

# COMP 110/L Lecture 22

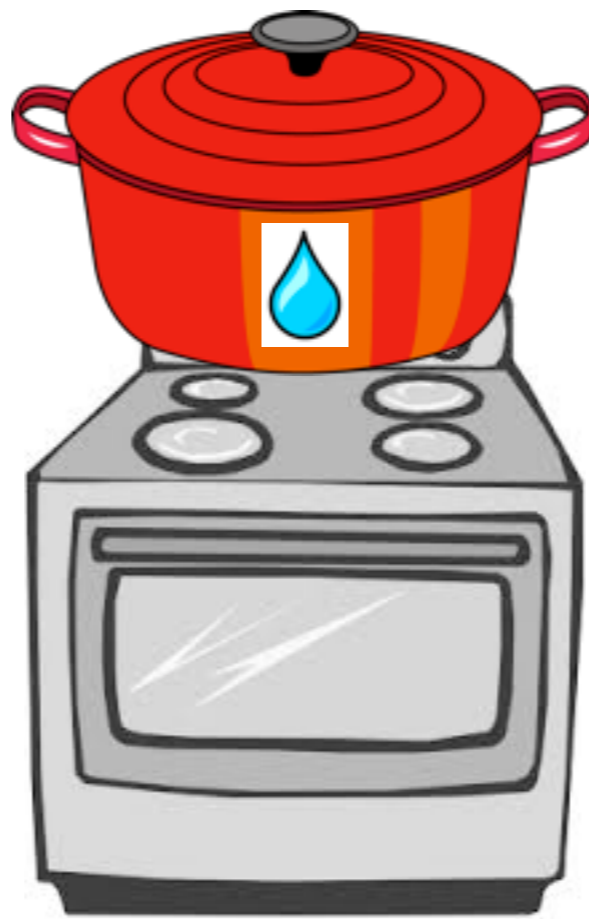
Kyle Dewey

# Outline

- Inheritance
  - `extends`
  - `super`
- Method overriding
- Automatically-generated constructors

# Inheritance

# Recap



-We talked about object-oriented programming being about objects interacting with each other in well-defined ways (i.e., through method calls)

Mammal

-Let's say we have mammal objects...

Mammal



-Along with cat objects...

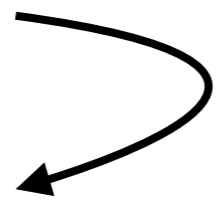
Mammal



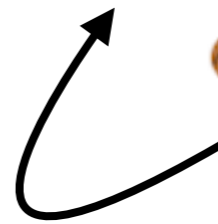
-...and dog objects

-Clearly there is some connection between these, as cats and dogs are both mammals

