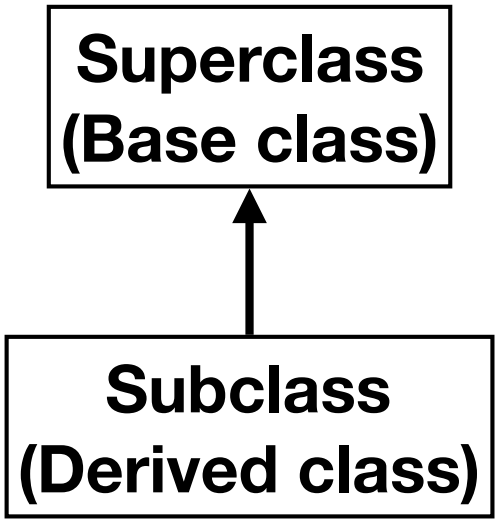
Class Hierarchy and Interfaces

David Greenstein
Monta Vista High School

Inheritance

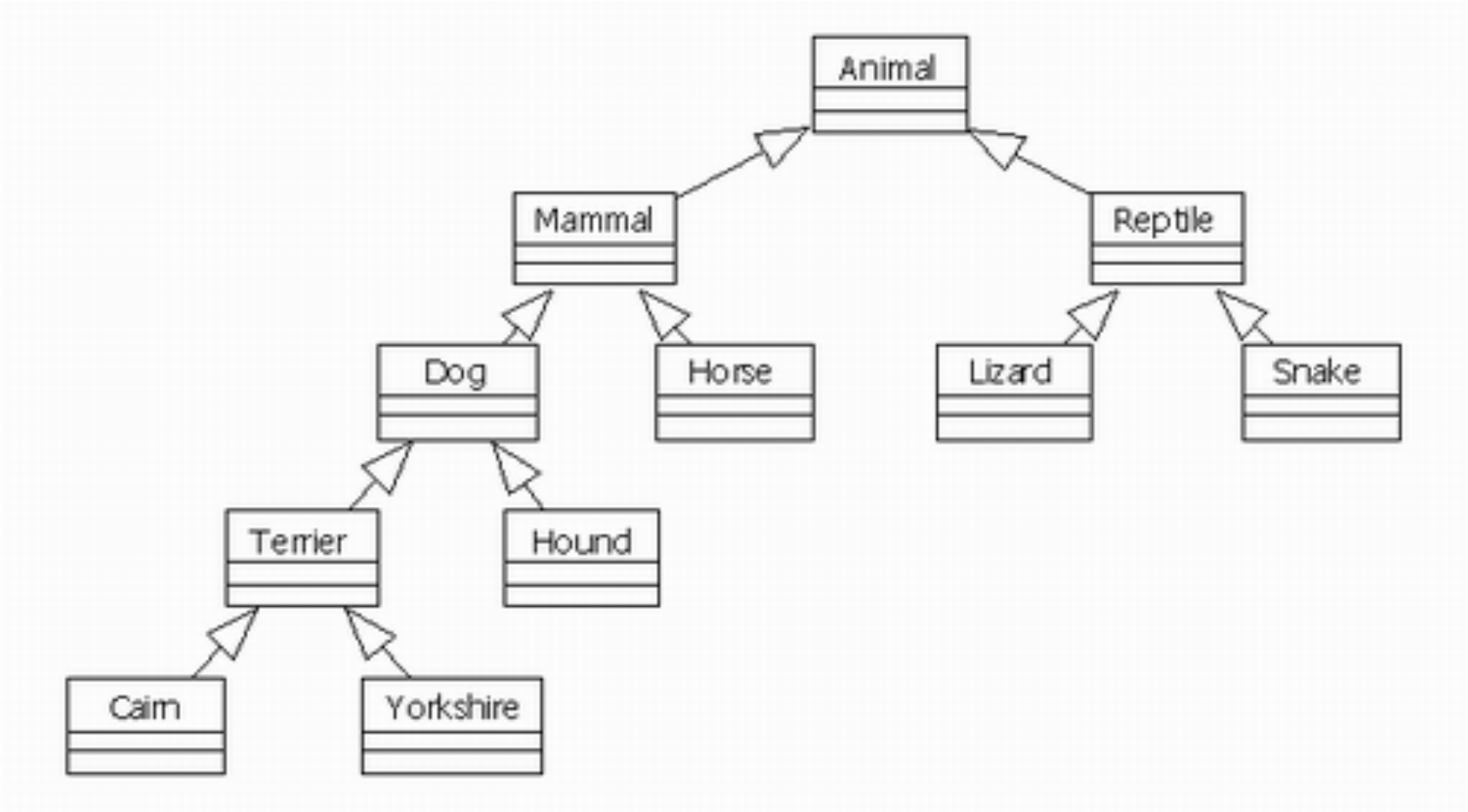
- Inheritance represents the IS-A relationship between objects.
  - an object of a subclass IS-A(n) object of the superclass



