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.
# Quick Comparison: Inheritance and Interfaces

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Each <strong>subclass</strong> can only inherit from one <strong>superclass</strong></td>
<td>• You can implement as many interfaces as you want</td>
</tr>
<tr>
<td>• Useful for when classes have more similarities than differences</td>
<td>• Useful for when classes have more differences than similarities</td>
</tr>
<tr>
<td>• <strong>is-a</strong> relationship: classes that extend another class</td>
<td>• <strong>acts-as</strong> relationship: classes implementing an interface define its methods</td>
</tr>
<tr>
<td>o i.e. A <strong>Convertible</strong> is-a <strong>Car</strong></td>
<td>• Can only use methods declared in the interface</td>
</tr>
<tr>
<td>• Can define more methods to use</td>
<td></td>
</tr>
<tr>
<td>o i.e. <strong>Convertible</strong> putting its top down</td>
<td></td>
</tr>
</tbody>
</table>
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
  - will use polymorphism with inheritance and interfaces in Fruit Ninja
Lecture 7

Static Methods, Constants, and Making Decisions
Outline

- Review: numbers in Java and arithmetic operations
- Static methods and static variables
- Constants – values that never change
- Decision making: boolean algebra, if-else statements and the switch statement
- Method overloading – defining multiple methods of the same name
Review: Numbers in Java

- Integers represented with base type `int`
- Floating point numbers (decimals) represented with base type `float` (32 bits) or `double` (64 bits)
# Review: Basic Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>%</td>
<td>remainder</td>
</tr>
</tbody>
</table>
# Basic Arithmetic Operators: Shorthand

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Equivalent Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>add and reassign</td>
<td>a += 5;</td>
<td>a = a + 5;</td>
</tr>
<tr>
<td>-=</td>
<td>subtract and reassign</td>
<td>a -= 5;</td>
<td>a = a - 5;</td>
</tr>
<tr>
<td>*=</td>
<td>multiply and reassign</td>
<td>a *= 5;</td>
<td>a = a * 5;</td>
</tr>
<tr>
<td>/=</td>
<td>divide and reassign</td>
<td>a /= 5;</td>
<td>a = a / 5;</td>
</tr>
<tr>
<td>%=</td>
<td>take remainder and reassign</td>
<td>a %= 5;</td>
<td>a = a % 5;</td>
</tr>
</tbody>
</table>
# Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>negate</td>
<td>b = -b; // negates b</td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>b++; // equivalent to: b = b + 1;</td>
</tr>
<tr>
<td>--</td>
<td>decrement</td>
<td>b--; // equivalent to: b = b - 1;</td>
</tr>
</tbody>
</table>
Increment and Decrement Operators

- `++` and `--` can be applied before (prefix) or after (postfix) the operand
  - `i++` and `++i` will both increment variable `i`
  - `i++` assigns, then increments
  - `++i` increments, then assigns

**Postfix example:**

```java
int i = 10;
int j = i++; // j becomes 10, i becomes 11
```

**Prefix example:**

```java
int i = 10;
int j = ++i; // i becomes 11, j becomes 11
```
java.lang.Math

- Extremely useful “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
**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
  - `public static int abs(int a) {...}
- A **static method** belongs to a class, rather than an instance of the class
  - 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
Lecture Question

Object *myCountdownClock* is an instance of the *Countdown* class. Which is the correct way to call this static method:

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

A. `int minutesLeft = Instance.minutesToFive();`
B. `int minutesLeft = Countdown.minutesToFive(static);`
C. `int minutesLeft = CountdownInstance.minutesToFive(static);`
D. `int minutesLeft = Countdown.minutesToFive();`
E. `int minutesLeft = myCountdownClock.minutesToFive();`
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**

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

```java
public class Lover {
    private static int _lovers = 0;

    public Lover () {
        _lovers++;  
    }

    public static int getNumLovers() {
        return _lovers;
    }
}
```
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
  - literal numbers, except for 0 and 1, should rarely appear in your code
  - makes code readable, easy to maintain
- 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)
```
Lecture Question

Which of the following constants is defined correctly?

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

- Contestant Oliver Beckham decides to prepare for the island 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
• **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)

• *Oliver* keeps track of instance variable `_weight`

• `_weight` initialized in constructor to starting weight defined in `WorkoutConstants`
Workout (4/6)

• Oliver's `gainMuscle` method changes his weight according to the amount of time he works out.

