Week 1 - Wednesday

# COMP 2100

# Last time

- What did we talk about last time?
- Course overview
- Policies
- Schedule
- Taste of data structures

# Questions?

# Assignment 1

# Programming Model

# Programming model

- The book talks about stuff that you know pretty well as Java programmers
- I just want to talk about a few issues
  - Primitive types
  - Shortcut notations
  - Short circuit logic
  - **`break`** and **`continue`**
  - Libraries
  - Strings

# Primitive types

- Java has relatively strong typing
  - Understand why you're making a cast, and try not to make casts for no reason
- Remember that all the primitive numerical types in Java are **signed**
  - Strange things can happen

```
byte x = -128;
x *= -1;
System.out.println(x); // Output?
```

# Shortcut notations

- Java has various shortcuts that are *almost* the same as combinations of other operators: **+=**, **-=**, ***=**, **/=**, **%=**, **++**, **--** (and a few others)

```
int i = 0;
while (i < 10)
  i += 0.1; // Legal but crazy
```

- And know what you're doing with **++** (it means add one to the variable **and** store that value back into the variable):

```
int i = 0;
i = i++; // Legal but crazy
i = ++i; // Legal, crazy, different result
```

# Short-circuit logic

- Short-circuit logic means:
  - `true || expression` won't even evaluate `expression`
  - `false && expression` won't even evaluate `expression`
- You can force evaluation with non-short-circuit operators `|` and `&`:

```
if (alwaysTrue() || explode())
  whatever(); // explode() didn't run

if (alwaysTrue() | explode())
  whatever(); // explode() did run
```

# break and continue

- I don't like **break** and **continue** inside of loops
- There is usually a more readable, more elegant way to write the code
- But you should know that Java has a seldom-used labeled **break** feature that allows you to break out of multiple loops
- Say you're searching through a multi-dimensional array for a value:

```
search:
for (i = 0; i < arrayOfInts.length; i++) {
  for (j = 0; j < arrayOfInts[i].length; j++) {
     if (arrayOfInts[i][j] == searchfor) {
           foundIt = true;
           break search;
     }
  }
}
```

Example from: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

# Libraries

- One thing worth mentioning is that you get **`java.lang.*`** "for free," without importing anything:
  - **`String`**
  - **`Math`**
  - **`Object`**
  - **`Thread`**
  - **`System`**
  - Wrapper classes (**`Integer`**, **`Double`**, etc.)
- Any other classes outside of the current package must be imported to be used

# Strings

- The **`String`** type is immutable in Java
  - You can never change a **`String`**, but you can create a new **`String`**
  - The second line creates a new **`String`**:

```
String stuff = "Break it down ";
stuff += "until the break of dawn";
```

- This approach can be very inefficient:

```
String values = "";
for (int i = 0; i < 1000000; ++i)
  values += i;
```