The Subclass
extends
the Superclass

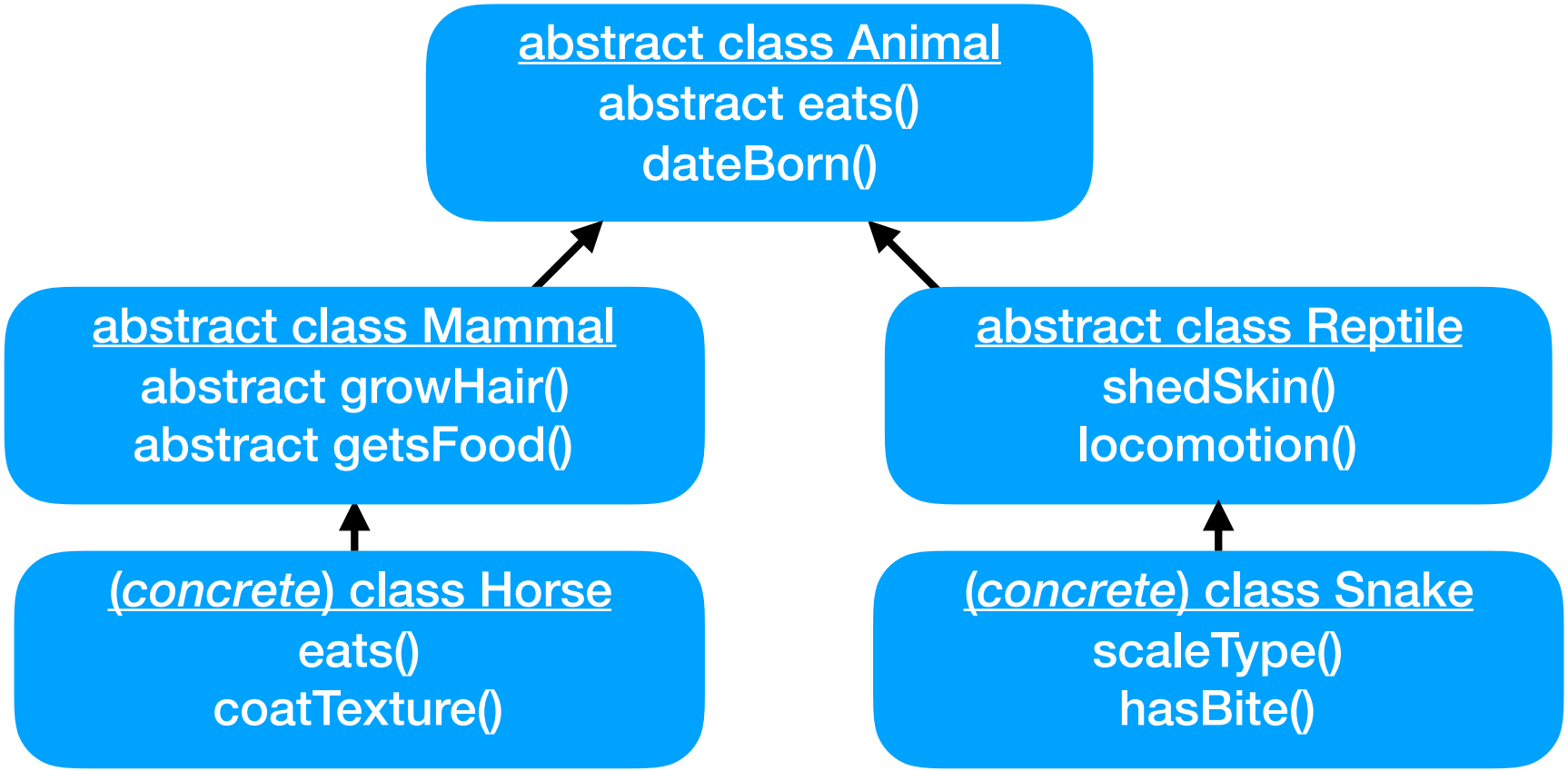
Class Hierarchy

- Using inheritance, a programmer can define a hierarchy of classes.



Class Hierarchy (cont)

- Hierarchy helps reduce duplication of code by **factoring out** common code from similar classes into a common superclass.



Class Hierarchy (cont)

- Hierarchy helps reduce duplication of code by letting you write more general methods in client classes.

