Problem 1: Code duplication

- Re-writing instance variables and methods for each new animal
  - Goal: Just write common code once!
  - Solved through inheritance: the creation of the Animal superclass
Roadmap (Cont.)

- Problem 2: Hard to update code
  - Examples:
    - Adding a weight for all animals later
    - Updating the way a common method, `display()`, prints an Animal's report
  - Goal: Want to make code updates in one place!
    - Also solved through inheritance
Roadmap (Cont.)

- Problem 3: Hard to extend code
  - Still has an `ArrayList<Cows>` and an `ArrayList<Birds>`
  - Has an add method for each type of animal: `addCow()`, `addBird()`
  - `exciteAll()` and `summary()` have to loop through both lists separately
  - A lot of work to just introduce another animal type to our farm

- We will solve through: **polymorphism**
public class Farm {
    private ArrayList<Cow> cows;
    private ArrayList<Bird> birds;

    public Farm() {
        cows = new ArrayList<Cow>();
        birds = new ArrayList<Bird>();
    }

    public void addCow(Cow newCow) {
        cows.add(newCow);
    }

    public void addBird(Bird newBird) {
        birds.add(newBird);
    }

    public void exciteAll() {
        for (int i = 0; i < cows.size(); i++) {
            cows.get(i).speak();
        }

        for (int i = 0; i < birds.size(); i++) {
            birds.get(i).speak();
        }
    }

    public void summary() {
        System.out.println("-------------");
        System.out.println("Farm Summary");
        System.out.println("-------------");

        //display cows
        int numAnimals = 1;
        for (int i = 0; i < cows.size(); i++) {
            System.out.println("# " + numAnimals);
            cows.get(i).display();
            System.out.println();
            numAnimals++;
        }

        //display birds
        for (int i = 0; i < birds.size(); i++) {
            System.out.println("# " + numAnimals);
            birds.get(i).display();
            System.out.println();
            numAnimals++;
        }
    }
}
Current Implementation: Farm

- Farm still runs as expected.

```java
Farm myFarm = new Farm();
Bird prailine = new Bird("Prailine", "Yelp");
Cow daisy = new Cow("Daisy", "Mooo0000000000!");

myFarm.addBird(prailine);
myFarm.addCow(daisy);
myFarm.summary();
```

- Problem: Still lots of duplicated code in Farm!
  - Key insight: Exploit the Animal class abstraction we already made!
Outline

- **Subtyping**
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class

- **Polymorphic Methods**
  - Overriding Methods
  - The `super` Reference Revisited (in Methods)
  - Dynamic Dispatch

- **Conclusion**
Object Variables as Containers

- Our current (CS 161) understanding of object variables:
  - A Cow variable is a container that can hold *any* Cow object
  - Let's say that the Cow container's shape looks like:

- Example:

```java
Cow c1, c2;
```
Our current understanding of object variables:

- A Cow variable is a container that can hold any Cow object
- Let's say that the Cow container's shape looks like:

Example:

```java
Cow c1, c2;
c1 = new Cow();
```
Our current understanding of object variables:

- A Cow variable is a container that can hold *any* Cow object
- Let's say that the Cow container's shape looks like:

Example:

```java
Cow c1, c2;
c1 = new Cow();
c2 = new Cow("Tuffy", "Mewww");
```
Object Variables as Containers (Cont.)

- Our current understanding of object variables:
  - A Cow variable is a container that can hold *any* Cow object
  - Let's say that the Cow container's shape looks like:

- Example:

```java
Cow c1, c2;
c1 = new Cow();
c2 = new Cow("Tuffy", "Mewww");
c1 = c2;
```
Our current understanding of object variables:

- A Cow variable is a container that can hold any Cow object.
- Let's say that the Cow container's shape looks like:

Example:

```java
Cow c1, c2;
c1 = new Cow();
c2 = new Cow("Tuffy", "Mewww");
c1 = c2;
```
Object Variables as Containers (Cont.)

- Our current understanding of object variables:
  - A Cow variable is a container that can hold any Cow object.
  - Let's say that the Cow container's shape looks like:

**Example:**

```java
Cow c1, c2;
c1 = new Cow();
c2 = new Cow("Tuffy", "Mewww");
c1 = c2;
c2 = null;
```
Subtyping and Polymorphism

- **Subtypes**: an object of a subclass is also an object of its superclass
  - True in real life, true in OOP
  - e.g., all Cows are also Animals
  - e.g., all Birds are also Animals

- Important: *Object variables* may hold objects of their `declared type` or of any subtype.
Important: *Object variables* may hold objects of their *declared type* or of any subtype.
Important: *Object variables* may hold objects of their *declared type or* of any subtype.

