Introduction to Java

Michelle Brush
Senior Software Architect
Cerner Corporation

Class Structure

- The Why and What of Java
- Basic Logic
- **Object-Oriented Programming**
- When Things Go Wrong
- Common Java APIs
- JUnit
- Generics
- Putting It All Together I & II

Java sees the world as a bunch of objects that do things.

Main

Main is a special function.

```
class hello {
   public static void main(String[] args) {
      System.out.println("Hello World.");
   }
}
```

When you “run” a Java program, it looks for main.

Main

If you want to run your code,
you need to call it from main.

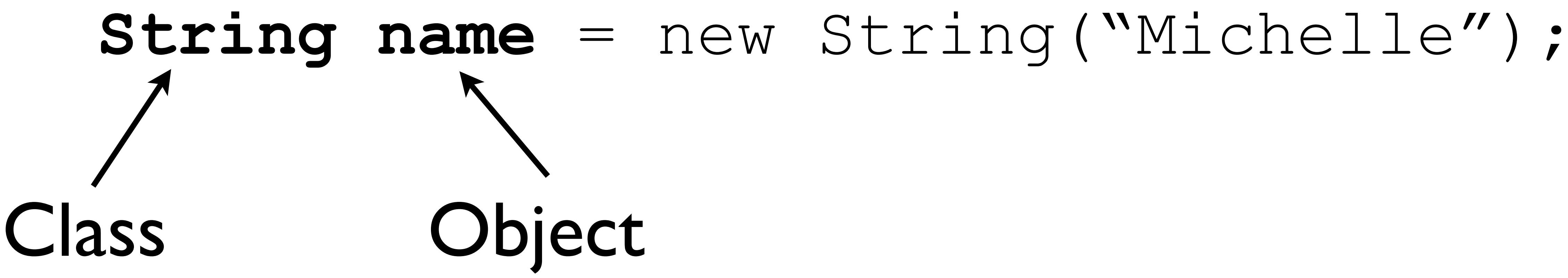
```
class hello {
  public static void main(String[] args) {
    System.out.println("Hello World.");
  }
}
```

(You can also call something in main that calls your code.)

Call Stack

The list of functions called is called the **Call Stack**.

Objects are created from classes.



```
String name = new String(“Michelle”);
```

Class

Object

Classes are templates or patterns for creating objects.

```
String name = new String(“Michelle”);
```

Objects are instances of the pattern or template.

```
String name = new String(“Michelle”);
```

Classes

- Classes can have data - called **member variables**.
- Classes can have functions - called **member methods**.

Classes

- member variables are **state**.
- member methods are **behavior**.

One way to think of a class is a means to group related **data** and **functions**.

Encapsulation

*The act of grouping **data** and related **functions** into a class and sometimes **hiding** the details of it.*

Anatomy of a Class

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

Anatomy of a Class

Class name

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

Anatomy of a Class

```
class Instructor {
   private String name;  Member variable

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

Anatomy of a Class

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

I hid it.

Anatomy of a Class

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

Constructor

Anatomy of a Class

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

A function that is called when I want to make one.

Anatomy of a Class

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public getName() {
      return name;
   }
}
```

Member method

Anatomy of a Class

Where's the word static?

```
class Instructor {
   private String name;

   public Instructor(String myName) {
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

Anatomy of a Class

static means something exists for the lifetime of the **class**.

We want our members to exist for the lifetime of the **object**.

How long does an object live?

An object lives as long as it remains scope.

Well, what's its scope?

The scope of an object is the code between when it was **created** and the end of the **block**.

What’s a block again?

```
{
    ...
}
```

Block:
An encapsulated section of code.

Using a Class

Instantiation

```
Instructor me = new Instructor(“Michelle”);

System.out.println(me.getName());
```

Using a Class

calling the constructor function

```
Instructor me = new Instructor(“Michelle”);

System.out.println(me.getName());
```

Object Scope

```
public void myFunction() {
    Instructor me = new Instructor(“Michelle”);
}
```

```
public void myFunction(int x) {
    if(x > 0) {
        Instructor me = new Instructor(“Michelle”);
    }
    System.out.println(me.getName());
}
```

Object Scope

```
public void myFunction() {
   Instructor me = new Instructor(“Michelle”);
} ← me doesn’t exist anymore.

public void myFunction(int x) {
   if(x > 0) {
      Instructor me = new Instructor(“Michelle”);
   } ← me doesn’t exist anymore.
   System.out.println(me.getName());
}
```

```
public void myFunction() {
   Instructor me = new Instructor("Michelle");
}
```

```
class Instructor {
   private String name;

   public Instructor(String myName)
      name = myName;
   }

   public String getName() {
      return name;
   }
}
```

When **me** doesn't exist, **name** doesn't exist either.

The **name** "Michelle" is part of **me**.

Object Scope

```
public void myFunction() {
    Instructor me = new Instructor(“Michelle”);
}


public void myFunction(int x) {
    if(x > 0) {
        Instructor me = new Instructor(“Michelle”);
    }
    System.out.println(me.getName());
}
```

This isn’t valid Java.

What about static?

Static means the object lives as long as the class lives.

This also means you only have **1 instance** of it no matter how many instances of the object you create.

Anatomy of a Class

```
class Instructor {
    private static String name;
    //...