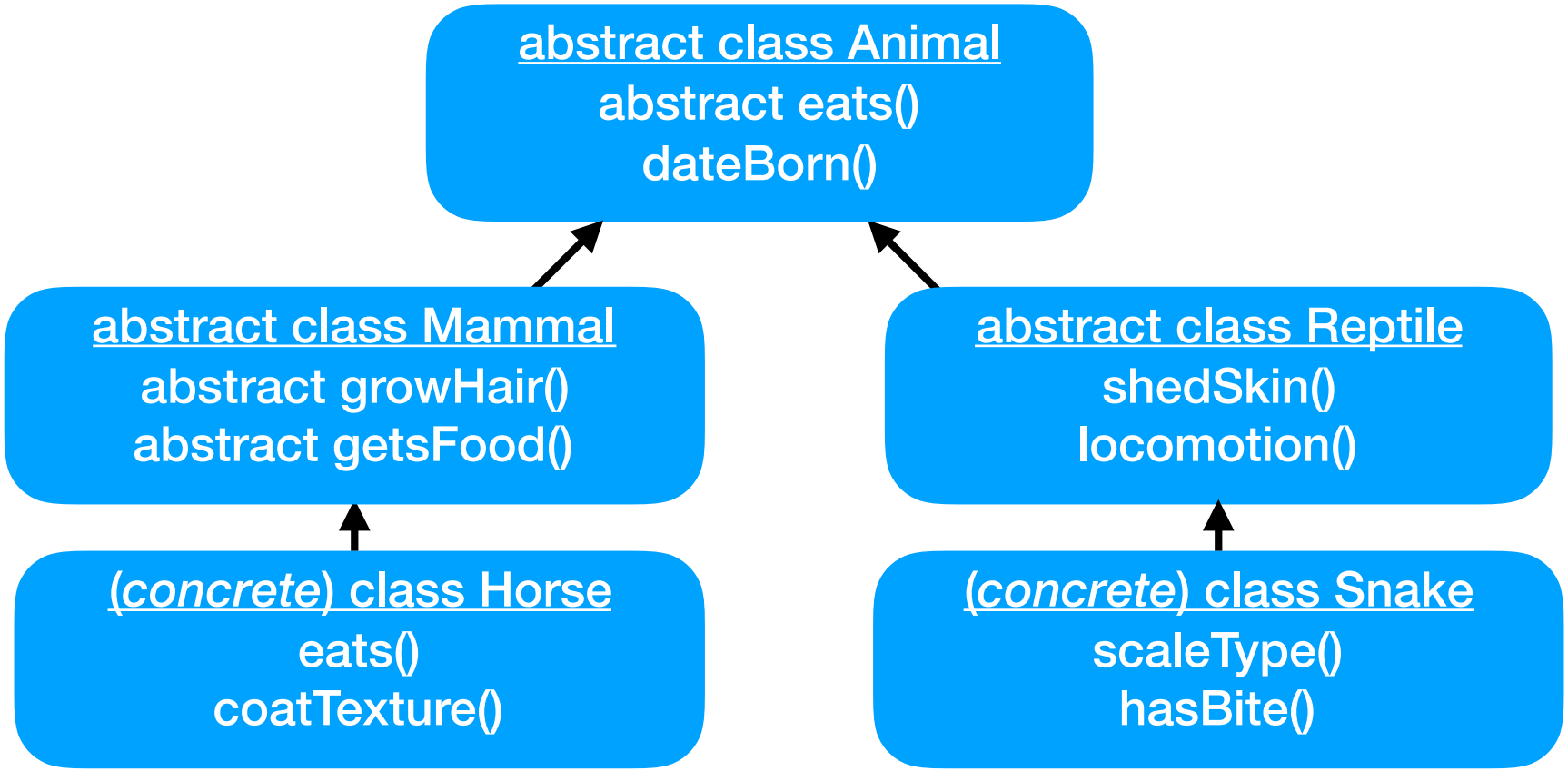
```
public void dateBorn(Horse h)
{
}
public void dateBorn(Dog d)
{
}
```

```
public void dateBorn(Mammal m)
{
}
```

