Week 1 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Java features
- **`if`** and **`switch`** statements
- Loops
- Arrays
- Static methods

Questions?

Project 1

Method practice

- Write a method with the following signature that converts a **`String`** representation of an integer into an **`int`** value
- **`public static int parseInt(String value)`**

Overloading

- You're allowed to have two different methods with the same name, in the same class
- Doing so is called **overloading**
- However, the methods must either have a different number of parameters or different types of parameters so that the compiler can tell which one you're calling

Overloading example

- Two **`max()`** methods, one that finds the maximum of two values and another that finds the maximum of three

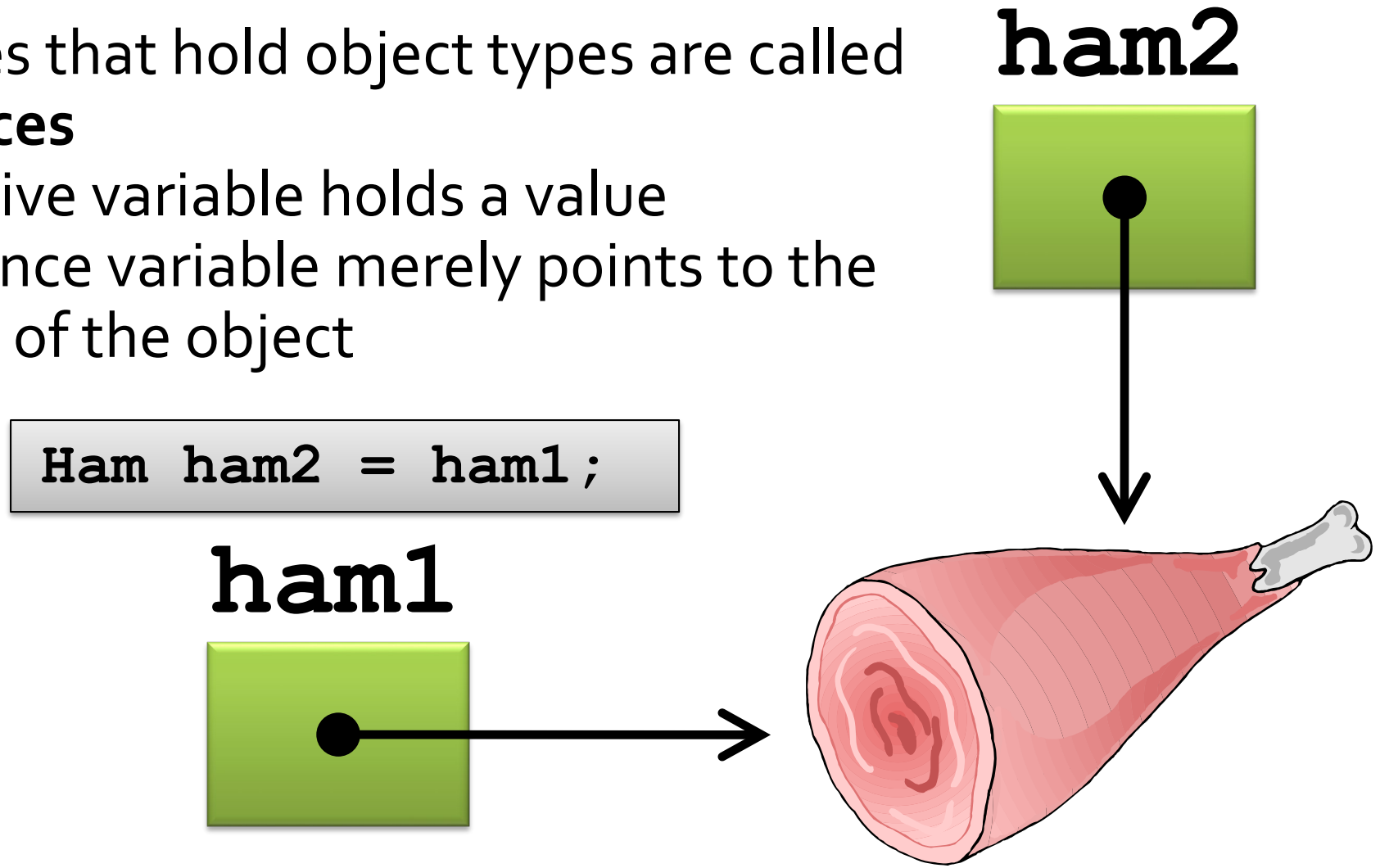
```
public static int max( int a, int b ) {
	if( a > b )
		return a;
	else
		return b;
}

public static int max( int a, int b, int c ) {
	if( a > b && a > c )
		return a;
	else if( b > a && b > c )
		return b;
	else
		return c;
}
```