}
```

Does it make sense for all instructors to share the same name?

Anatomy of a Class

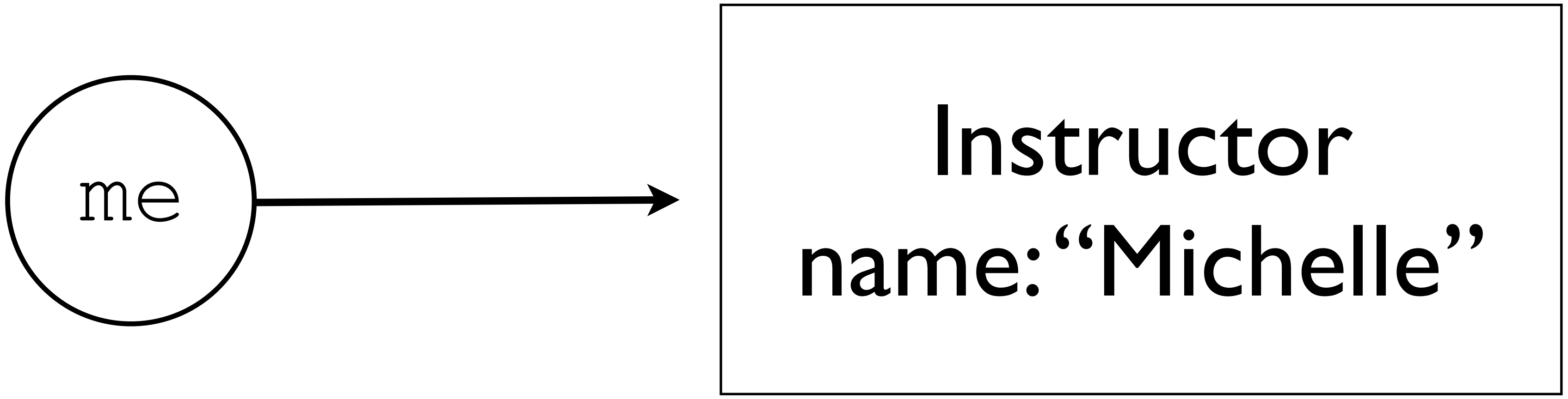
```
class Instructor {
   private ~~static~~ String name;
   //...



}
```

Probably not.

References

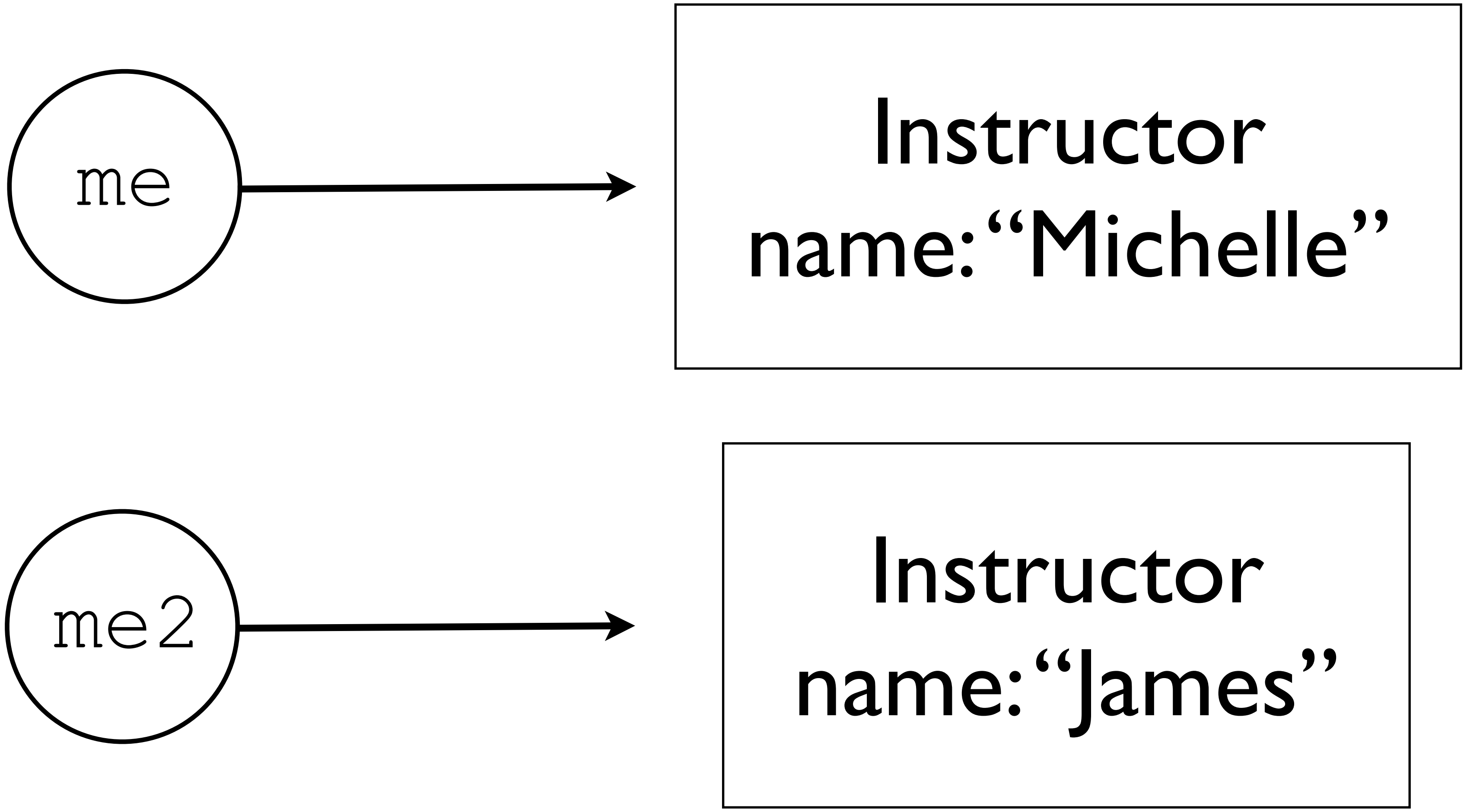
When you create a new object in Java, you are actually creating a reference to some memory.