Works for a **Horse** or **Dog** or any **Mammal**

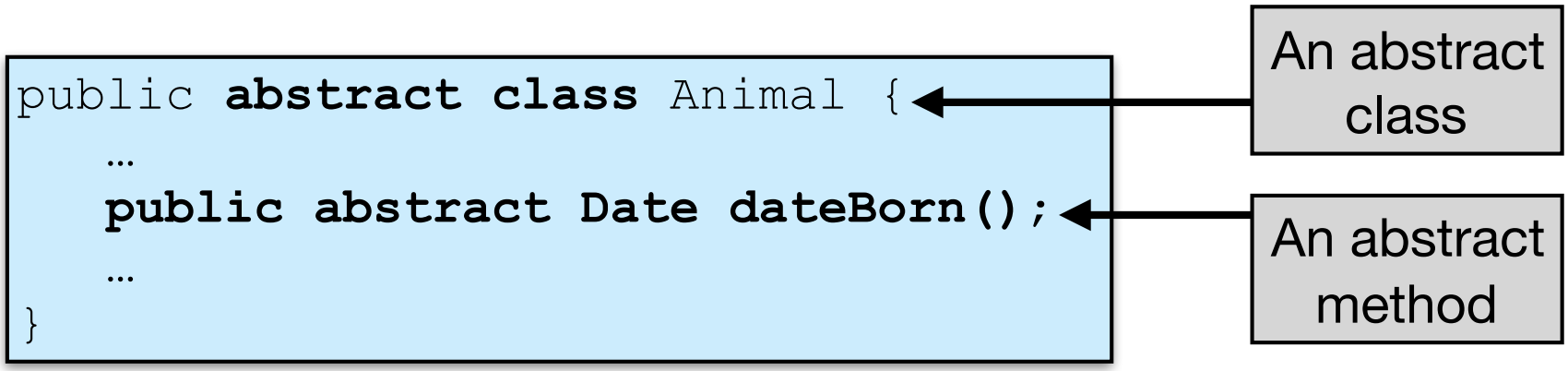
Abstract Classes

- Some superclasses are generic classes used as a basis for other subclasses. These are called **abstract classes** (e.g. Animal).



Abstract Classes (cont)

- Abstract classes are **closer to the root** of the hierarchy; they describe more abstract objects.
- **Abstract classes** must define one or more abstract methods.
- Conversely, any class that has one or more abstract methods must be **abstract**.



Abstract Classes (cont)

- **Abstract classes** are not instantiated as objects (no **new**) because they are too general.
- An **abstract** class can still define constructors (ie. implement them).
- A **concrete class** is one which has no abstract methods.
  - Therefore, the class cannot be labeled **abstract**.
- An **abstract** class can be derived (extend) from a **concrete** class.

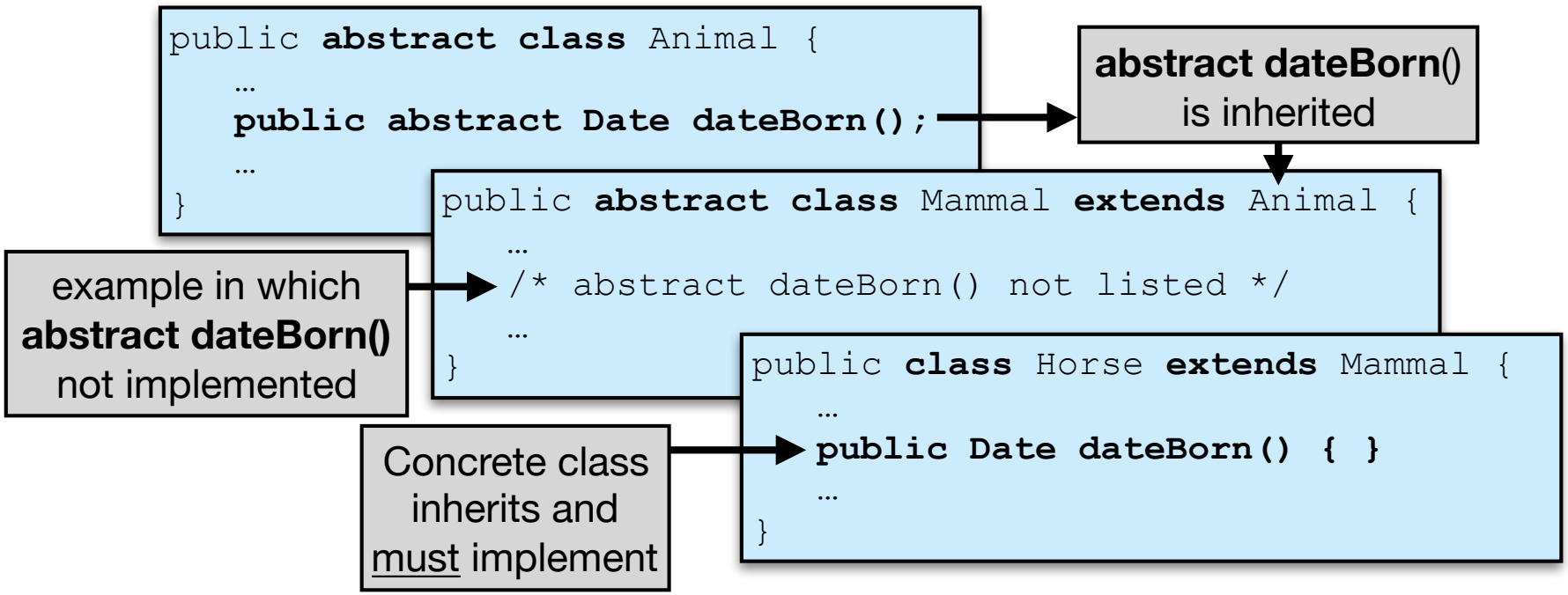
```
public abstract class Animal {
    …
    public abstract Date dateBorn();
    …
}
```