- When a lot of concatenation is expected, use **`StringBuilder`**

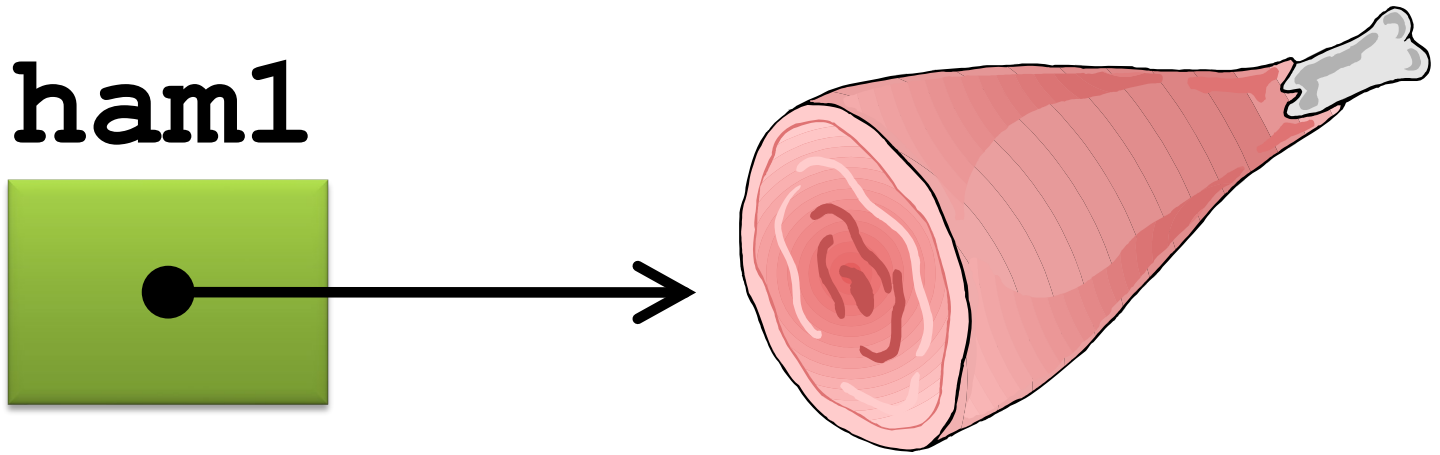
# References

# Pointers in Java

- Technically, Java doesn't have pointers
- Instead, every object in Java is referred to with a reference
- A reference is just an arrow that points at an object