```
Instructor me = new Instructor("Michelle");
```



References

```
Instructor me = new Instructor(“Michelle”);
Instructor me2 = new Instructor(“James”);
```



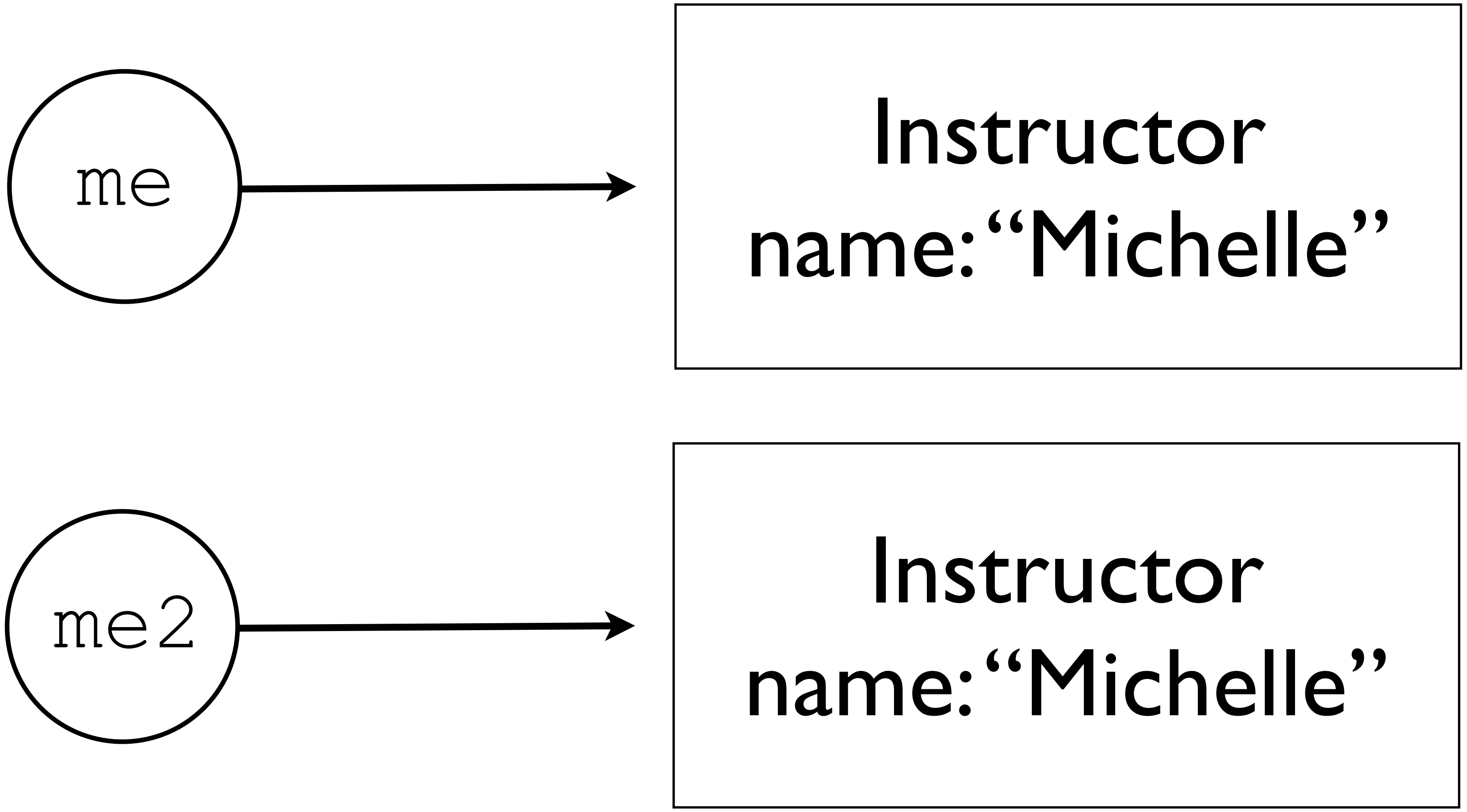
Important Java Rule #1

Never use == on an object.

You must use the equals method.

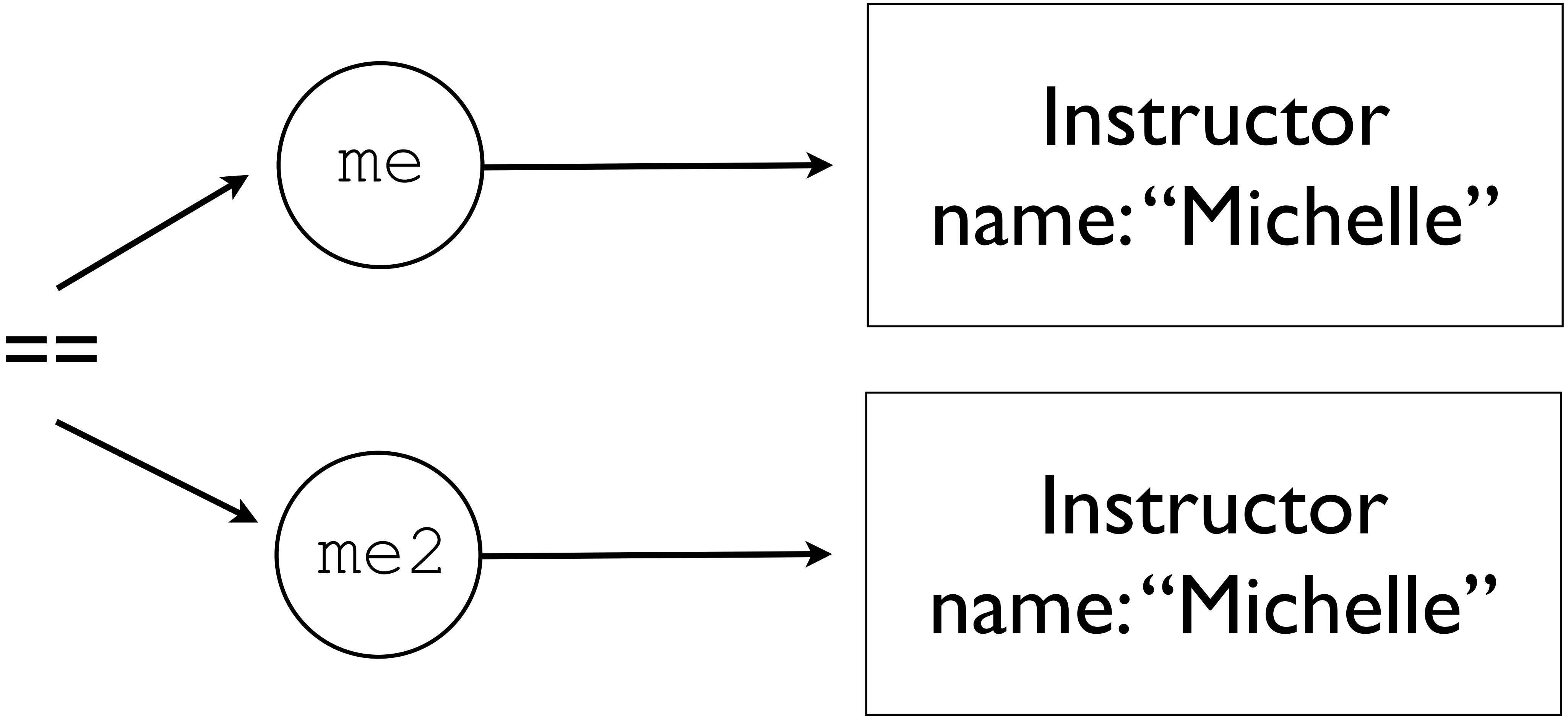
References

