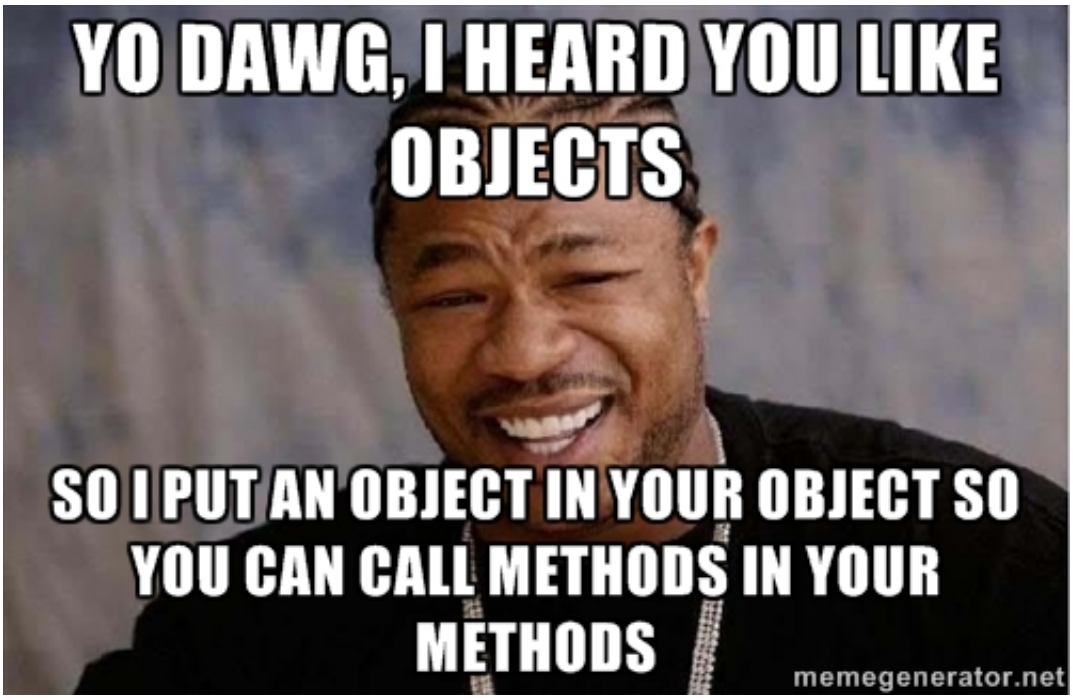
Reusing Classes

Composition

Objects in your objects

Objects can contain references to other objects



Composition provides access to functionality

- Member objects' methods and values can be used inside of the containing object's

Composition is a "has a" relationship

- A house **has a** door

- A unicycle **has a** wheel
- a chair **has a** seat and it **has a** back

Example: House

A house **has** rooms, a door, and windows.

```
class House{
    Room[] rooms;
    public Door frontDoor;
    Window[] windows;
    public House(...){...}
    public void ventilate(){
      for( Window w : windows)
        w.open();
    }
}
```

Example: Unicycle

A unicycle **has** a wheel and a seat.

```
class Unicycle{
  private Wheel wheel;
  private Seat seat;
  public Unicycle(int wheelSize, int height){
    wheel = new Wheel(wheelSize);
    seat = new Seat(height);
  }
  public int getSeatHeight(){return seat.getHeight();}
  public void adjustSeat(int height){ seat.setHeight(height); }
  public int getWheelSize(){ return wheel.getSize(); }
  public void tuneUp(){ if(wheel.isFlat()) wheel.inflate(); }
}
```

Example: Chair

A chair **has** a seat, legs, a back, and armrests (sometimes).

```
class Chair{
  Seat seat;
  Leg[] legs;
  Back back;
  Armrest[] arms;
}
```

Access to member objects

- Subject to the same access modifiers as primitives and methods
- Often kept private, with access methods handling access to member objects
    - This helps to decouple implementation from interface
    - In some cases public member objects are sensible

Inheritance

Terminology

- Subclass - a class that inherits some of its behavior from another class
- Superclass - the class from which a subclass inherits
- Parent/child class - synonyms for superclass and subclass; slightly imprecise Note: the parent/child/ancestor/descendant metaphor is very common, despite being somewhat misleading.