Objects

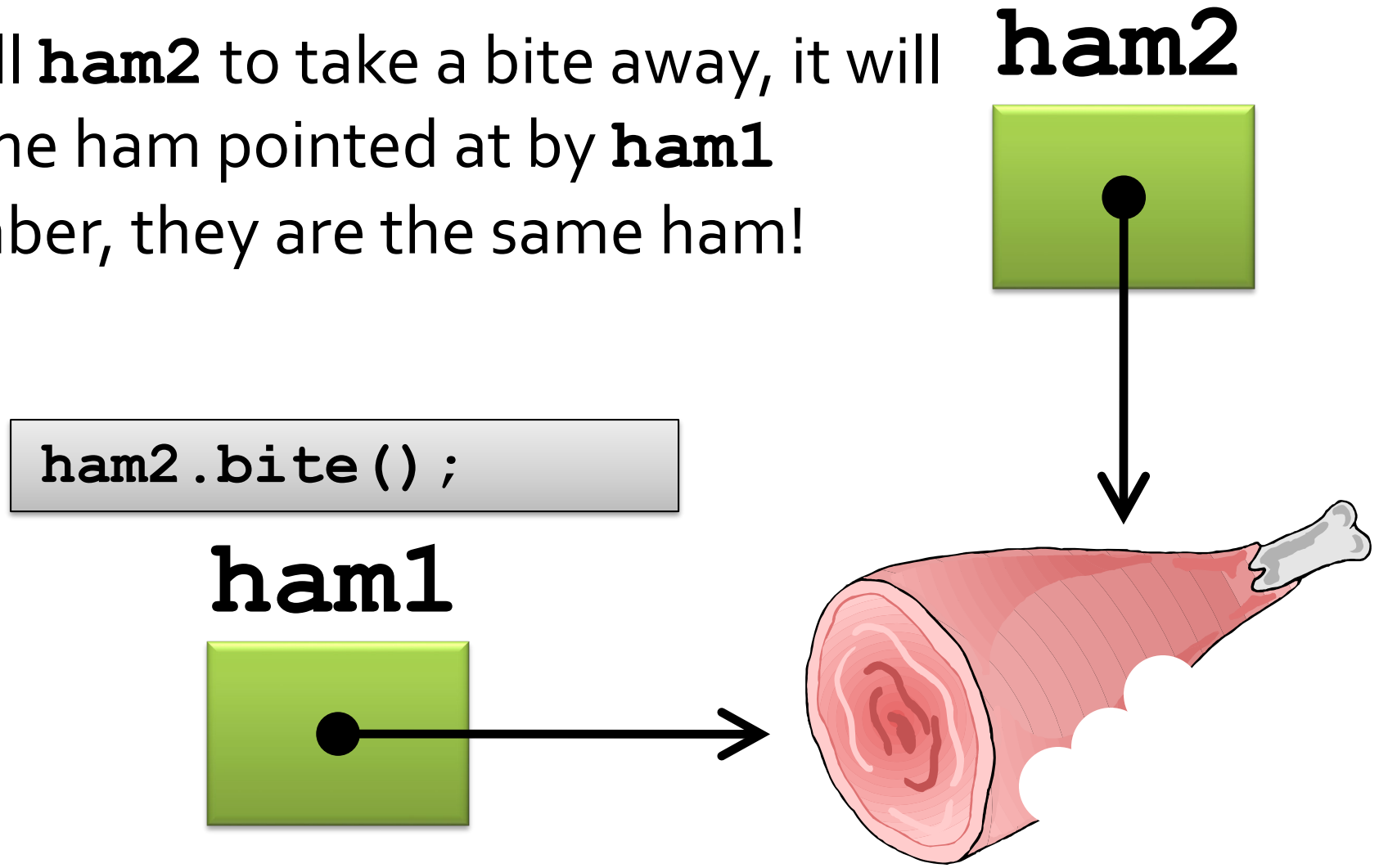
Reference types

- Variables that hold object types are called **references**
- A primitive variable holds a value
- A reference variable merely points to the location of the object



Reference variables

- If we tell **ham2** to take a bite away, it will affect the ham pointed at by **ham1**
- Remember, they are the same ham!



Compared to primitive variables

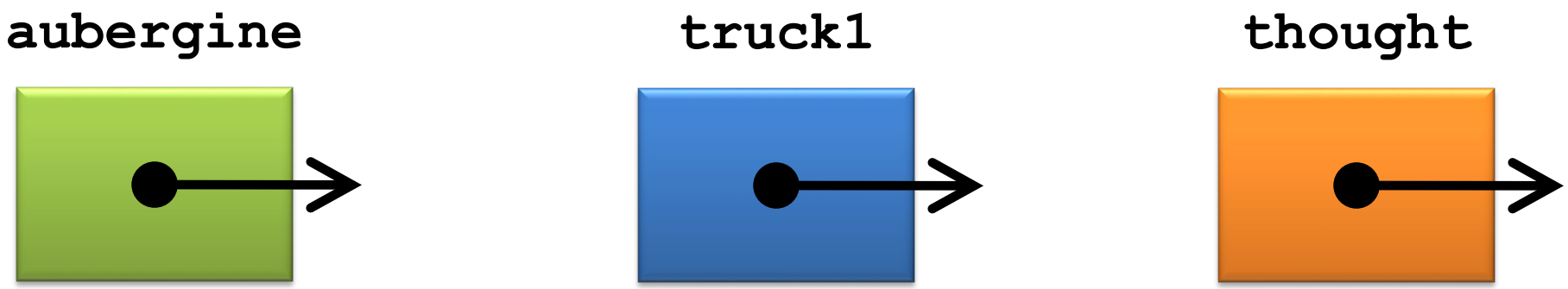
- Now consider **`int`** variables **`x`** and **`y`**, both with value **`37`**
- If we change **`x`**, it only affects **`x`**
- If we change **`y`**, it only affects **`y`**

```
int x = 37;
int y = x;
x++;
y--;
```

A reference is just an arrow

- If you declare a lot of references, you have not created any objects, just lots of arrows (unlike primitive types)

```
Eggplant aubergine;
DumpTruck truck1;
Idea thought;
```



Invoking the constructor

- To call a constructor, you use the **`new`** keyword with the name of the class followed by parentheses:

```
Ham ham1 = new Ham(); // Default constructor
```

- Perhaps there is a **`Ham`** constructor that lets you take a **`double`** that is the number of pounds that the ham weighs:

```
Ham ham2 = new Ham( 4.2 ); //weight constructor
```

Calling methods

- To call methods on objects
  - Type the name of the object
  - Put a dot
  - Type the method name, with the arguments in parentheses:

```
String s = new String("Help me!");
char c = s.charAt(3); //c gets 'p'
Ham h = new Ham(3.2);
h.bite(); // Takes bite out of ham
double weight = h.getWeight(); //Gets current ham weight
```

Equivalence confusion

```
String s1 = new String("identical");
String s2 = new String("identical");
if( s1 == s2 )
  System.out.println("Same!");
else
  System.out.println("Different!");
if( s1.equals( s2 ) )
  System.out.println("Same!");
else
  System.out.println("Different!");
```