```
Instructor me = new Instructor(“Michelle”);
Instructor me2 = new Instructor(“Michelle”);
```



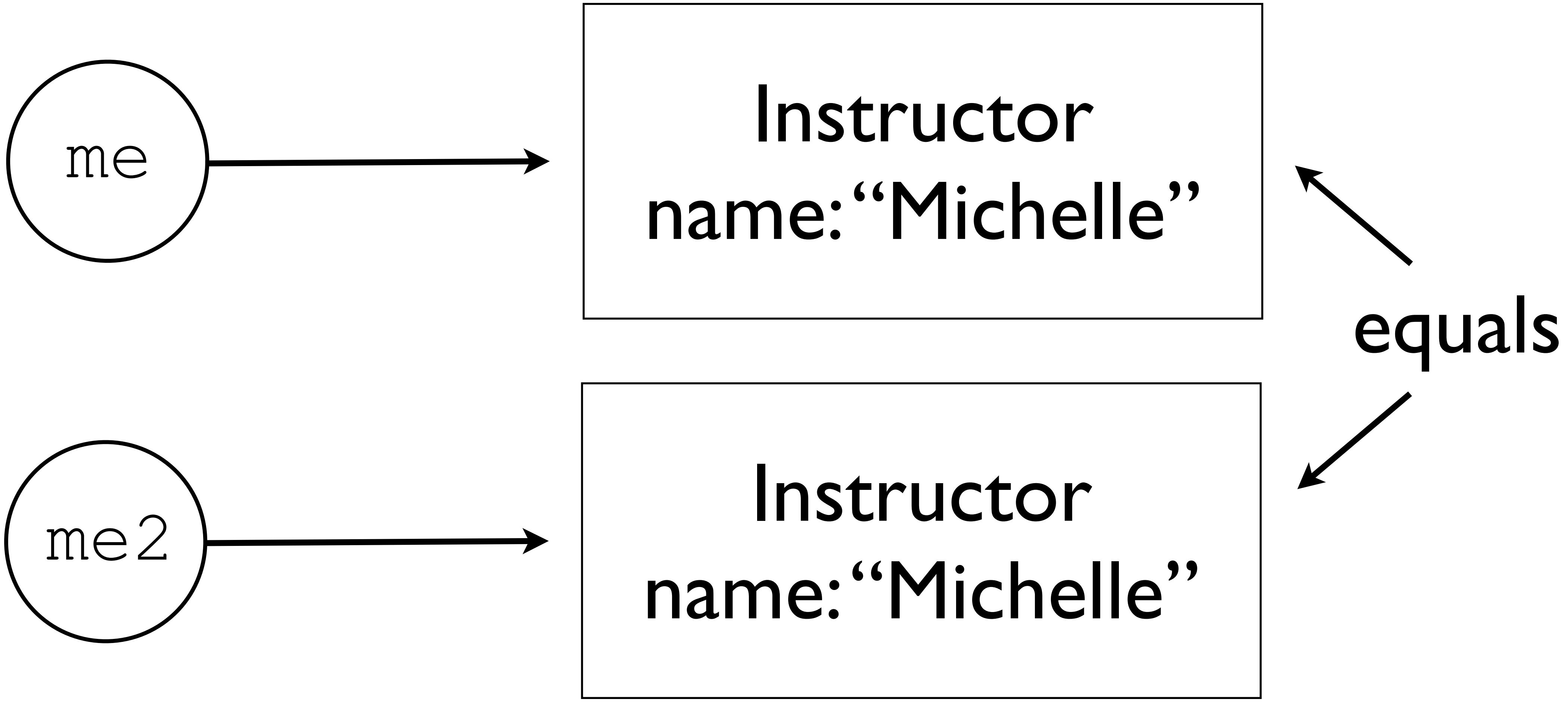
References

```
Instructor me = new Instructor(“Michelle”);