```java
import java.lang.Math;
public class Oliver {
    private double _weight;
    public Oliver() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```
Workout (5/6)

• First, effort is computed
• Second, `muscleGained` is calculated according to the formula
• `Math.sqrt` is a static method from `java.lang.Math` that computes the square root of a value
• Increment the weight with the muscle gained

```java
import java.lang.Math;
public class Oliver {
    private double _weight;
    public Oliver() {
        _weight = WorkoutConstants.START_WEIGHT;
    }
    public void gainMuscle(double workoutTime) {
        double effort = workoutTime * _weight;
        double muscleGained = (1/10) * Math.sqrt(effort);
        _weight += muscleGained;
    }
}
```
Workout (6/6)

• Now fill in `calfWorkout` method

• Oliver 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 Oliver {
    private double _weight;

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

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

    public void calfWorkout() {
        //code to workout!
    }
}
```
**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:
  ```java
  boolean foo = true;
  boolean bar = false;
  ```

The terms foo, bar, etc. are often used as placeholder names in computer programming or computer-related documentation: 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 (not equal to 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);
    }
}
```
Lecture 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. `boolean sameStudent = (s1 == s2);
   System.out.println(sameStudent);`

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

C. `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 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^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|
Lecture Question

Which if clause statement will run if Oliver is not eliminated and there is a challenge tomorrow? (The variables below are of type boolean)

A. if(!oliverEliminated && !challengeTomorrow){…}
B. if(!oliverEliminated && challengeTomorrow){…}
C. if(oliverEliminated && !challengeTomorrow){…}
D. if(oliverEliminated && challengeTomorrow){…}
### if Statements: More Examples

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

- **Never do this (inefficient):**
  - if (boolean expression == true)
  - if (boolean expression == false)

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

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

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

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 `calfWorkout` method.

If Oliver’s weight is less than the maximum weight when the method is called, he gains muscle.

Otherwise, he stops and sails to the island!

Does this code limit the final calf weight to `MAX_WEIGHT`?

```java
import java.lang.Math;

public class Oliver {
    private double _weight;

    // constructor elided

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

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

if (cs15Student.hasProject()) {
    if (cs15Student.hasInitiative()) {
        cs15Student.workOnProject();
    }
    else {
        cs15Student.doChallenge();
    }
}
Lecture 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
  
  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
}
```

```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);
```

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

- If want to do something different for every possible value of a
  an integer variable, have two options:
  
  o 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
    }
    ```

  o 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)
Let’s make a `SwimsuitCreator` that produces different colored swimsuits for contestants using a switch statement.

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

`SwimsuitCreator` generates random values using `Math`.

Based on random value, creates and returns a swimsuit 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.
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 our swimsuit to null, and switch on the random value we’ve generated

```java
public class SwimsuitCreator{
    public Swimsuit generateSwimsuits() {
        int rand_int = (int) (Math.random() * 10);
        Swimsuit swimsuit = null;
        switch (rand_int) {
        }
    }
}
```
• *Swimsuit* 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 *Swimsuit* and assign it to *swimsuit*

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

```java
// imports elided - Math and Color
public class SwimsuitCreator{
    // constructor elided
    public Swimsuit generateSwimsuits() {
        int rand_int = (int)(Math.random() * 10);
        Swimsuit swimsuit = null;
        switch (rand_int) {
            case 0: case 1:
                swimsuit = new Swimsuit(Color.ORANGE);
                break;
```
**switch** Example (5/6)

- If our random value is 2, 3, or 4, we instantiate a red **Swimsuit** and assign it to **swimsuit**
- **Color.RED** is another constant of type **Color** – check out Javadocs for **javafx.scene.paint.Color**!

```java
public class SwimsuitCreator{
    // constructor elided
    public Swimsuit generateSwimsuits() {
        int rand_int = (int) (Math.random() * 10);
        Swimsuit swimsuit = null;
        switch (rand_int) {
            case 0: case 1:
                swimsuit = new Swimsuit(Color.ORANGE);
                break;
            case 2: case 3: case 4:
                swimsuit = new Swimsuit(Color.RED);
                break;
        }
    }
}
```
We skipped over the cases for values of 5, 6, and 7; assume they create green, blue, and yellow Swimsuits, respectively.