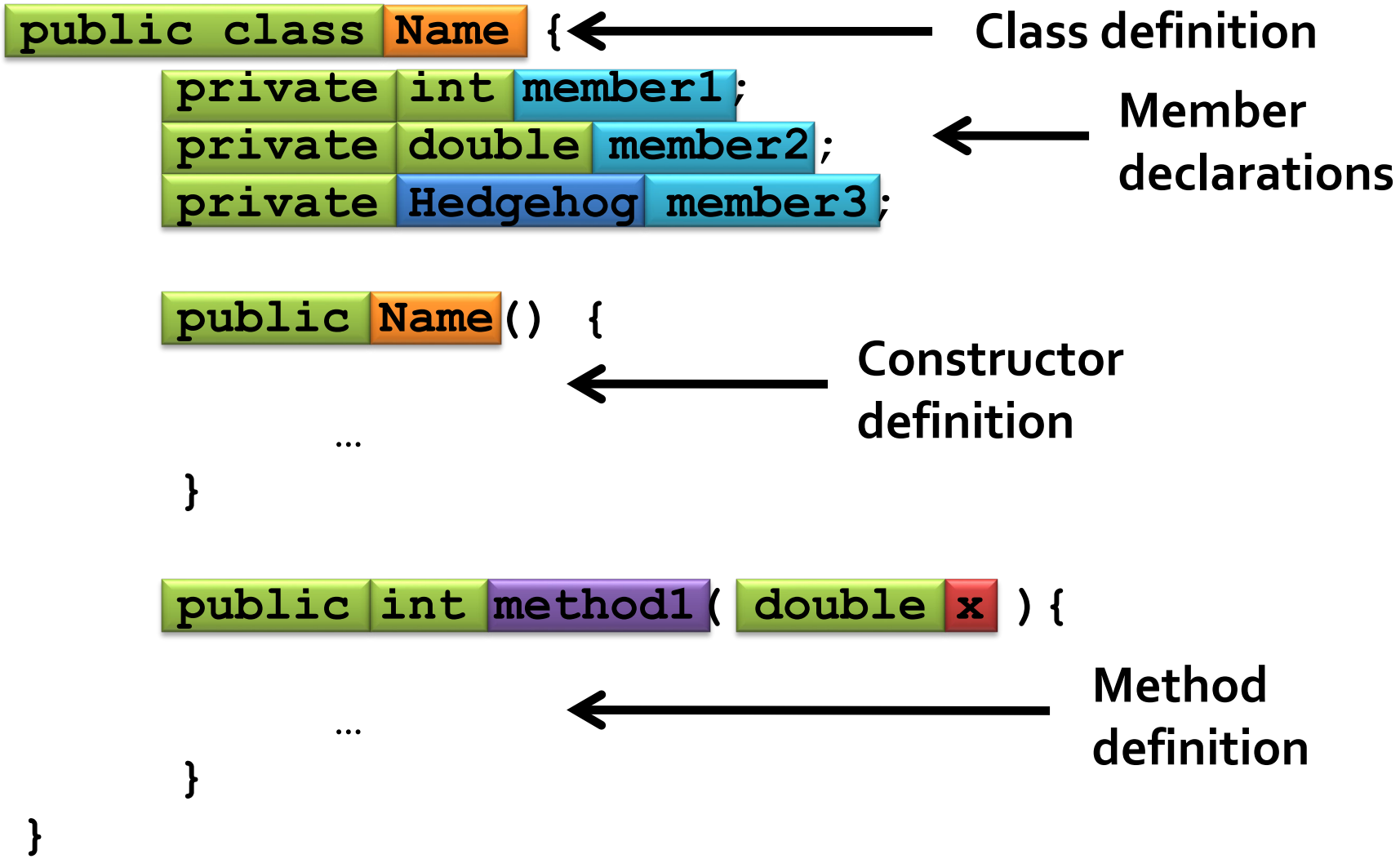
- In this example, the **==** operator will say they are different, but the **`equals()`** method will say that they are the same
- Every object has an **`equals()`** method
- Always call **`equals()`** to see if too objects are identical
- Only use **==** if you want to see if the two references are pointing at the exact same object