  - A reference can point at nothing (**`null`**)
  - A primitive type can never be **`null`**

# How should you think about this?

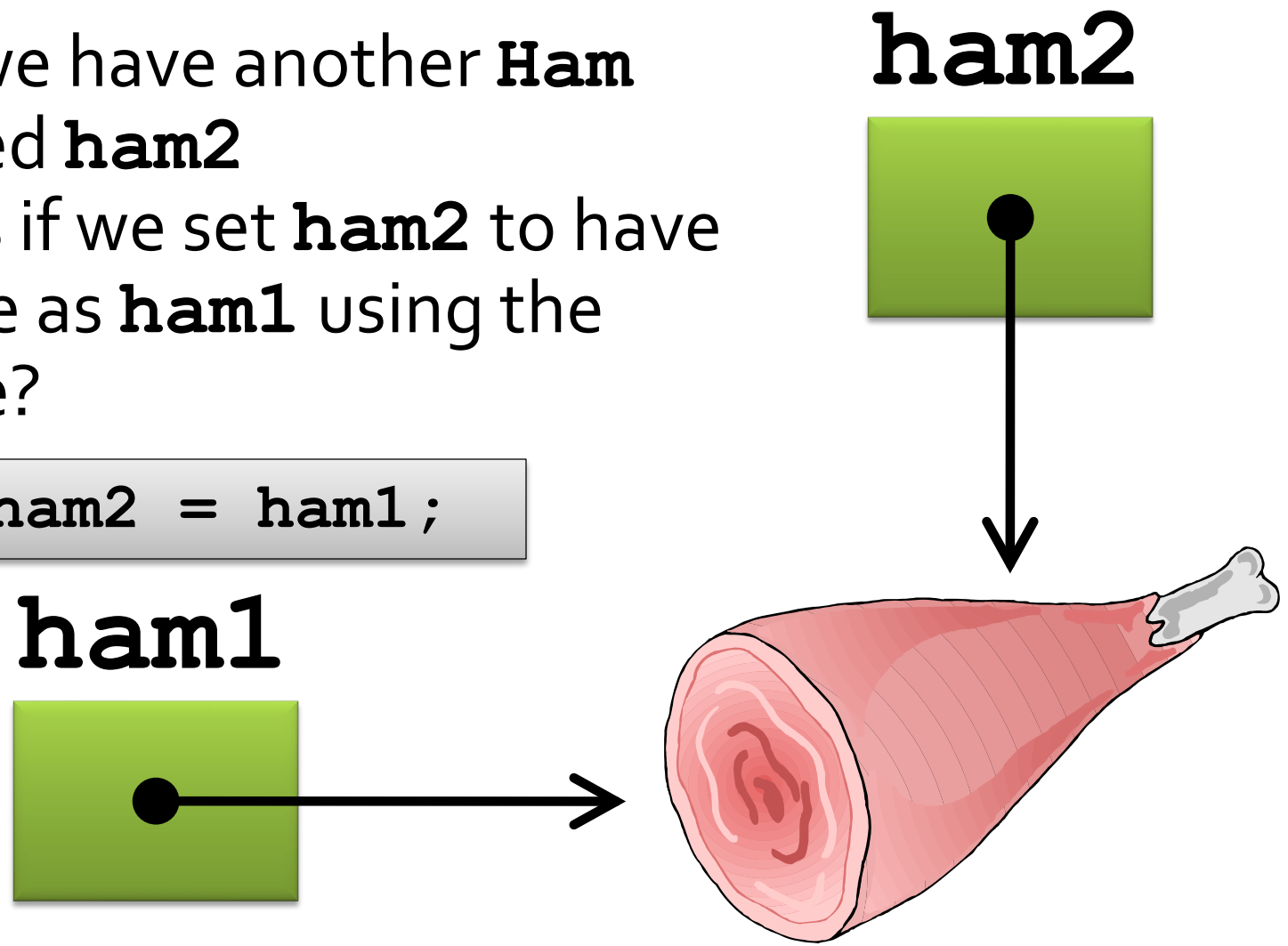
- Picture a ham…
- Imagine that this ham is actually a Java object
- You may want a reference of type **`Ham`** to point at this ham
- Let's call it **`ham1`**



