Outline

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

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

- Conclusion
Recall: `display()` Method Problem

- Before any refactoring:

```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:

```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();
```
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 after refactoring:

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

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

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

```
Bird
- int eggs
  + layEgg()
```
Let's Review the Code Again

- Take out both handouts:
  - Before refactoring Bird and Cow
  - After refactoring Bird and Cow
A Problem with \texttt{display()}\hfill

\begin{itemize}
\item \textbf{Old:}
\begin{verbatim}
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");
    }
}
\end{verbatim}
\end{itemize}

\begin{itemize}
\item \textbf{After refactoring for polymorphism:}
\begin{verbatim}
public class Animal {
    public void display() {
        System.out.println("Name: "+ name);
        System.out.println("Quote: "+ quote);
    }
}
\end{verbatim}
\end{itemize}
Well, `display()` does have different code for Cows and Birds...

- Maybe we never should have abstracted it into `Animal` superclass.

Solution(?): We could undo and add `display()` back into subclasses?

Maybe We Were Wrong?

```java
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()
```
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 you saw in Lab 2)
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 an if-statement for each Animal type.. defeats purpose!
            if (animals.get(i) instanceof Cow) {
                Cow cow = (Cow) animals.get(i);
                cow.display();
            }
            else if (animals.get(i) instanceof Bird) {
                Bird bird = (Bird) animals.get(i);
                bird.display();
            }
        }
    }
}
```
Solution 2?

- Okay, we *should* leave `display()` in `Animal` superclass.

- Try 2:
  - Leave `display` in `Animal`, but in subclasses, **rewrite** the `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 {
    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 {
    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Quote: " + quote);
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```
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 {
    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 {
    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!*
Solution 2.5 (the super reference)

- 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 {
    public void display() {
        super.display(); //reuse Animal's display()
        System.out.println("Milk produced: " + gallonsMilked + " gallons");
    }
}

public class Bird extends Animal {
    public void display() {
        super.display(); //reuse Animal's display()
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```
Solution 2.5 (the super reference)

- And use the `@Override` annotation! (Next)

```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 for defensive-programming.
  - Required in this course
The @Override Annotation

And use the @Override annotation! (Next)

```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(int x) {
        super.display(); //reuse Animal's display()!
        System.out.println("Eggs produced: " + eggs + " eggs");
    }
}
```

<------ This won't compile!!!
Outline

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

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

- Conclusion
Now: `display()` in superclass and overridden in subclasses

- Must be available in `Animal` *(to compile)*
- But it's overridden in its subclasses *(for specific Animal)*
Back in the client classes that call `display()`...

- 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(); // what gets // called?
            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

```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
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
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

```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)

# 4
Name: Anonymous
Quote: Grunt
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

Animal a;

Known at compile time

Animal methods
  .getName()
  .speak()
  .display()
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

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!"
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();
}
```
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
- `.milk()`
- `.getName()`
- `.speak()`
- `.display()`

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();
```

Name: Bessie
Quote: Moo
GallonsMilked: 0
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");
        this.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:

```java
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");
        this.test2();
    }
}

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

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

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

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

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
Hwk 1 due Friday

Walk-In CS tutoring schedule is finalized!
  • Sunday - Thursday: 4-6pm @ 4th Floor CS Lounge

Last time...
  • Object variables as "boxes."
    - Box shapes go from *generic* to more *refined* down the hierarchy
  • Can store an object Y in a X-type variable if X is more generic than Y
Lab 2 post-mortem:

- What happens when you create an object of a subclass?
  - Chaining-effect all the way up to top-most superclass
- 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
- Calling methods in:
  - Less specific classes? (Up the chain)
  - *More specific* classes...? (In `rollRepeatedly`)
Hwk 1 tonight at 11:59p

Keywords from previous lectures:
- extends, this(..), super(..)
- The Object class
- The instanceof boolean operator

This set of lectures introduces:
- The super.method(..) call
  - Don't confuse with super(..)
- The @Override directive
- Dynamic method dispatch