Mammal



breathe

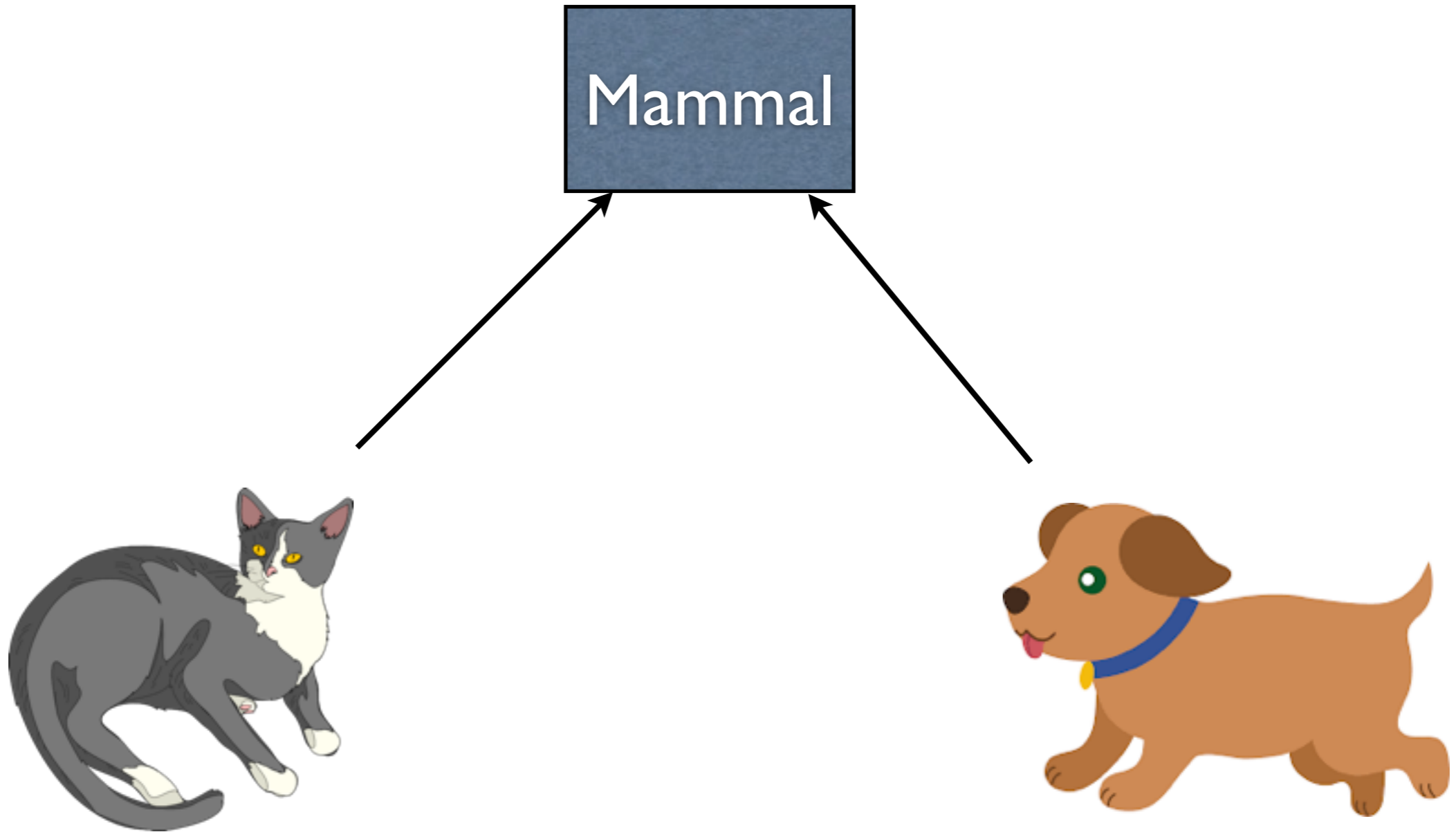


breathe

-Both cats and dogs breathe, but these aren't actions which are unique to cats and dogs