Instructor me2 = new Instructor(“Michelle”);
```



References

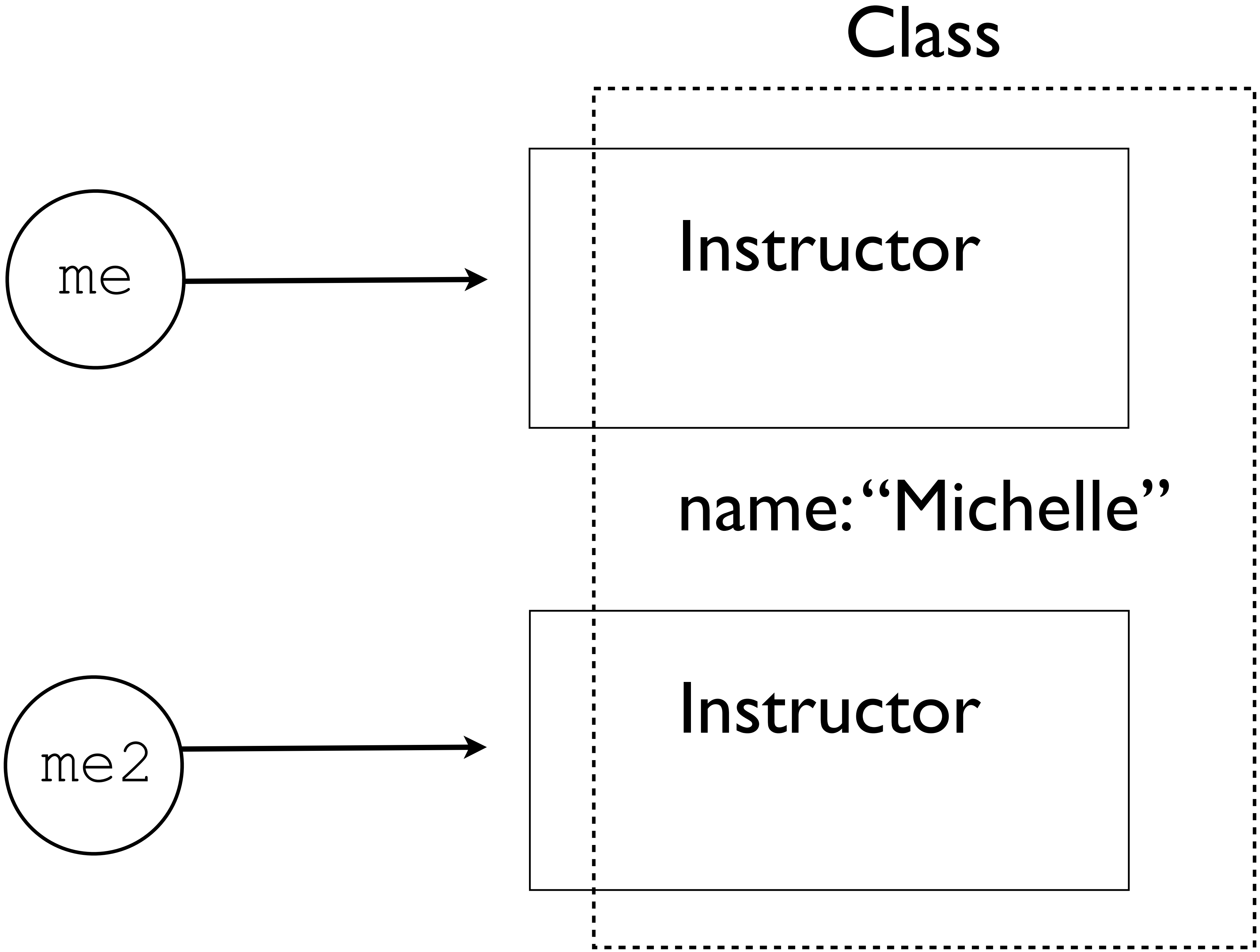
```
Instructor me = new Instructor(“Michelle”);
Instructor me2 = new Instructor(“Michelle”);
```



Using a Class