An abstract class

An abstract method

Abstract Classes (cont)

- If the superclass and subclass are both **abstract**, then the subclass implicitly inherits all of the **abstract** methods of the superclass.
- An **abstract** class may or may not implement the **abstract** methods of its superclass.
- If not implemented by the **abstract** class(es), eventually a derived class in the hierarchy must implement the **abstract** method.



Abstract Methods

- **Abstract methods** are not implemented so they have no body, just a signature with a semicolon.

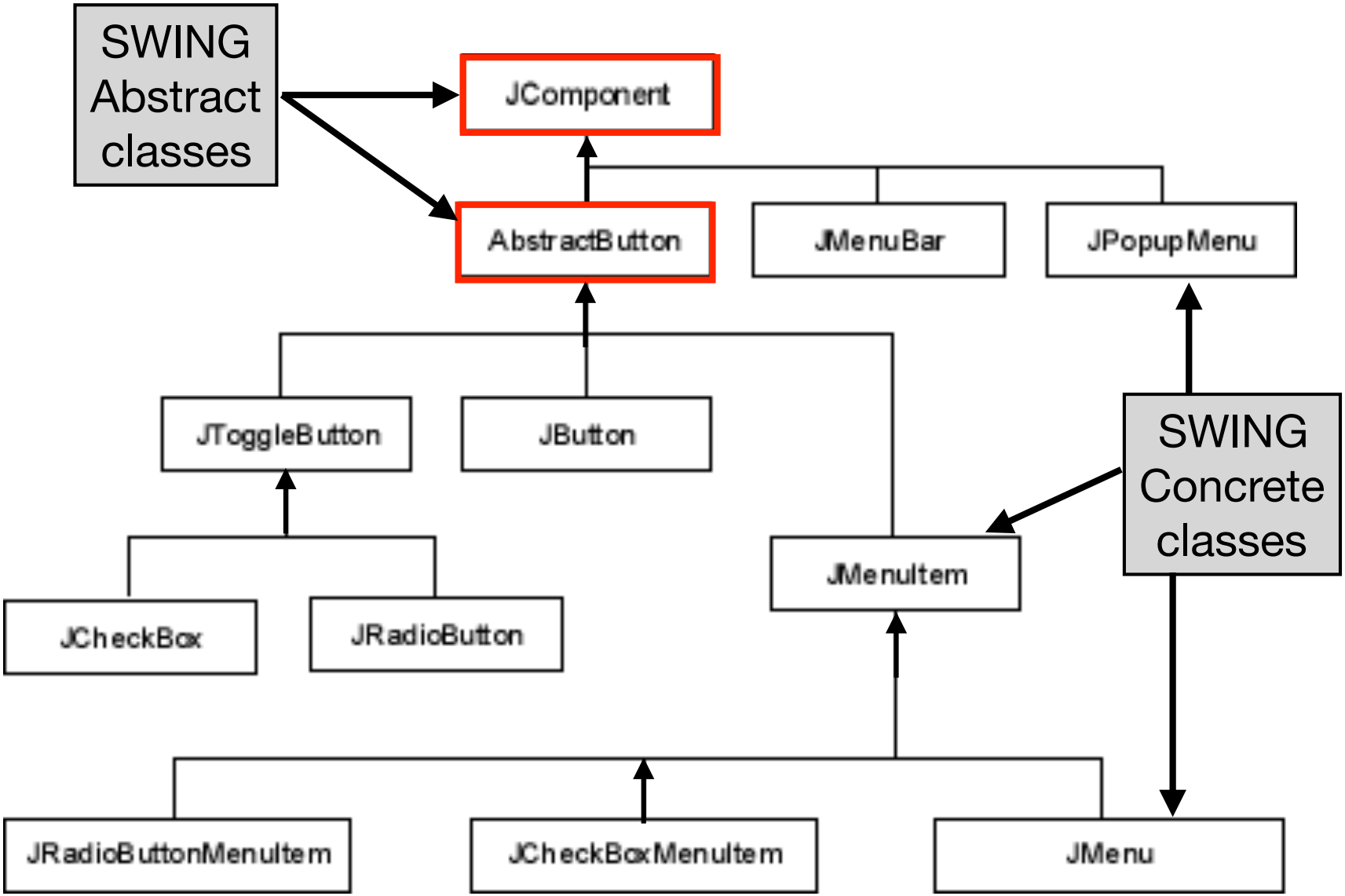
- **Abstract methods** provide the compiler an opportunity for additional error checking.

```
public abstract class Animal {
    …
    public abstract Date dateBorn();
    …
}
```