Our default case (if random value is 8 or 9) creates a brown Swimsuit.

Last, we return swimsuit, which was initialized in this switch with a color depending on the value of rand_int.
Lecture Question

Which of the following switch statements is correct?

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

A. int rand = (int) (Math.random() * 10);
   Island island = null;
   
   switch (rand) {
      case 0: case 1: case 2: case 3:
         island = new Island("Maui");
      
      case 4: case 5: case 6: case 7:
         island = new Island("Bali");
      
      default:
         island = new Island("Fiji");
      break;
   }

B. 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. 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;
   }
Method Overloading (1/3)

• Can define multiple methods of same name within a class, as long as method signatures are different, i.e., different parameters as in max

• Method signature: name, number, and types of parameters and their order

• Signature does NOT include return type

• Two methods with identical signatures but different return types (and different bodies) will yield a compiler error – why?

• Compiler (and you, the reader) can’t distinguish between two methods with the same signature and different return types when an instance calls those methods – method name and argument types passed in are exactly the same! So, signature is just name and parameter list

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

public class Math {
    // other code elided

    public static int max(int a, int b) {
        // return max of two ints
    }

    public static float max(float a, float b) {
        // return max of two floats
    }

    public static double max(double a, double b) {
        // return max of two doubles
    }
}
Lecture Question

Which of the following is true of a class that contains an overloaded method? The class has…

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
D. Two methods that are the same, except one contains an error
Method Overloading (2/3)

- Example: `java.lang.Math`
- `static` method `max` takes in two numbers and returns the greater of the two
- There are actually three `max` methods— one for `ints`, one for `floats`, one for `doubles`
- When you call an overloaded method, the compiler infers which method you mean based on types and number of arguments provided

```java
/* this is an approximation of what Math’s three max methods look like */
public class Math {
    // other code elided
    public static int max(int a, int b) {
        // return max of two ints
    }

    public static float max(float a, float b) {
        // return max of two floats
    }

    public static double max(double a, double b) {
        // return max of two doubles
    }
}
```
Method Overloading (3/3)

- Be careful not to confuse **overloading** and **overriding**!
  - **Overriding an inherited method in a subclass**: signatures and return types must be the same
  - **Overloading methods within the same class**: names are the same but the rest of the signatures (ie., the parameters) must be different so the compiler can differentiate; the return types may also differ (see `max`)

- Using same signatures and return types in different classes is OK because the compiler can differentiate by class/type of instance on which the method is called
Method Overloading: Constructors

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

  ```java
  String s = "CS15 Rocks!";
  System.out.println(s);
  ```

  The above code would print out `CS15 Rocks!` in the console.

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

      public Cook() {
          _dessert = "Birthday cake";
          _entree = "Sandwich";
      }

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

      public Cook(String dessert, String entree) {
          _dessert = dessert;
          _entree = entree;
      }
  }
  ```
Method Overloading: Example

- An overloaded method can call other overloaded methods

```java
public class FriendMakeOver {

    public FriendMakeOver(Wardrobe wardrobe) {
        Hat hat = wardrobe.getHat();
        this.wearAwesomeOutfit(hat);
    }

    public void wearAwesomeOutfit(Hat hat) {
        Tie tie = hat.getMatchingTie();
        this.wearAwesomeOutfit(hat, tie);
    }

    public void wearAwesomeOutfit(Hat hat, Tie tie) {
        // code to wearAwesomeOutfit elided
    }