What's the "shape" of the Animal object container?
Subtyping and Polymorphism

- Important: *Object variables* may hold objects of their declared type *or* of any subtype.

- What's the "shape" of the Animal object container?
Important: *Object variables* may hold objects of their *declared type* or of any subtype.

What's the "shape" of the Animal object container?
Important: *Object variables* may hold objects of their *declared type* or of any subtype.

**Key point:**
Polymorphism = "Many shapes"

A superclass can take on the "many shapes" of its *subclasses*

What's the "shape" of the Animal object container?

*(Remember the shape of this Animal container for remainder lecture.)*
Polymorphic Variables (Important!)

- Java supports subtypes via *Polymorphic Variables* (many shapes)
  - Think of these as those general shape-shifting *containers* that can hold any object of a subclass

```java
Bird b1 = new Bird(); // we always knew this works. Store the Bird object in a
// Bird variable
```

b1 (Bird variable stores a Bird object)
Object Substitution

- Java supports subtyping via *Polymorphic Variables*
  - Think of these as those general shape-shifting *containers* that can hold any object of a subclass

```java
Bird b1 = new Bird(); // we always knew this works. Store the Bird object in a // Bird variable

Animal b2 = new Bird(); // THIS WORKS TOO! A Bird object stored in an Animal // variable. This is called *substitution*.
```

b2 (Animal variable *can also* store a Bird object)

b1 (Bird variable stores a Bird object)
Does Reverse Substitution Work?

- Does the reverse work? ❌
  - Remember that inheritance is a one-way street.
  - Same "one-way-street" applies to subtyping

- Examples:
  ```java
  Bird daffy = new Animal();  // The Bird container is more specific than what you're trying to store inside of it!
  ```

- Intuition:
  - We "molded" a container to store Birds only. Too specific now!!!
    - It can not be re-molded to store something even more general.
Sanity Check: Polymorphic Variables

Given the class hierarchy, which of the following assignments are valid?

- `Student s1 = new DoctoralStudent();` ✔
- `GradStudent s2 = new PhDStudent();` ✔
- `PhDStudent s3 = new GradStudent();` ✗
- `UgradStudent s4;
s4 = new AssocStudent();` ✔
- `s4 = new BachelorStudent();` ✔
- `DentalStudent s5 = new MedStudent();` ✗
Trickier Still: Polymorphic Variables

Given the class hierarchy, which of the following assignments are valid?

Student s;
GradStudent g = new GradStudent();
s = g; ✓
g = s; X //won't compile, but shouldn't it?
g = (GradStudent) s; ✓ //need to type-cast

GradStudent g;
MasterStudent m;
DoctoralStudent d;
m = new MasterStudent();
g = m; ✓
d = (DoctoralStudent) m; X //won't compile, they don't share
   //subtype-supertype relationship!
d = (DoctoralStudent) g; X //compiles, but runtime error when
   trying to call DoctoralStudent methods!
Problem 3: Hard to extend code

- Still has an `ArrayList<Cow>` and an `ArrayList<Bird>`
- Has an add method for each type of animal: `addCow()`, `addBird()`
- `exciteAll()` and `summary()` have to loop through both lists separately
- Hard to add another animal

Insight: Animal is a **polymorphic type**!
Problem 3: Hard to extend code

- Farm now has a single ArrayList<Animal>
- Farm has a single method for all types of animals: addAnimal()
- exciteAll() and summary() are vastly simplified
- Introduce another animal type now requires no change to Farm class!

Insight: Animal is a polymorphic type!

[Let's update our Farm code]
Outline

- Subtyping
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class
- Polymorphic Methods
  - Overriding Methods
  - The `super` Reference Revisited (in Methods)
  - Dynamic Dispatch
- Conclusion
The instanceof Operator

- Consider the following conundrum:
  - Write a method `exciteCows()` that makes only the cows speak

```java
public void exciteCows() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Cow) {
            animals.get(i).speak();
        }
    }
}
```

- Need a way to determine whether an object belongs to a certain class!
  - Enter: instanceof operator
The instanceof Operator (Cont.)