Just signature; no body

- **Abstract fields or constructors????** **DO NOT EXIST!!!!**

Abstract Classes (cont)



The Object Class

- The superclass of all classes is the **Object** class from the **java.lang** package.

- The **Object** class is a concrete class.

```
public class Object
{
    public String toString {...}
    public boolean equals (Object other) {... }
    public int hashCode() { ... }

    // a few other methods
    ...
}
```

Methods overridden as necessary

The Object Class (cont)

- All classes extend from the **Object** class, even the classes you write.

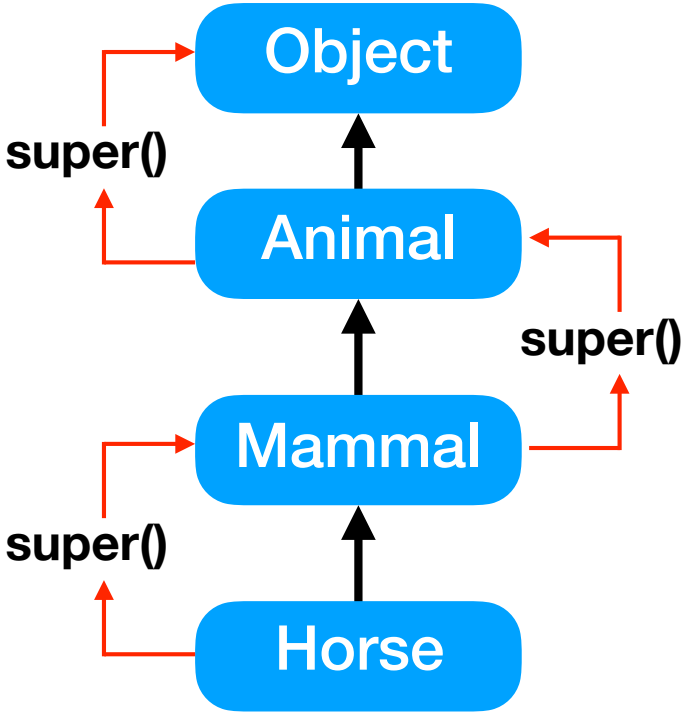
- If not explicitly coded, the class extends **Object**.

```
public class MyClass extends Object
{
    // a few fields

    // maybe a constructor defined

    // a few other methods
    ...
}
```

If not explicitly mentioned, the compiler "**extends Object**"

Constructor Hierarchy

- Constructors are invoked through a superclass chain - up the inheritance hierarchy. (not inherited)



- If no constructor is explicitly defined in a class, then a no-args constructor is generated automatically by the compiler.

Implicit Constructor

- The no-args constructor generated by the compiler calls the constructor of the superclass (**super()**).
- **Warning!** The superclass must have a no-args constructor or a syntax error is generated.

```
public class MyClass extends MySuper
{
    // a few fields

    /* no constructor defined */

    // a few other method
    ...
}
```