Classes

Templates for objects

- An object is the actual data that you can use in your code
- A **class** is a template whereby you can create objects of a certain kind
  - Class = Car
  - Object = Mitsubishi Lancer Evolution X
- Just like `int` is a type and `34` is an instance of that type
- A key difference is that you can define new classes
- Classes contain members and methods

Anatomy of a class definition



Members are data inside an object

- Members are the actual data inside an object
- They can be primitive types or other object types
- They are usually hidden (**`private`**) from the outside world

```
public class Point {
  private double x; // member variable
  private double y; // member variable
}
```

Data visibility

- **`private`** and **`public`** allow you to specify the scope or permissions of members and methods
- **`private`** means that only methods from the same class can access an item
- **`public`** means that any method can access the item
- **`protected`** means that classes in the package and child classes can access the data (but not someone outside of the inheritance hierarchy)
- No modifier means "package private" or default
  - Only code in the same package can access the item
  - More restrictive than **`public`** and **`protected`** but less restrictive than **`private`**

Methods are ways to interact with objects

- Methods allow you to do things
- Object methods usually allow you to manipulate the members
- They are usually visible (**`public`**) to the outside world
- Methods can be static or non-static
- Only non-static methods can interact with the members of an object

Constructors

- Constructors are a special kind of method
- They allow you to customize an object with particular attributes when it is created

```
public class Point {
  private double x; // member variable
  private double y; // member variable

  //constructor
  public Point( double newX, double newY ){
     x = newX;
     y = newY;
  }
}
```

Accessors

- Because members are usually **`private`**, it is common to use methods specifically just to find out what their values are
- A method that just returns the value of a member variable is called an **accessor**

```
public double getX() { //accessor for x
   return x;
}

public double getY() { //accessor for y
   return y;
}
```

Mutators

- Again, because members are usually **`private`**, it is common to use methods specifically just to change their values
- A method that just changes the value of a member variable is called a **mutator**

```
public void setX( double newX ) { //mutator for x
    x = newX;
}

public void setY( double newY ) { //mutator for y
    y = newY;
}
```

Class Variables

Static members

- Static members are stored with the class, not with the object

```
public class Item {
  private static int count = 0;   // one copy total
  private String name;            // one copy per object

  public Item( String s ) {
     name = s;
     ++count;                  // updates global counter
  }

  public String getName() { return name; }

  public static int getItemsInUniverse() {
     return count;
  }
}
```

Static rules

- Static members are also called **class variables**
- Static members can be accessed by either static methods or regular methods (unlike normal members which cannot be accessed by static methods)
- Static members can be either **`public`** or **`private`**

Members can be constant

- Sometimes a value will not change after an object has been created:
  - Example: A ball has a single color after it is created
- You can enforce the fact that the value will not change with the **`final`** keyword
- A member declared **`final`** can only be assigned a value once
- Afterwards, it will never change

Enums

Enums

- An enum is a special kind of class that has pre-defined constant objects
- These objects are intended to represent a fixed collection of named things:

```
public enum Day {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
  SATURDAY
}
```

- Individual days can be referenced like static variables: **`Day.MONDAY`** or **`Day.FRIDAY`**
- Since enum values are constants, it's convention to name them in ALL CAPS

Enums in `switch` statements

- Enums can be used in **`switch`** statements to make decisions

```
switch(day) {
case SUNDAY: System.out.println("Ice Cream"); break;
case MONDAY: System.out.println("Garfield"); break;
case TUESDAY: System.out.println("Taco"); break;
case WEDNESDAY: System.out.println("Addams"); break;
case THURSDAY: System.out.println("Throwback"); break;
case FRIDAY: System.out.println("I'm in Love"); break;
case SATURDAY: System.out.println("Surf NYC"); break;
}
```

- Note that only the value (**`SUNDAY`**) not the full name (**`Day.SUNDAY`**) is used
- This kind of behavior makes enums a useful way to record state information with a fixed number of values