```
Instructor me = new Instructor(“Michelle”);
Instructor me2 = new Instructor(“Michelle”);

if(me == me2) {
   System.out.println(“This is bad.”);
}

if(me.equals(me2)) {
   System.out.println(“This is better.”);
}
```

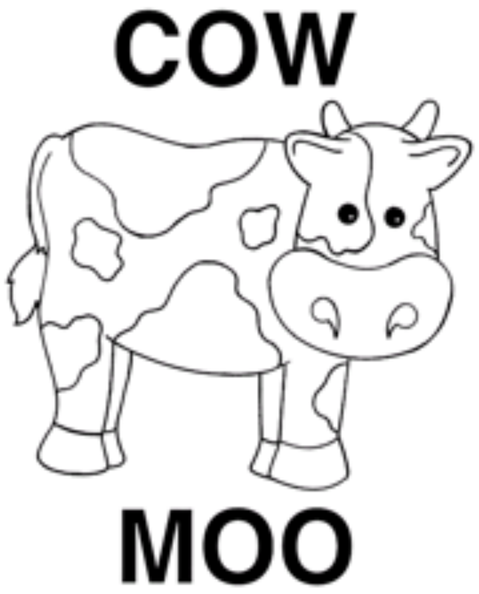
What if name was static?



Remember

- Classes are templates for objects.
- Classes have member variables and functions.
  - This is called encapsulation.
- Static makes members live at the class level.
  - If something is at the class level, there's only one copy of it shared by all instances.
- Objects in Java are actually references.

Objects in the World



Objects in the World

```
class Cat {
   public void meow() {
     System.out.println(“meow”);
   }
}
```

Objects in the World

```
class Cow {
   public void moo() {
     System.out.println(“moo”);
   }
}
```

Objects in the World

```
class Dog {
   public void bark() {
     System.out.println(“woof”);
   }
}
```

Objects in the World

```
class Animal {
   public void speak() {
     ???
   }
}
```

Objects in the World

```
interface Animal {
    void speak();
}
```

Objects in the World

```
class Cat implements Animal {
   public void speak() {
      System.out.println(“meow”);
   }
}
```

Objects in the World

```
class Cow implements Animal {
   public void speak() {
      System.out.println(“moo”);
   }
}
```

Interfaces

```
interface Animal {
    void speak();
}
```

A description of an object that has no behavior or data.

Interfaces

```
class Cat implements Animal {
   public void speak() {
      System.out.println(“meow”);
   }
}
```

Classes can implement interfaces.

Interfaces

```
class Cat implements Animal {
    public void speak() {
        System.out.println(“meow”);
    }
}
```

They have to provide behavior for all of the functions of the interface.

Interfaces

```
class Cat implements Animal {
    public void speak() {
        System.out.println(“meow”);
    }
}
```

This is called “implementing the interface.”

Why Interfaces?

```
class Cat implements Animal {
    public void speak() {
        System.out.println(“meow”);
    }
}
```

I can write code that knows about Animals without having to know about Cats.

Why Interfaces?

Like this toy.

The toy just knows it makes an Animal speak.

It doesn’t have to know what the Animal will say.

Why Interfaces?