# How many hams?

- Now, what if we have another **`Ham`** reference called **`ham2`**
- What happens if we set **`ham2`** to have the same value as **`ham1`** using the following code?

```
Ham ham2 = ham1;
```



# There is only one ham!

- When you assign an object reference to another reference, you only change the thing it points to
- This is different from primitive types
- When you do an assignment with primitive types, you actually get a copy

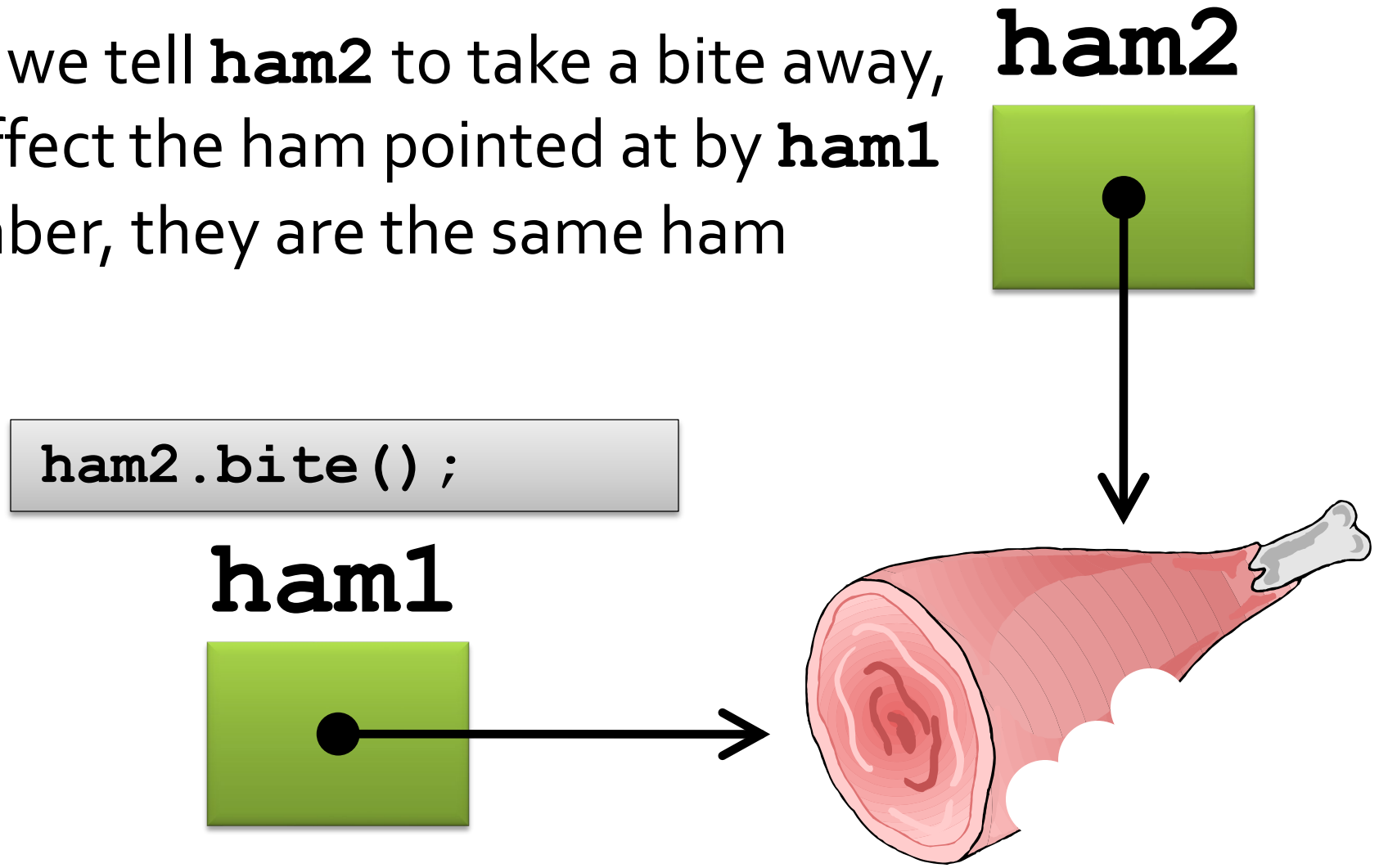
```
int x = 37;
int y = x;
```

# Reference vs. primitive variables

- Since reference variables are only **pointers** to real objects, an object can have more than one name
- These names are called **aliases**
- If the object is changed, it doesn't matter which reference was used to change it

# Ham solo

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



# Remember that primitives make copies

- We have **`int`** values **`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;
```

# The `clone()` method

- Sometimes you want to make a full copy of an object
- Every object has a **`clone()`** method that allows you to do this
  - **`clone()`** is intended to make a **deep copy** instead of a **shallow copy**
  - Ideally, all the objects inside of the object are cloned as well
  - There is no way to guarantee that **`clone()`** gives deep copies for arbitrary objects
- **`clone()`** works well for Java API objects
- You have to write your own if you want your objects to work right
  - Doing so can be tricky

# Static

# What is static?

- There are three ways that static can be used in Java
  - Static methods
  - Static members
  - Static inner classes
- "Staticness" is a confusing concept, but it boils down to missing a connection to a particular object

# Static methods

- A **static method** is connected to a class, not an object
- Thus, static methods cannot directly access non-static members
  - You also can't use **`this`** inside them
- Static methods **can** indirectly access members since they have the privileges to access private and protected data
  - You just have to pass them an object of the class they're in
- Static methods are slightly more efficient since they do not have dynamic dispatch
  - Thus, they cannot be overridden, only hidden

# Static methods

```
public class X {
  private int x;
  public static void print() {
     System.out.println("X");
     // x = 5;
     // previous line would not compile
     // if uncommented
  }
}

public class Y extends X {
  public static void print() {
     System.out.println("Y");
  }
}
```

# Static methods

```
X x = new X();
Y y = new Y();
X z;

x.print(); // prints X
y.print(); // prints Y

z = x;
z.print(); // prints X
z = y;
z.print(); // prints X
```

# Static members

- A **static member** is stored with the class, not with the object
- There is only ever one copy of a static member
- Static members are a kind of global variable
  - They should be used very rarely, for example, as a way to implement the singleton design pattern
- Static members can be accessed by static methods and regular methods

# Static members

```
public class Balloon {
   private String color;
   private int size;
   private static int totalBalloons = 0;

   public Balloon(String color, int size) {
      this.color = color;
      this.size = size;