Your code

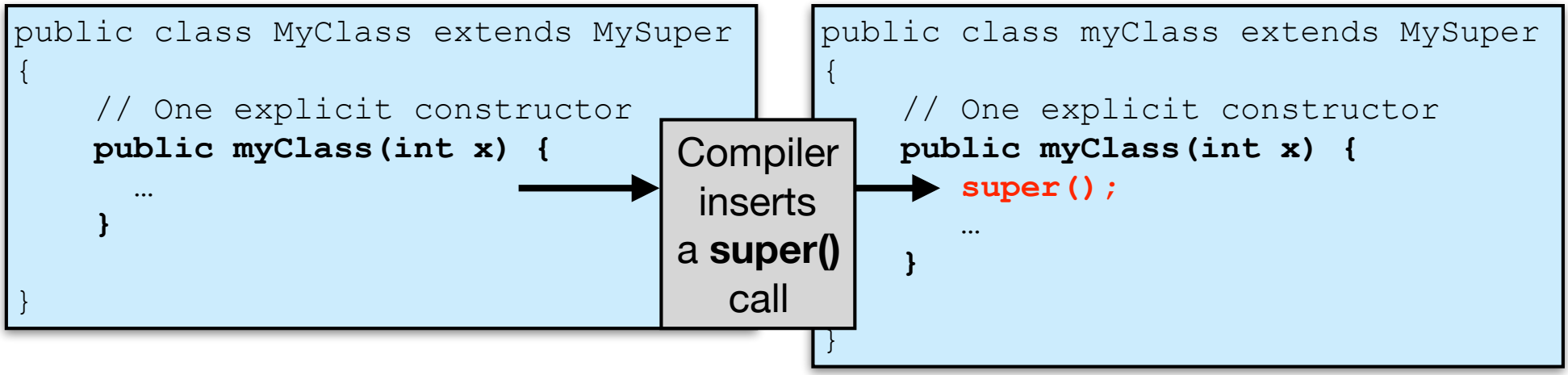
What the compiler inserts

```
public class MyClass extends MySuper
{
    // a few fields

    public myClass() { super(); }

    // a few other methods
    ...
}
```

Explicit Constructor

- If you explicitly write a constructor in a class, then the compiler will not generate a constructor, not even a no-args constructor!

- ***But*** inside your constructor, a **super()** call might get inserted by the compiler if you do not explicitly include it.
  - If you put **super()** in your first line, then the compiler adds nothing.
  - If you do not put **super()** in your first line, then the compiler adds a no-args **super()** automatically.

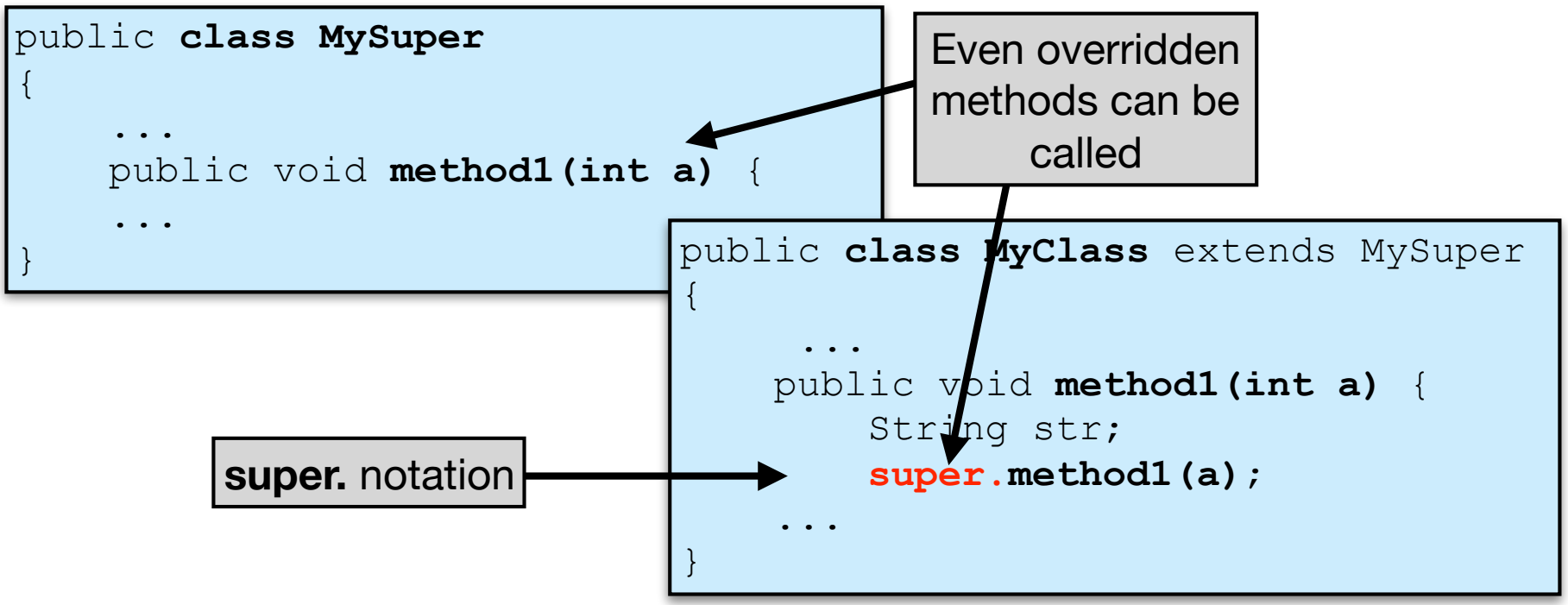


super Constructor Call

- Only a constructor can call a **super** constructor.
- By default, one of the superclass's constructors is always called. (with the exception of the **Object** class)
- If you explicitly put a **super** in your constructor, it must be on the first line of the constructor.
- The **super** method can have parameters. Whatever parameters are in the **super** call, there must be a superclass constructor with the same signature.
- If you do not explicitly call **super**, then the superclass's no-args constructor is called by default.