```
class Toy {
   public void pull() {
      Animal animal = getAnimal();
      animal.speak();
   }
}
```

Classes vs. Interfaces

Classes have **behavior**.	Interfaces have no **behavior**.
Classes have **state**.	Interfaces have no **state**
Classes have **constructors**.	Interfaces have no **constructors**.
You can create a **new** instance of a class.	You can’t use **new** on an interface.

Classes vs. Interfaces

Bad

```
Animal animal = new Animal();
```

```
Animal animal = new Cow();
```

Good

Classes vs. Interfaces

Bad

~~Animal animal = new Animal();~~

Okay

```
Cow cow = new Cow();
```

```
Animal animal = new Cow();
```

Best Practice

Inheritance

- In Java, inheritance is the act of saying one object **is a** subtype of another.
- The subtype inherits all of the members of the base class or interface.
- The subtype must implement any unimplemented methods.

Cow is an Animal

Cat is an Animal

Objects in the World

```
class Animal {
   protected String sound;
   public void speak() {
      System.out.println(sound);
   }
}
```

Objects in the World

```
class Cow extends Animal {
   public Cow() {
      sound = “Moo”;
   }
}
```

Objects in the World

```
class Cat extends Animal {
   public Cat() {
      sound = “Meow”;
   }
}
```

Abstract Classes

```
abstract class Animal {
    public void speak() {
        System.out.println(getNoise());
    }
    abstract protected String getNoise();
}
```

Abstract Classes

```
abstract class Animal {
    public void speak() {
        System.out.println(getNoise());
    }
    abstract protected String getNoise();
}
```

I have to implement this
if I extend the class.

Abstract Classes

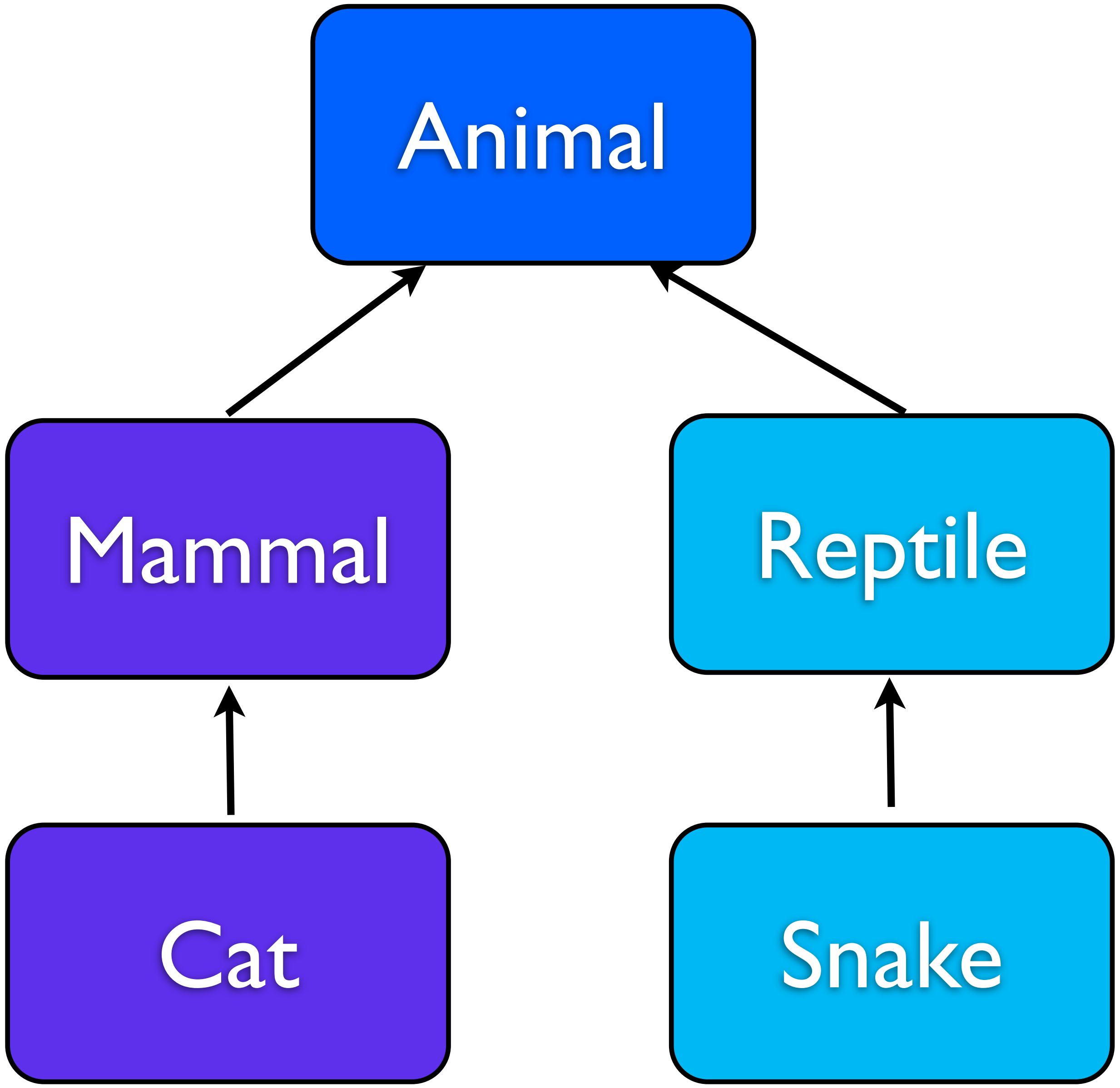
```
public class Cow extends Animal {
    protected String getNoise() {
        return "Moo";
    }
}
```

Abstract Classes