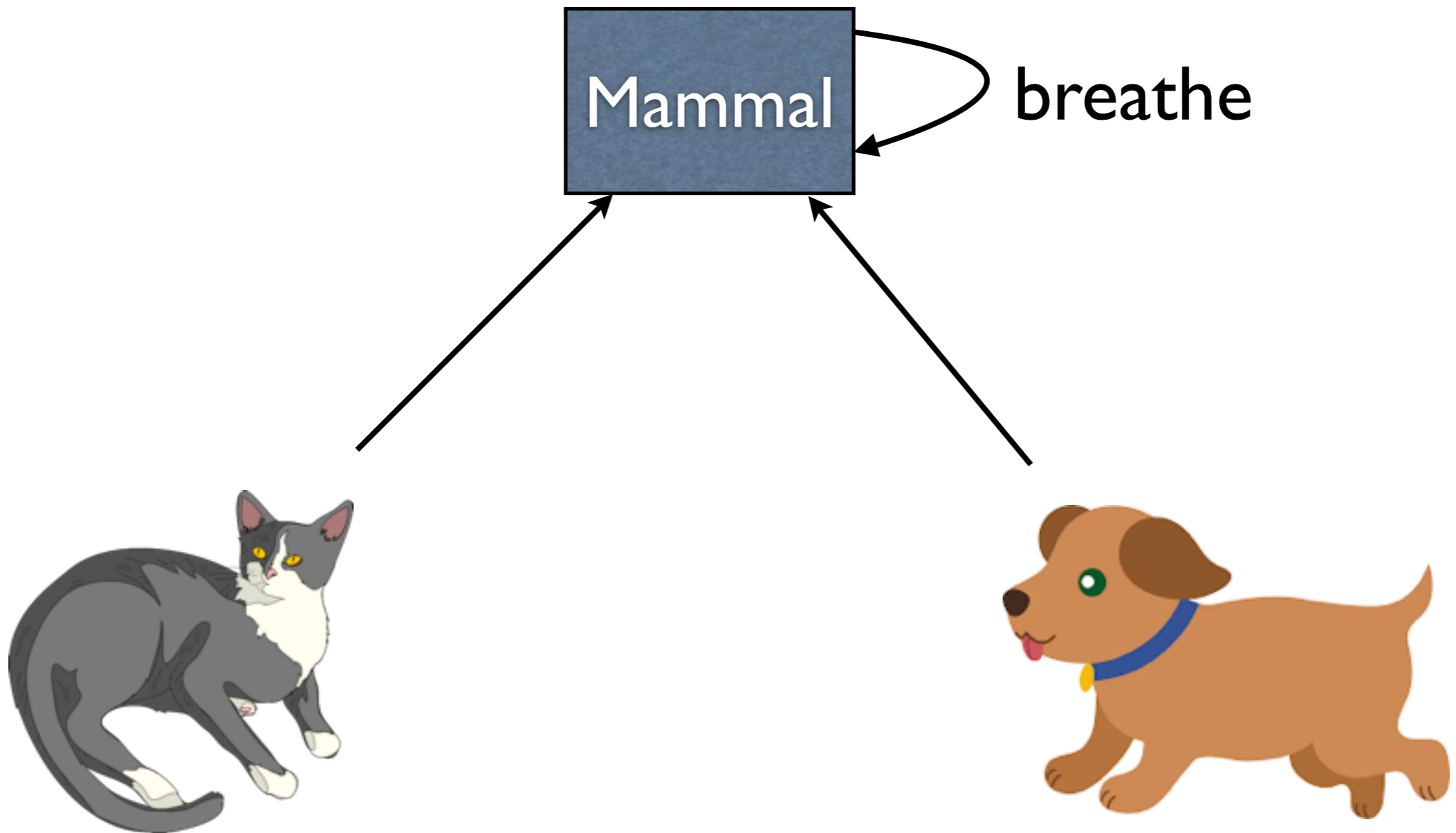


# Inheritance



-Inheritance allows us to effectively say that cats and dogs are both mammals...

# Inheritance



- ...and that mammals breathe
- Transitively, this means that cats and dogs both breathe, too
- The big advantage here code-wise is that we only need to define breathe once

extends

# extends

States that a *subclass* inherits from a *parent* class

# extends

States that a *subclass* inherits from a *parent* class

---

```
public class Mammal {  
    ...  
}
```

# extends

States that a *subclass* inherits from a *parent* class

```
public class Mammal {  
    ...  
}
```

```
public class Cat extends Mammal {  
    ...  
}
```

# extends

States that a *subclass* inherits from a *parent* class

```
public class Mammal {  
    ...  
}
```

```
public class Cat extends Mammal {  
    ...  
}
```

```
public class Dog extends Mammal {  
    ...  
}
```

super



# super

Used to invoke the constructor of the parent class.  
Another name for the parent class is the *superclass*.

# super

Used to invoke the constructor of the parent class.  
Another name for the parent class is the *superclass*.

---

```
public class BaseClass {  
    public BaseClass(String s) {...}  
}
```

# super

Used to invoke the constructor of the parent class.  
Another name for the parent class is the *superclass*.

```
public class BaseClass {  
    public BaseClass(String s) {...}  
}
```

```
public class Child extends BaseClass {  
    public Child(String s) {  
        super(s);  
    }  
}
```

# Example

- `Mammal.java`
- `Cat.java`
- `Dog.java`
- `MammalMain.java`

# Method Overriding

toString () **Revisit**

# toString() Revisit

---

```
public String toString() {  
    ...  
}
```

-In the last lab, you had to define your own toString() method

# toString () Revisit

```
public String toString() {  
    ...  
}
```

```
Rectangle(3, 4)
```

-If you defined it correctly, it would output something like this



# toString() Revisit

```
public String toString() {  
    ...  
}
```

Rectangle(3, 4)

Rectangle@302b09c9

-If you defined it incorrectly, it would give you something like this

# toString () Revisit

```
public String toString() {  
    ...  
}
```

```
Rectangle(3, 4)
```

```
Rectangle@302b09c9
```

**Key point: even without toString () defined,  
a String was still produced.**

-If you defined it incorrectly, it would give you something like this

# Base toString() Origin

- All classes inherit from `Object`,  
**even if** you don't explicitly say so
- `Object` **defines its own** `toString()`  
**that produces** `Rectangle@302b09c9`

# Base toString() Origin

- All classes inherit from Object, **even if** you don't explicitly say so
- Object **defines its own** toString() **that produces** Rectangle@302b09c9

```
public class Object {  
    public String toString() { ... }  
}
```