Special enum features

- Though they aren't often useful, enums have some information baked into them
  - You can use the static **`values()`** method on the enum class to get an array containing all the enum values
  - You can call the **`ordinal()`** method on an enum object to get its zero-based numbering in the list
  - You can pass a **`String`** into the static **`valueOf()`** method to retrieve the enum object with a given name

Enum feature examples

- Sometimes it's useful to iterate over all the enum values
- Or get their number
- Or map a name to the enum value, but that will crash if you don't spell them right

```
Day[] days = Day.values();
for(Day day : days)
      System.out.println(day + " has index " + day.ordinal());

Day manic = Day.valueOf("MONDAY");
Day iDontHaveToRun = Day.valueOf("SUNDAY");
Day francais = Day.valueOf("DIMANCHE");  // Crashes!
```

Enums as full classes

- People usually use enums simply as lists of constant values
- However, enums are actually full classes whose objects can contain constant data and methods
- Note that the data inside can't be changed

```
public enum Planet {
    MERCURY(2440, 3.3E23, 5.79E7),
    VENUS(6052, 4.9E24, 1.08E8),
    EARTH(6371, 6.0E24, 1.50E8),
    MARS(3390, 6.4E23, 2.28E8),
    JUPITER(69911, 1.9E27, 7.78E8),
    SATURN(58232, 5.7E26, 1.42E9),
    URANUS(25362, 8.7E25, 2.87E9),
    NEPTUNE(24622, 1.0E26, 4.50E9);

    private int radius;       // km
    private double mass;      // kg
    private double distance; // km
```

Enum continued

- Here are the methods for the **`Planet`** enum from the previous slide

```
    private Planet(int radius, double mass, double distance) {
        this.radius = radius;
        this.mass = mass;
        this.distance = distance;
    }
    public int getRadius() {
        return radius;
    }
    public double getMass() {
        return mass;
    }
    public double getDistance() {
        return distance;
    }
}
```

Packages

Classes are files, packages are folders

- To organize classes, they are often inside of packages
- This approach allows to tell the difference between two different classes with the same name that are in different libraries:
  - **`java.util.List`** is the interface for list data structures
  - **`java.awt.List`** is a class that stores GUI lists
- Packages correspond to folders with the same names
- Most packages are inside of other packages
- The default package (no package) should not be used for professional programming
- Since we are transitioning in this class, look carefully at assignment requirements for packages

Package conventions

- By convention, class names (and interface, enum, and exception names) start with uppercase letters, such as **`ArrayList`**
- Packages should be written in lowercase letters, such as **`java.util`**
- Periods are used to separate the parent packages from their child packages
- A common convention is to use the reversed domain name of your company or institution to make your packages unique
  - We would be **`edu.otterbein`**

Imports

- When importing, you can import all of the classes in a package with an asterisk:
  - **`import java.util.*;`**
- However, the asterisk does not import the classes in any sub-packages
- If you want to import two classes that have the same name, one of them has to be called by its fully qualified name
  - In other words, you can't import **`java.util.*`** and **`java.awt.*`** because it wouldn't know which you mean when you say **`List`**
  - You could import **`java.util.*`** and refer in code to **`java.awt.List`** (which is ugly but doesn't happen too often)
- All classes in **`java.lang`** are automatically imported

Static imports

- Are you tired of how verbose it is to write **`System.out.println()`** or **`Math.sqrt()`**?
- A feature called **static imports** allows you to access static methods and members without specifying the class name
- If you put this at the top of your program:

```
import static java.lang.Math.*;
import static java.lang.System.*;
```

- You could write this:

```
out.println(sqrt(3));
```

- Instead of this:

```
System.out.println(Math.sqrt(3));
```

Upcoming

Next time…

- No class on Monday!
- Next Wednesday, we'll talk about interfaces

Reminders

- Read Chapter 10
- Pick your teammates for Project 1
- Afternoon office hours canceled today due to meetings
- Office hours canceled next Tuesday between 3 and 4 p.m. due to meetings