```
public class Cat extends Animal {
    protected String getNoise() {
        return "Meow";
    }
}
```

What can I inherit?

- Interfaces
- Abstract Classes
- Classes



Why do I inherit?

Generally, to add or refine object behavior.

Is there anything I can’t inherit?

Final

Anything marked as final cannot be overridden.

- Classes
- Methods
- Variables

Final Classes

```
public final class Cat extends Animal {
    protected String getNoise() {
        return "Meow";
    }
}
```

Final Classes

```
public class Tiger extends Cat {
    protected String getNoise() {
        return "Roar";
    }
}
```

Not valid Java!

If Cat is final, I can’t extend it.

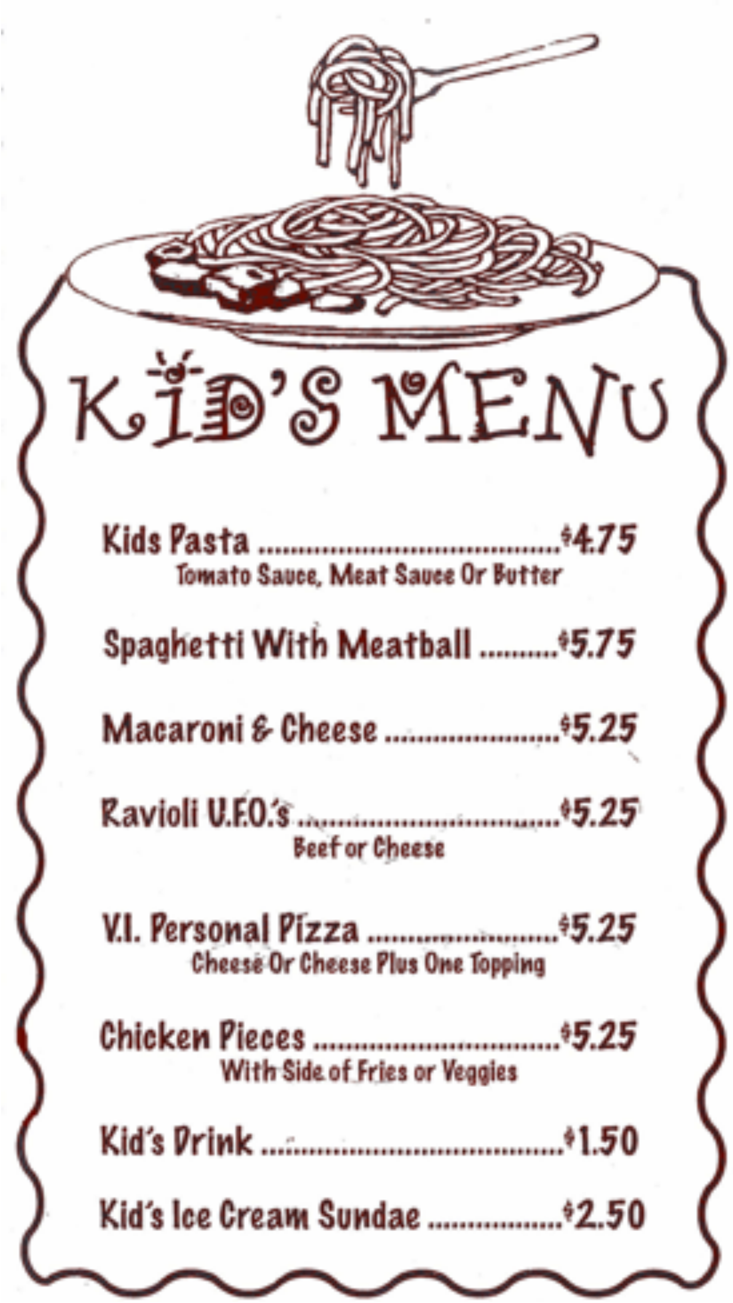
Why Final?

Safety
Protection

Activity

Let's build (and print) a menu system.

What classes should you have?



KID'S MENU

Kids Pasta Tomato Sauce, Meat Sauce Or Butter	$4.75
Spaghetti With Meatball	$5.75
Macaroni & Cheese	$5.25
Ravioli U.F.O.'s Beef or Cheese	$5.25
V.I. Personal Pizza Cheese Or Cheese Plus One Topping	$5.25
Chicken Pieces With Side of Fries or Veggies	$5.25
Kid's Drink	$1.50
Kid's Ice Cream Sundae	$2.50