-So somewhere in Java, there is a class definition like this

# Base toString() Origin

- All classes inherit from Object, **even if** you don't explicitly say so
- Object **defines its own** toString() **that produces** Rectangle@302b09c9

```
public class Object {  
    public String toString() { ... }  
}
```

```
public class Rectangle { ... }
```

-You defined your Rectangle class like this

# Base toString() Origin

- All classes inherit from Object, **even if** you don't explicitly say so
- Object **defines its own** toString() **that produces** Rectangle@302b09c9

```
public class Object {  
    public String toString() { ... }  
}
```

```
public class Rectangle { ... }
```

```
public class Rectangle extends Object {  
    ...  
}
```

-That code without the explicit extends Object is equivalent to code that does explicitly extend Object

-This is how we end up with Object's toString() method

# Overriding Methods

- You can *override* a method definition in a base class by defining a method with the same signature in a subclass
- The method in the subclass will execute *instead of* the method in the parent class

# Overriding Methods

- You can *override* a method definition in a base class by defining a method with the same signature in a subclass
- The method in the subclass will execute *instead of* the method in the parent class

```
public class Rectangle {  
    public String toString() {  
        ...  
    }  
}
```

-So when you were defining your toString()...



# Overriding Methods

- You can *override* a method definition in a base class by defining a method with the same signature in a subclass
- The method in the subclass will execute *instead of* the method in the parent class

```
public class Rectangle extends Object {  
    public String toString() {  
        ...  
    }  
}
```

–...you were actually overriding the toString() in Object, since Rectangle implicitly extends from Object

–If you didn't define the toString() method right (e.g., having the wrong signature), then you don't override Object's toString(), and so you end up with Object's (mostly useless) toString() getting used instead of your own

# Example

- `OverrideBase.java`
- `OverrideSub.java`
- `OverrideMain.java`

# Automatically- Generated Constructors

# Automatic Constructors

If you don't define any constructors,  
Java will define one for you which takes no arguments.

# Automatic Constructors

If you don't define any constructors,  
Java will define one for you which takes no arguments.

---

```
public class MyClass {  
}
```

-So if you write this code...

# Automatic Constructors

If you don't define any constructors, Java will define one for you which takes no arguments.

```
public class MyClass {  
}
```

```
public class MyClass {  
    public MyClass() {}  
}
```

-...you actually get this code

-The code itself isn't written in the file, but it will behave as if it were written in the file

**Example:**

`AutomaticConstructor.java`

# Automatic Constructors

This also applies to subclasses,  
as long as the base class has a no-argument constructor



# Automatic Constructors

This also applies to subclasses,  
as long as the base class has a no-argument constructor

---

```
public class MyBase {}  
public class MySub extends MyBase {}
```

-So if you were to write this code...

# Automatic Constructors

This also applies to subclasses,  
as long as the base class has a no-argument constructor

```
public class MyBase {}  
public class MySub extends MyBase {}
```

```
public class MyBase {  
    public MyBase() {}  
}  
  
public class MySub extends MyBase {  
    public MySub() { super(); }  
}
```

-...it's actually equivalent to this code

# Automatic Constructors

This also applies to subclasses,  
**as long as the base class has a no-argument  
constructor**

-If this doesn't hold, then the code won't compile

# Automatic Constructors

This also applies to subclasses,  
**as long as the base class has a no-argument  
constructor**

```
public class MyBase {  
    // explicit non-no-arg constructor  
    // defined - no automatically  
    // generated constructors  
    public MyBase(int x) {}  
}  
public class MySub extends MyBase {}
```

-If this doesn't hold, then the code won't compile

# Automatic Constructors

This also applies to subclasses,  
**as long as the base class has a no-argument constructor**

```
public class MyBase {  
    // explicit non-no-arg constructor  
    // defined - no automatically  
    // generated constructors  
    public MyBase(int x) {}  
}  
public class MySub extends MyBase {  
    public MySub() { super(); }  
}
```

-If this doesn't hold, then the code won't compile

# Automatic Constructors

This also applies to subclasses,  
**as long as the base class has a no-argument constructor**

```
public class MyBase {  
    // explicit non-no-arg constructor  
    // defined - no automatically  
    // generated constructors  
    public MyBase(int x) {}  
}  
    Does not exist - code will not compile  
public class MySub extends MyBase {  
    public MySub() { super(); }  
}
```

-If this doesn't hold, then the code won't compile