- **Java Syntax:**
  
  ```java
  obj instanceof Classname
  ```

  - Results in a boolean
  - Determines if an object reference, `obj`, is of type `Classname`

- **Sample usage:**

  ```java
  Animal a = new Bird();
  a instanceof Bird
  > true (boolean)

  a instanceof Animal
  > true (boolean)

  a instanceof Student
  > false (boolean)

  a instanceof Object
  > true (boolean)
  ```
The instanceof Operator (Cont.)

Solution:

```java
public void exciteCows() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Cow) {
            animals.get(i).speak();
        }
    }
}
```

Trickier now: Make all Birds lay an egg

- What makes this tricky...? The following doesn't compile!

```java
public void layEggs() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Bird) {
            animals.get(i).layEgg();
        }
    }
}
```
Recall: Animals generally don't have a `layEgg()` method

- Only *specific* Animals do

```java
public void layEggs() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Bird) {
            animals.get(i).layEgg(); // Problem here! No such method in Animal!
        }
    }
}
```
Solution: "Down"-casting (Cont.)

- But **programmers** know inside this if-statement that it must be a Bird!
  - Need to inform Java of this! (Use a type-cast)

```java
public void layEggs() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Bird) {
            Bird b = (Bird) animals.get(i); // Down cast to Bird!
            b.layEgg(); // Now this works!
        }
    }
}
```
Solution: "Down"-casting (Cont.)

- Same idea, but a one-liner would be:

```java
public void layEggs() {
    for (int i = 0; i < animals.size(); i++) {
        if (animals.get(i) instanceof Bird) {
            ((Bird) animals.get(i)).layEgg(); // Down cast to Bird and call layEgg
        }
    }
}
```
Outline

- Subtyping
  - Polymorphic Variables
  - The Object Class
  - The instanceof Operator

- Polymorphic Methods
  - Overriding Methods
  - The super Reference Revisited (in Methods)
  - Dynamic Dispatch

- Conclusion
Important!! The Object Class in Java

- In truth all classes have a superclass*
  - Any class that doesn't inherit from superclass explicitly inherits from the Object superclass.
  - Java inserts the `extends Object` statement when you don't!

    ```java
    public class Student {
        ...
    }
    
    - Same as:
    
    public class Student extends Object {
        ...
    }
    * Except for the Object class
Why Object Class?

- Allows Java to provide a good deal of consistency across all objects
  - The following methods pertain to the Object class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object clone()</td>
<td>Constructs and returns a copy of this object</td>
</tr>
<tr>
<td>boolean equals(Object other)</td>
<td>Returns true if the given object is &quot;equal to&quot; this one</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns a distinct integer representation of this object. Useful for identifying an object within a Collection.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string representation of this object</td>
</tr>
<tr>
<td>(others omitted)</td>
<td></td>
</tr>
</tbody>
</table>

- Java can now make simplifying assumptions, like:
  - Every object has an equals(..) method
  - Every object has a toString(..) method
Also, Object are the ultimate polymorphic variable type:

• Example:
  - Not very useful though, as you always need to down-cast to call any specific method

```java
Object a = new Student(); // Works
Object b = new Bird();    // Works
...
```

• Another Example: This array list holds anything

```java
ArrayList<Object> list = new ArrayList<Object>();
```
Outline

- Subtyping
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class

- Polymorphic Methods
  - Overriding Methods
  - The super Reference Revisited (in Methods)
  - Dynamic Dispatch

- Conclusion
A Problem with `display()`

- Old class diagram:

  - `Farm`
    - `ArrayList<Cow> cows`
    - `ArrayList<Bird> birds`
    - `addCow(Cow c)`
    - `addBird(Bird b)`
    - `exciteAll()`
    - `summary()`

  - `Cow`
    - `String name`
    - `String quote`
    - `double gallonsMilked`
    - `display()`
    - `getName()`
    - `speak()`
    - `milk()`

  - `Bird`
    - `String name`
    - `String quote`
    - `int eggs`
    - `display()`
    - `getName()`
    - `speak()`
    - `layEgg()`

- Class diagram now:

  - `Farm`
    - `ArrayList<Animal> animals`
    - `addAnimal()`
    - `exciteAll()`
    - `summary()`

  - `Cow`
    - `double gallonsMilked`
    - `milk()`

  - `Bird`
    - `int eggs`
    - `layEgg()`

  - `Animal`
    - `String name`
    - `String quote`
    - `display()`
    - `getName()`
    - `speak()`