    // other methods elided
}
```
That’s It!

Important Concepts:

• static methods and static variables
• Constants
• booleans
• Making decisions with if, if-else, switch
• Method overloading
• Method signatures: (name, number and types of parameters)
Announcements

• Leap Frog late hand-in **Friday 2/12** at 11:59PM EST
  o you must submit on CodePost in order for TAs to grade your project
• Polymorphism Section next Tues and Wed
  o email section TAs Mini-Assignment **before** your section begins
  o if you want to switch sections for the week, email your section TAs and the TAs for the section you want to switch into
• HTA Open Hours today 5-6pm EST
  o want to talk about how the course is going for you? Strategies for debugging or understanding material? Stop by!
• Mentorship survey going out in next few days
  o opt in – default mentor will be a section TA
An Object-Oriented Mindset to Leadership
By former CS15 HTA Dhruv

Encapsulation: I think of Google’s cross-functional teams as organizational objects. I work with them to define what properties they own (PRDs, UX mocks, eng designs) and what functions they have (build/test/ship a feature, collect user feedback, etc.).

Polymorphism: When a team is given a business problem to solve, I get different responses based on the kind of problem being tackled – sometimes, a finished product, and other times, data confirming why something shouldn’t be done.

Interfaces: I clarify expectations between teams using interfaces. I define the methods of engagement, but it’s up to the teams to implement them. The organization aligns around these interfaces/contracts.

Classes vs. Instances: Instead of tackling one instance of a problem at a time, I factor out commonalities to solve an entire class of problems at once. This prevents future instances of the problem from recurring.

OOP is a way to model the world around you. Java is merely the digital representation of that model. What you’re learning in CS15 is incredibly powerful, regardless of whether you choose to have a career in technology!
IT in the News

ft. Socially Responsible Computing!
Recall: IT in the News Topics of Focus

So far (will return!)

Social media & democracy

Social (Ir)responsibility in Tech Industry

Broader Impacts & Responsibilities of Technology

AI/ML, Algorithms, and Decision-Making

Up next (starting today!)
Surveillance State

- Surveillance state predates the computer
  - term originates from the 1970s-80s, when CCTV became widely used
  - today digital surveillance is more threatening than analog
    - cameras, GPS, smart appliances, facial recognition, data fusion from many (social media and e-commerce) sources
- China has extreme surveillance state which includes social credit scores
- In the U.S. we are surveilled more by private companies than the government
Surveillance Capitalism

- Describes system based on gathering data via surveillance and using it to predict/control behavior
  - term coined by retired HBS Professor Shoshana Zuboff in 2014

**Industrial capitalism** relied on **work** and **land** for the market dynamic

**Surveillance capitalism** translates **private experience** into commodities
- **We are the product!**
Surveillance Capitalism

“I describe surveillance capitalism as the unilateral claiming of private human experience as free raw material for translation into behavioral data. These data are then computed and packaged as prediction products and sold into behavioral futures markets.”  
– Shoshana Zuboff

- Behavior modification: Surveillance capitalists understood that the surest way to predict behavior is to intervene at its source and shape it
  - data collection via online tracking
  - expansion of surveillance, data collection to other fields (e.g. cars)
  - buying & selling of data between companies
  - behavior manipulation via addictive software design, targeted ads
  - information manipulation via “recommendations”
  - use of ML models & data in policing
  - use of ML models & data in job applicant screening
Tracking You, On and Off Social Media

- Massive amounts of data collected by companies on and off their platforms
  - other companies allow Facebook et al. to harvest data from their platforms
- What if you aren’t a social media user?
  - companies can still track you with set of techniques called **fingerprinting**
- Fingerprinting allows companies to pinpoint you using:
  - your hardware (e.g. Mac/PC)
  - browser type
  - browser behavior
  - and more (whether or not you share information online!)

Source: Facebook data for Prof. Norm Meyrowitz, CS1951V
THE TECHNOLOGY AND STRUCTURAL INEQUITY SERIES

Privacy and Surveillance

Wednesday, February 17, 2021 • 12:00 pm to 1:30 pm

The Technology and Structural Inequity series focuses on the impact of technology on marginalized communities. The series will bring together leading academics and activists whose work is influencing how we think about and how we fight against the harms that technology is causing. The speakers will examine how technology is being used to increase the surveillance and policing of marginalized communities and how many of these technologies are inherently biased and discriminatory.

This roundtable discussion on privacy and surveillance will feature:

Seny Kamara
Glencora Borradaile
Elissa M. Redmiles
Matt Mitchell

Find more info and register here.
“Algorithmically guided decisions are becoming increasingly prevalent and, if left unchecked, can amplify pre-existing societal biases. In this talk, I use modern computational tools to examine the equity of decision-making in two complex systems: automated speech recognition and online advertising. I demonstrate large racial disparities in the performance of popular commercial speech-to-text systems developed by Amazon, Apple, Google, IBM, and Microsoft, a pattern likely stemming from a lack of diversity in the data used to train the systems....In the second part of the talk,...I discuss how to formulate fair decisions considering budget-constrained trade-offs between English-speaking and Spanish-speaking SNAP applicants.”