      ++totalBalloons;
   }

   public String getColor() {
      return color;
   }

   public static int getBalloons() {
      return totalBalloons;
   }
}
```

# Inner Classes

# Static inner classes

- The simplest kind of inner class is a static inner class
- It's a class defined inside of another class purely for organizational purposes
- It cannot directly access the member variables or non-static methods of a particular outer class object

# Static inner class example

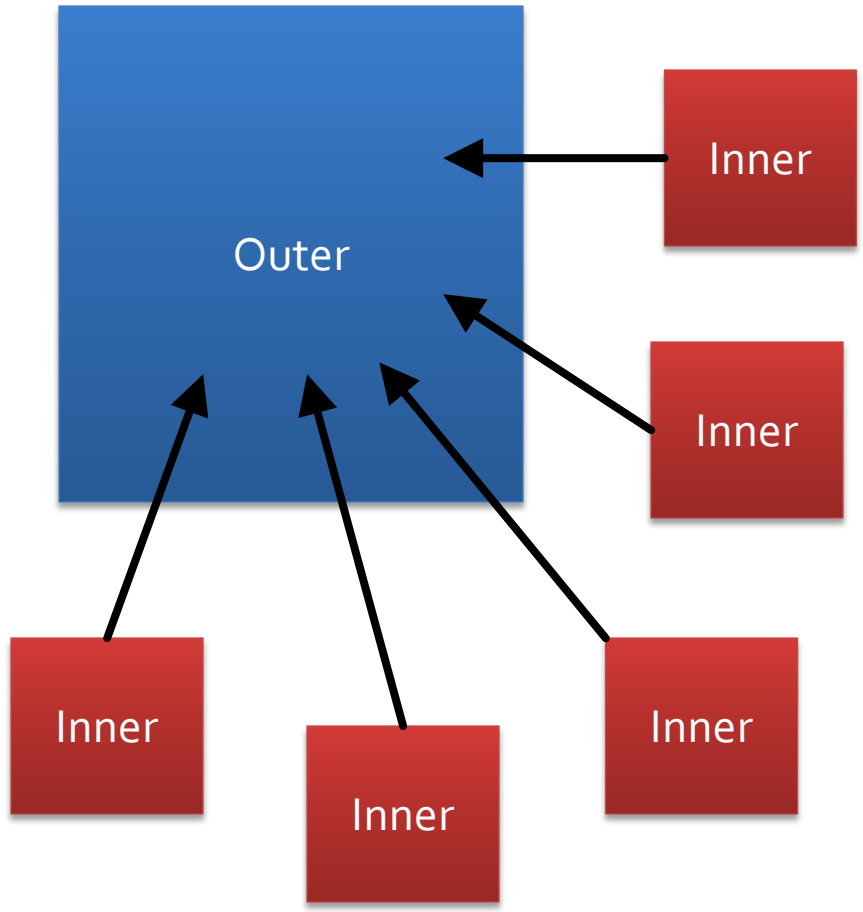
```
public class LinkedList {
  private Node head;

  private static class Node {
     public int value;
     public Node next;
  }
}
```

- In this example, the **`Node`** class is used like a struct from C or C++ to hold values

# Inner classes

- A non-static inner class is connected to a specific outer class *object*
- It can directly access the members and non-static methods of the outer object



# Inner class example

```
public class LinkedList {
  private Node head;
  private int size;

  private class Node {
     public int value;
     public Node next;

     public Node() {
          if (size > 100)
               System.out.println("Your list is long!");
     }
  }
}
```

# Creating inner classes

- If a static inner class is public, you can create it directly

```
Outer.StaticInner inner;
inner = new Outer.StaticInner();
```

- However, a non-static inner class requires an instance of the outer class to be created (with weird syntax)

```
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner();
```

- Inside the outer class, it is not necessary to give a reference to the outer class, since **`this`** is assumed

# When to use which

- Most of the time, a static inner class is fine
  - It isn't attached to a specific outer object
  - Most languages **only** have the equivalent of static inner classes
- However, if you want an inner class to automatically have access to a specific outer object, you might need a non-static inner class
  - For example, if a node needs to know the total number of nodes in a linked list
  - Iterators are another common example
  - **Beware of bugs:** a node created in one linked list can be moved to another linked list but will *still* be connected to the first one
- Use static inner classes unless there's a compelling reason not to

# Upcoming

# Next time…

- Exceptions
- OOP
- Interfaces
- Generics
- Java Collection Framework

# Reminders

- Come to lab tomorrow to keep working on Assignment 1 and start on Project 1
- Continue to read section 1.1
- Keeping brushing up on Java if you're rusty
- **Decide your teammates on Brightspace for Project 1 by this Friday!**