  - `Farm <<uses>> Animal`
A New Problem: The display() Method

Before:

```java
Farm myFarm = new Farm();
Bird prailine = new Bird("Prailine", "Yelp");
Cow daisy = new Cow("Daisy", "Mooo000000000!");
myFarm.addBird(prailine);
myFarm.addCow(daisy);
myFarm.summary();
```

After refactoring for polymorphism:

```java
Farm myFarm = new Farm();
Bird prailine = new Bird("Prailine", "Yelp");
Cow daisy = new Cow("Daisy", "Mooo000000000!");
myFarm.addAnimal(prailine);
myFarm.addAnimal(daisy);
myFarm.summary();
```

---

Farm Summary
---

# 1
Name: Daisy
Quote: Mooo000000000!
Milk produced: 0 gallons

# 2
Name: Prailine
Quote: Yelp
Eggs produced: 0 eggs

---

Farm Summary
---

# 1
Name: Daisy
Quote: Mooo000000000!

# 2
Name: Prailine
Quote: Yelp
A Problem with `display()`

- Old class diagram:

- Class diagram after refactoring for polymorphism:
A Problem with display()

- Old:

```java
public class Cow {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Milk produced: " + gallonsMilked + " gallons");
    }
}

public class Bird {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```

- After refactoring for polymorphism:

```java
public class Animal {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
    }
}
```
Maybe We Were Wrong?

- Maybe we were too aggressive in extracting `display()` into `Animal` superclass.
  - Remove the Cow and Bird specific print-outs, since they don't apply to every animal.
  - Solution(?): Undo and add `display()` back into subclasses?
Solution 1?: Undo the Extraction of display()

Would the following work?

- Put `display()` back inside the specific animal classes

```java
public class Farm {
    private ArrayList<Animal> animals;

    public void summary() {
        System.out.println("-------------");
        System.out.println("Farm Summary");
        System.out.println("-------------");

        for (int i = 0; i < animals.size(); i++) {
            System.out.println("# " + (i+1));
            animals.get(i).display();
            System.out.println();
        }
    }
}
```

Ans: Animal would no longer have a `display()` method

(Just like we saw in Lab)
But couldn't we just down-cast like we did in Lab?

- Could, but not ideal:

```java
public class Farm {
    private ArrayList<Animal> animals;

    public void summary() {
        System.out.println("-------------");
        System.out.println("Farm Summary");
        System.out.println("-------------");

        for (int i = 0; i < animals.size(); i++) {
            System.out.println("# " + (i+1));

            // need to add another if-statement for every new Animal type.. defeats purpose!
            if (animals.get(i) instanceof Cow) {
                ((Cow) animals.get(i)).display();
            }
            else if (animals.get(i) instanceof Bird) {
                ((Bird) animals.get(i)).display();
            }
            System.out.println();
        }
    }
}
```
Solution 2?

- Okay, we **ought to** leave `display()` in `Animal` superclass.
  - Note it again does the common functionality of printing name and quote

- Solution 2: In subclasses, **override** superclass' `display()` method
Solution 2?

- Implementation of Solution 2:

```java
public class Animal {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
    }
}

public class Cow extends Animal {
    @Override
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Milk produced: " + gallonsMilked + " gallons");
    }
}

public class Bird extends Animal {
    @Override
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```
Solution 2?

Implementation of Solution 2:

```java
public class Animal {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
    }
}

public class Cow extends Animal {
    @Override
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Milk produced: " + gallonsMilked + " gallons");
    }
}

public class Bird extends Animal {
    @Override
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```

Code duplication makes method hard to maintain in the future!
Important: The `super` keyword references the superclass.

```java
public class Animal {
    public void display() {
        System.out.println("Name: "+ name);
        System.out.println("Quote: "+ quote);
    }
}

public class Cow extends Animal {
    @Override
    public void display() {
        super.display(); //reuse Animal's display()!
        System.out.println("Milk produced: "+ gallonsMilked + " gallons");
    }
}

public class Bird extends Animal {
    @Override
    public void display() {
        super.display(); //reuse Animal's display()!
        System.out.println("Eggs produced: "+ eggs + " eggs");
    }
}
```
The @Override Directive

- When you annotate a method with `@Override`:
  - You inform Java that this method has the same signature as one of the methods in superclass
  - You also inform other programmers

- Java will hold you to it!
  - (Won't compile if signatures don't match)

- `@Override` is entirely optional, but strongly recommended
  - Required in this course
Outline

- Subtyping
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class
- Polymorphic Methods
  - Overriding Methods
  - The `super` Reference Revisited (in Methods)
    - `Dynamic Dispatch`
- Conclusion
Focus on `display()` method.

- Must be available in `Animal (to compile)`
  - But it's overridden in its subclasses.

- What version gets executed in `summary()`?

