are mortal, Socrates is a man, therefore…”

• boolean (named after Boole) is simplest Java base type
  o You’ve seen this in Pong!

• 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:
  
  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?
  
  o Assume these three examples are run in order
  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
class DogPark {
    // constructor elided

    public void compareReferences() {
        // Dog class defined elsewhere in code
        Dog d1 = new Dog();
        Dog d2 = new Dog();

        1 boolean foo = (d1 == d2);
        System.out.println(foo);
        d2 = d1;
        2 foo = (d1 == d2);
        System.out.println(foo);
        3 foo = !(d1 != d2);
        System.out.println(foo);
    }
}
```
Which of the following will print false?

```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);
    }
}
```

A. `System.out.println(sameStudent);`

B. `System.out.println(sameStudent);`

C. `System.out.println(student1Exists);`

Join Code: 504547
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

- Previous Statements
- Is condition true?
  - No: Execute rest of method
  - Yes: Execute if clause
**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
  
  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)

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

|    |    | 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 |
Which `if` clause statement will run if the game has started and the tools have been gathered? (The variables below are of type `boolean`)

A. `if(!gameStarted && !toolsGathered){...}
B. `if(!gameStarted && toolsGathered){...}
C. `if(gameStarted && !toolsGathered){...}
D. `if(gameStarted && tools Gathered){...}

Join Code: 504547
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)

```java
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

Previous Statements

Is condition true?

Yes → Execute if clause

No → Execute else clause

Execute rest of method
Can use **if-else** to fill in the `sellBread` method

If Peeta’s loaves sold are less than amount needed when method is called, he makes bread

Otherwise, he stops and wins the competition!

Does this code limit the final number of loaves sold to `MAX_LOAVES`?
Complex \textbf{if-else} Statements

- If $<\text{boolean expression 1}>$ is true, block 1 is executed and blocks 2 and 3 are skipped

- If $<\text{boolean expression 1}>$ is false and $<\text{boolean expression 2}>$ 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

if (cs15Student.hasBug()) {

    if (cs15Student.hasInitiative()) {
        cs15Student.debug();
    } else {
        cs15Student.giveUp();
    }
}

```
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
  
  o **&&**: if left argument of conditional evaluates to **false**, right argument not evaluated
  
  o **||**: 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
}

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);
// system output: 2
```

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

- To do something different for every possible value of an integer variable, have two options:
  - use a lot of `else-if`s:
    ```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:**

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

**Rules:**

- `<variable>` usually an `integer` – `char` and `enum` (discussed later) also possible
- `values` have to be mutually exclusive
- If `default` is not specified, Java compiler will not do anything for unspecified values
- `break` indicates the end of a `case` – skips to end of switch statement *(if you forget `break`, the code in next case will execute)*
switch Example (1/6)

• Let’s make a **ScarfCreator** class that produces different colored scarves for our players using a switch statement.

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

• **ScarfCreator** generates random values using **Math**.

• Based on random value, creates and returns a **Scarf** of a particular type.

```java
// imports elided - Math and Color
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        // more complicated logic...
    }
}
```

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.
To generate a random value, we use the 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 `myScarf` to `null`, and `switch` on the random value we’ve generated

```java
// imports elided – Math and Color
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {
            // code elided
        }
    }
}
```
switch Example (4/6)

- **Scarf** 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 `Scarf` and assign it to `myScarf`

- `break` breaks us out of `switch` statement
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {
            case 0: case 1:
                myScarf = new Scarf(Color.ORANGE);
                break;
            case 2: case 3: case 4:
                myScarf = new Scarf(Color.YELLOW);
                break;
        }
    }
}

• If our random value is 2, 3, or 4, we instantiate a yellow \texttt{Scarf} and assign it to \texttt{myScarf}

• \texttt{Color.YELLOW} is another constant of type \texttt{Color} – check out Javadocs for \texttt{javafx.scene.paint.Color}!
public class ScarfCreator{
    // constructor elided
    public Scarf generateScarf() {
        int randInt = (int) (Math.random() * 10);
        Scarf myScarf = null;
        switch (randInt) {
        case 0: case 1:
            myScarf = new Scarf(Color.ORANGE);
            break;
        case 2: case 3: case 4:
            myScarf = new Scarf(Color.YELLOW);
            break;
        // cases 5, 6, and 7 elided.
        // they are green, blue, red.
        default:
            myScarf = new Scarf(Color.BROWN);
            break;
        }
        return myScarf;
    }
    
    // Example (6/6)
    
    • We skipped over the cases for values of 5, 6, and 7; assume they create green, blue, and red Scarfs, respectively
    • Our default case (if random value is 8 or 9) creates a brown Scarf
    • Last, we return myScarf, which was initialized in this switch with a color depending on the value of randInt
TopHat Question

Which of the following `switch` statements is correct?