Class inheritance

- classes can inherit the interfaces of classes and extend their feature set
- this is achieved with the `extends` keyword
- If `B` extends `A` then `B` has all of `A`'s public and protected members and methods

    - package access only within the same package

Inheritance is an "is a" or "is like a" relationship

- An SUV **is a** Vehicle
- A corgi **is a** Dog

Example: SUV

```
public class Vehicle{
  public void start(){...}
}

class SUV extends Vehicle{
  public void drive(){
    start();
    ...
  }
}
```

Example: Corgi

```
class Dog{public void wag(){...}}
public class Corgi{
  public static void main(String[] args){
    Corgi thorgi = new Corgi();
    thorgi.wag();
  }
}
```

Upcasting

- Objects can be treated as instances of any superclass in the class heirarchy

```
public class App{
 public static void main(String[] args){
  Dog pembroke = new Corgi();
  pembroke.wag();
  }
}
```

---

break;

Continue;

All classes inherit from the Object class

- `Object` is the immediate superclass if `extends` is omitted

Extending Object class

- valid but redundant
- ...unless you've defined an `Object` class of your own (please don't)

Example: Explicit "extends Object"

Both the same:

```
class Thing {}
```

```
class Thing extends Object {}
```

Any class can have many subclasses

The `Object` class is a good example of this because every class is descended from Object.
Subclasses can also have their own subclasses.

Example: Multiple subclasses

```
class Animal {}

class Mammal extends Animal {}

class Bird extends Animal {}

class Penguin extends Bird {}
```

No multiple- or cyclic-inheritance

Classes in Java can only extend one class type. A class cannot extend one of its subclasses

---

Counter-example: Multiple Inheritance

```
class Five{ public static int getNum(){return 5;} }
class Ten{ public static int getNum(){return 10;} }
class ClimbingShoes extends Five Ten{};
```

If we called `ClimbingShoes.getNum();` what would it return?

Counter-example: Cyclic Inheritance

```
class Foo extends Bar{}

public class Bar extends Foo{}
```

```
$ javac Bar.java
Bar.java:1: error: cyclic inheritance involving Foo
class Foo extends Bar{}
^
1 error
```

Overriding methods

Subclasses can define new behavior for preexisting methods.

```
class Foo{
  public int getNum(){return 4;}
}

class Bar extends Foo{
  public int getNum(){return 8;}
}
```

Overloaded methods aren't hidden

If the superclass has overloaded methods and one is overridden in a subclass, the other overloaded methods are still available in the subclass.

```
class Foo{
  public int newNum(int x){ return 4*x;}
  public int newNum(int x, int y){ return 2* (x + y);}
}
public class Bar extends Foo{
  public int newNum(int x){ return 12*(x-1); }
  public static void main(String[] args){
    Bar louie = new Bar();
    System.out.println(louie.newNum(4, 3));
  }
}
```

```
class Dog {
  public void speak(){
    System.out.println("Woof woof!");
}}
class MultilingualDog extends Dog {
  public void speak(String language){
    switch(language){
      case "Spanish":
          System.out.println("Guau guau!");
          break;
      case "Japanese":
          System.out.println("ワンワン"); // "wan wan"
          break;
      default:
          System.out.println("Bow wow!");
}}}
```

See previous slide for MultilingualDog definition.

```
public class App{
  public static void main(String[] args){
    MultilingualDog doge = new MultilingualDog();
    doge.speak("Spanish");
    doge.speak();
  }
}
```

Output:

`Guau guau!`

`Woof woof!`

---

break;

---

final

The keyword: `final`

- prevents values and references from changing after initialization
- prevents inheritance of classes
- prevents overriding of methods (but not overloading)
- `static final` produces a compile-time constant

Blank final fields

- A field can be marked final but not initialized until the constructor runs
- All constructors must initialized blank finals, or it is an error.

```
public class Foo{
  final int x;
  public Foo(){
    x = 4;
  }
}
```

---

final pitfalls

- final object references cannot be changed but the underlying object can
- private methods are implicitly final
- blank final fields must be initialized in every constructor
- `final` != `finally` -- these are two different keywords