```java
public class Farm {
    private ArrayList<Animal> animals;

    public void summary() {
        System.out.println("-------------");
        System.out.println("Farm Summary");
        System.out.println("-------------");

        for (int i = 0; i < animals.size(); i++) {
            System.out.println("# " + (i+1));
            animals.get(i).display(); // ??
            System.out.println();
        }
    }
}
```

Ans: Depends on what's being stored in the variable.

The version belonging to the **most specific** class is always called!
Example of Dynamic Method Dispatch

```java
Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore!"));
my_farm.addAnimal(new Animal());
my_farm.summary(); // what gets printed??
```
Example of Dynamic Method Dispatch

Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore!"));
my_farm.addAnimal(new Animal());
my_farm.summary();

-------------
Farm Summary
-------------
# 1
Name: Tuffy
Quote: Mew
Milk produced: 0.0 gallons
Example of Dynamic Method Dispatch

Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore!"));
my_farm.addAnimal(new Animal());
my_farm.summary();

------------
Farm Summary  
------------

# 1
Name: Tuffy
Quote: Mew
Milk produced: 0.0 gallons

# 2
Name: Bessie
Quote: Moo
Milk produced: 0.0 gallons

---

Farm
- ArrayList<Animal> animals
  + addAnimal()
  + exciteAll()
  + summary()

Animal
- String name
- String quote
  + display()
  + getName()
  + speak()

Cow
- double gallonsMilked
  + display()
  + milk()

Bird
- int eggs
  + display()
  + layEgg()
Example of Dynamic Method Dispatch

```java
Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore!"));
my_farm.addAnimal(new Animal());
my_farm.summary();
```

---

Farm Summary
---

# 1
Name: Tuffy
Quote: Mew
Milk produced: 0.0 gallons

# 2
Name: Bessie
Quote: Moo
Milk produced: 0.0 gallons

# 3
Name: Raven
Quote: Nevermore!
Eggs produced: 0 egg(s)
Example of Dynamic Method Dispatch

Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore");
my_farm.addAnimal(new Animal());
my_farm.summary();
Let's Upcast to an Animal. summary() Now?

```java
Farm my_farm = new Farm();
my_farm.addAnimal(new Cow("Tuffy", "Mew"));
my_farm.addAnimal(new Cow());
my_farm.addAnimal(new Bird("Raven", "Nevermore!");
my_farm.addAnimal(new Animal());
my_farm.summary();
```

```java
public class Farm {
    private ArrayList<Animal> animals;

    public void summary() {
        System.out.println("-------------");
        System.out.println("Farm Summary");
        System.out.println("-------------");

        for (int i = 0; i < animals.size(); i++) {
            System.out.println("# " + (i+1));
            // Upcast to Animal to call Animal's display()
            ((Animal) animals.get(i)).display();
            System.out.println();
        }
    }
}
```

Super tricky! Even if you up-cast to a more general class, the method had already been overridden!
Outline

- Subtyping
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class

- Polymorphic Methods
  - Overriding Methods
  - The `super` Reference Revisited (in Methods)
    - Dynamic Dispatch
      - Mechanism

- Conclusion
Recall an Animal container (can be used to store Cows, Birds, ...):

- Certain constructors and methods can be called

```java
Animal a;
```

Known at compile time
Why? How Does Dispatch Work? (1)

- Recall an Animal container (can be used to store Cows, Birds, ...):
  - Certain constructors and methods can be called

```java
Animal a;
// But what goes inside 'a' isn't known // until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
}
else {
    a = new Bird();
}
```

Animal methods
- `getName()`
- `speak()`
- `display()`

Known at compile time
Recall an Animal container (can be used to store Cows, Birds, ...):

- Certain constructors and methods can be called

```java
Animal a;
// But what goes inside 'a' isn't known // until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
// Won't compile because compiler doesn't // know whether milk() will be available!
a.milk();
```

Compiler: "milk()? Not in Animal. It might be defined at runtime, but too dangerous to allow; complain and fuss!"
Imagine if the compiler allowed this.

- What could happen at runtime?
- Program crashes 50% of the time!