- In the constructor for `Weapon`, the parameter is a string.

A. 

```java
int rand = (int) (Math.random() * 10);
Weapon weapon = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        weapon = new Weapon("Axe");
    case 4: case 5: case 6: case 7:
        weapon = new Weapon("Poison");
    default:
        weapon = new Weapon("Knife");
    break;
}
```

B. 

```java
int rand = (int) (Math.random() * 10);
Weapon weapon = null;

switch (rand) {
    case 0: case 1: case 2: case 3:
        weapon = new Weapon("Axe");
    break;
    case 4: case 5: case 6: case 7:
        weapon = new Weapon("Poison");
    break;
    default:
        weapon = new Weapon("Knife");
    break;
}
```

C. 

```java
WeaponType type = type.random();
Weapon weapon = null;

switch (type) {
    case Axe:
        weapon = new Weapon("Axe");
    break;
    case Bali:
        weapon = new Weapon("Poison");
    break;
    default:
        weapon = new Weapon("Knife");
    break;
}
```
That’s It!

Important Concepts:

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

• FruitNinja (handout and help slides) released today
  - **Early handin:** 10/8 (+2 bonus points)
  - **On-time handin:** 10/10
  - **Late handin:** 10/12 (-8 for late handin, but 4 late days to use throughout semester)

• Debugging Hours start **Thursday, October 5**
  - More information on the course website

• Polymorphism section this week
  - email your section TAs mini-assignment on time

• SNC Deadline today at 5pm!! (Not CS15 enforced, University Policy)
SRC: Ethics and Labor Practices in Big Tech

CS15 Fall 2023
The Power of Big Tech

As of 2022…

• 50% of global online ad spending goes through Meta or Alphabet
• Amazon takes in more than 40% of online spending in the US
• In search, Google has more than a 60% share in the US
• Microsoft is a top-three vendor to 84% of businesses

Source: Harvard Business Review (2022)
How Big Tech Does Ethics: Internal Guidelines

- Internal advisory teams that create guidelines for responsible use of AI and other technologies
- Reports with established ethical principles for teams to follow

Program overview

We built our compliance and ethics program on three pillars: Prevention, Detection, and Remediation. We continually evolve our programs to meet these goals.
How Big Tech Does Ethics: Google’s “AI Applications We Will Not Pursue”

1. Technologies that cause or are likely to cause overall harm. Where there is a material risk of harm, we will proceed only where we believe that the benefits substantially outweigh the risks, and will incorporate appropriate safety constraints.

2. Weapons or other technologies whose principal purpose or implementation is to cause or directly facilitate injury to people.

3. Technologies that gather or use information for surveillance violating internationally accepted norms.

4. Technologies whose purpose contravenes widely accepted principles of international law and human rights.

As our experience in this space deepens, this list may evolve.
The Technology Facebook and Google Didn’t Dare Release

Engineers at the tech giants built tools years ago that could put a name to any face but, for once, Silicon Valley did not want to move fast and break things.

‘We decided to stop’

Source: NYT (2023)
Abuse of Power in Big Tech

Elon Musk Has Fired Twitter’s ‘Ethical AI’ Team
As part of a wave of layoffs, the new CEO disbanded a group working to make Twitter’s algorithms more transparent and fair.

Microsoft lays off team that taught employees how to make AI tools responsibly

Microsoft Agrees to Pay $20 Million Civil Penalty for Alleged Violations of Children’s Privacy Laws

Exclusive: Google cancels AI ethics board in response to outcry
The controversial panel lasted just a little over a week.
By Kelley Piper | Apr 4, 2023, 7:00pm EDT

European watchdog fines Meta $1.3 billion over privacy violations
May 22, 2023 - 1:38 PM ET
By Mary Yang, Eleanor Beardsley

TikTok Fined $370 Million for Mishandling Child Data

Working Conditions

Workers at Apple iPhone factory in China beaten in COVID protest

Gig Workers Behind AI Face ‘Unfair Working Conditions,’ Oxford Report Finds

Sources: CBS (2022), Forbes (2021), Time (2022)
Proposition 22

- Classifies Uber/Lyft drivers as independent contractors, not as employees
- Reduces benefits like insurance, saving companies money
- Gig companies spent >$200 million pushing for Proposition 22

Source: NYT (2023)
U.S. Accuses Amazon of Illegally Protecting Monopoly in Online Retail

The Federal Trade Commission and 17 states sued Amazon, saying its conduct in its online store and services to merchants illegally stifled competition.

Source: NYT (2023)
Next lecture… antitrust laws!

Designed to increase consumer welfare

Involves breaking up firms that get “too big”, or preventing mergers and acquisitions (M&A)

Highly debated subject