```java
Animal a;
// But what goes inside 'a' isn't known
// until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
a.milk(); // works half the time!
```
Now let's consider a method that is overridden in the subclass.
Now let's consider a method that is overridden in the subclass.

```java
Animal a;

// But what goes inside 'a' isn't known
// until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
```

Animal methods
getName()
speak()
display()
Now let's consider a method that is overridden in the subclass.

```
Animal a;
// But what goes inside 'a' isn't known
// until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
```

Known at runtime
Now let's consider a method that is overridden in the subclass.

```java
Animal a;
// But what goes inside 'a' isn't known // until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
a.display();
```

![Diagram showing polymorphism with a random selection between a Cow and a Bird.](image)

- Name: Bessie
- Quote: Moo
- Gallons Milked: 0
Now let's consider a method that is overridden in the subclass.

```java
Animal a;
// But what goes inside 'a' isn't known
// until runtime!!
Random rng = new Random();
if (rng.nextDouble() < 0.5) {
    a = new Cow();
} else {
    a = new Bird();
}
a.display();
// Upcasting isn't telling Java anything
// it doesn't already know, and display() // is still overridden
((Animal) a).display();
```
Outline

- **Subtyping**
  - Polymorphic Variables
  - The `instanceof` Operator
  - The `Object` Class

- **Polymorphic Methods**
  - Overriding Methods
  - The `super` Reference Revisited (in Methods)
  - Dynamic Dispatch
    - Examples

- **Conclusion**
public class A {
    public void test() {
        System.out.println("A");
    }
    public void test2() {
        System.out.println("AA");
    }
}

public class B extends A {
    @Override
    public void test() {
        System.out.println("B");
        test2();
    }
}

public class C extends B {
    @Override
    public void test2() {
        System.out.println("CC");
    }
}

A testing = new B();
testing.test(); //
testing.test2(); //

B testing2 = new C();
testing2.test(); //
testing2.test2(); //

A testing3 = new C();
testing3.test(); //
testing3.test2(); //

What is the output from the following:

Output:

B
AA
AA
B
CC
CC
CC
B
CC
CC
Practice Problems (Self-Test)

- Q1: Assume `Printer` is a subclass of `Device`. Which of these classes must define `getName()` for the following code to compile?

```java
Device dev = new Printer();
dev.getName();
```

- Q2: `Student` does not have a declared superclass. You don't write a `toString()` method. Will the following code compile? What happens when it executes?

```java
Student st = new Student();
String s = st.toString();
```
Outline

- Subtyping
  - Polymorphic Variables
  - The instanceof Operator
  - The Object Class
- Polymorphic Methods
  - Overriding Methods
  - The super Reference Revisited (in Methods)
  - Dynamic Dispatch
- Conclusion
Important Takeaways

Know the following concepts (and when they're used):

- Polymorphism as "container shapes"
- Up-casting, down-casting
- Dynamic dispatch rules
- Method overriding
- Java's automatic code generation for constructors
Tutoring schedule is set

Hwk 1 due Friday

Keywords from previous lecture:
• extends, this(..), super(..)

This set of lectures introduces:
• The `Object` class
• The `instanceof` boolean operator
• The `super.method(..)` call
  - Don't confuse with `super(..)`
• The `@Override` directive
Lab 2 post-mortem:

- Implicit `super()` calls in subclass constructors
  - What if the default constructor in the superclass wasn't written?
- Implicit default constructors inserted by Java if one is not provided
- The `instanceof` operator answers "is-a" questions:

```java
public void test(BasicDie die) {
    if (die instanceof CrookedDie) {
        System.out.println(1);
    }
    if (die instanceof HistoryDie) {
        System.out.println(2);
    }
    if (die instanceof BasicDie) {
        System.out.println(3);
    }
}
```

- Calling methods in:
  - Less specific classes? (Up the chain). More specific classes...? (In rollRepeatedly)
Administrivia 01/26

- Lab 2 solution posted
- Hwk 1 due tonight
- Homework 2 will be posted over the weekend
  - Teams will be randomly assigned too

- Last time:
  - The Object class
  - Method overriding: When a subclass re-defines a method that appeared in a superclass.
    - super.method(..)
    - @Override directive