super Method Call

- Subclass methods can call superclass methods using "**super.**" (dot) notation.
- A "**super.**" method call can be called anywhere inside a subclass method and it can be done one or more times.
- Superclass method calls only go up one level of hierarchy. There is no such thing as **super.super**!!



Polymorphism

- Ensures that the correct method is called for an object of a specific type, even when that object is disguised as a reference to a more generic type, that is, the type of the object's superclass or some ancestor higher up the inheritance line.

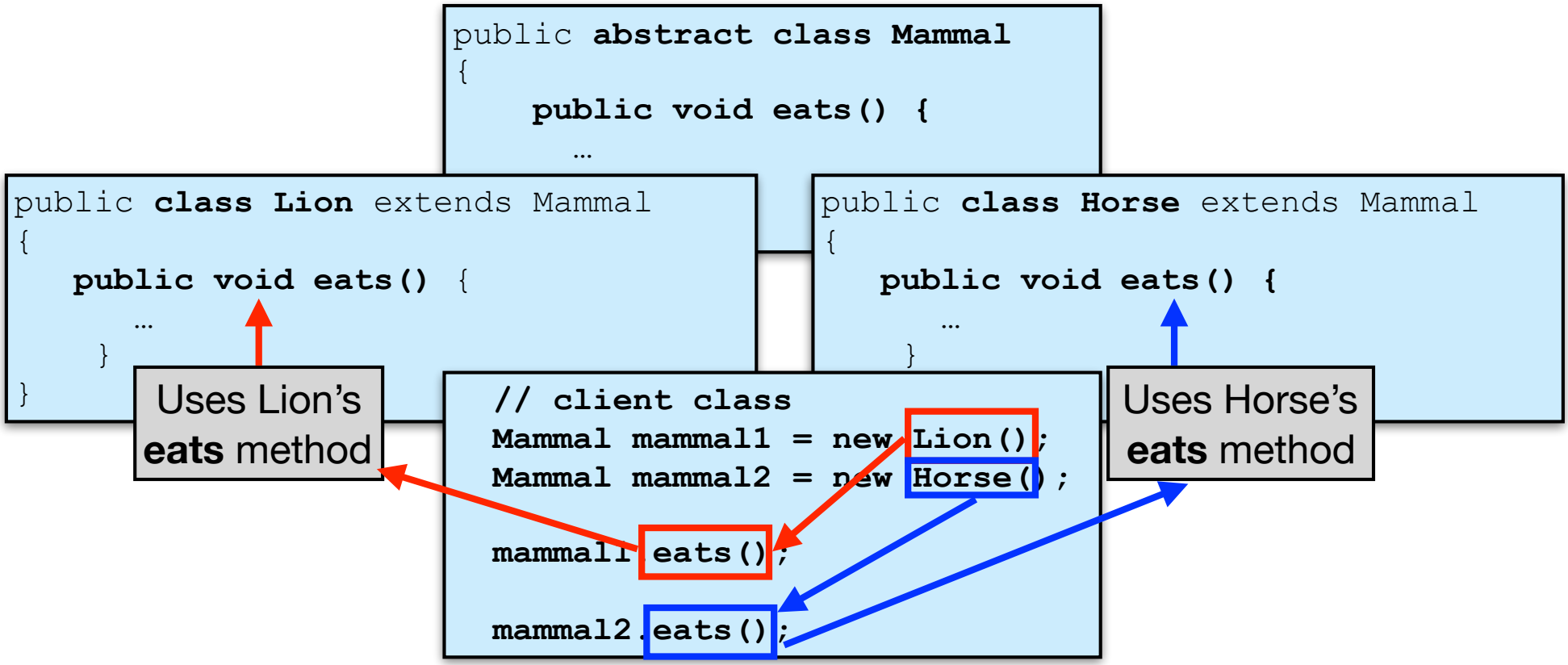
- Once you define a common superclass, polymorphism is just there — no need to do anything special.

```
// client class
Mammal mammal1 = new Lion();
Mammal mammal2 = new Horse();

mammal1.eats();

mammal2.eats();
```

Polymorphism (cont)

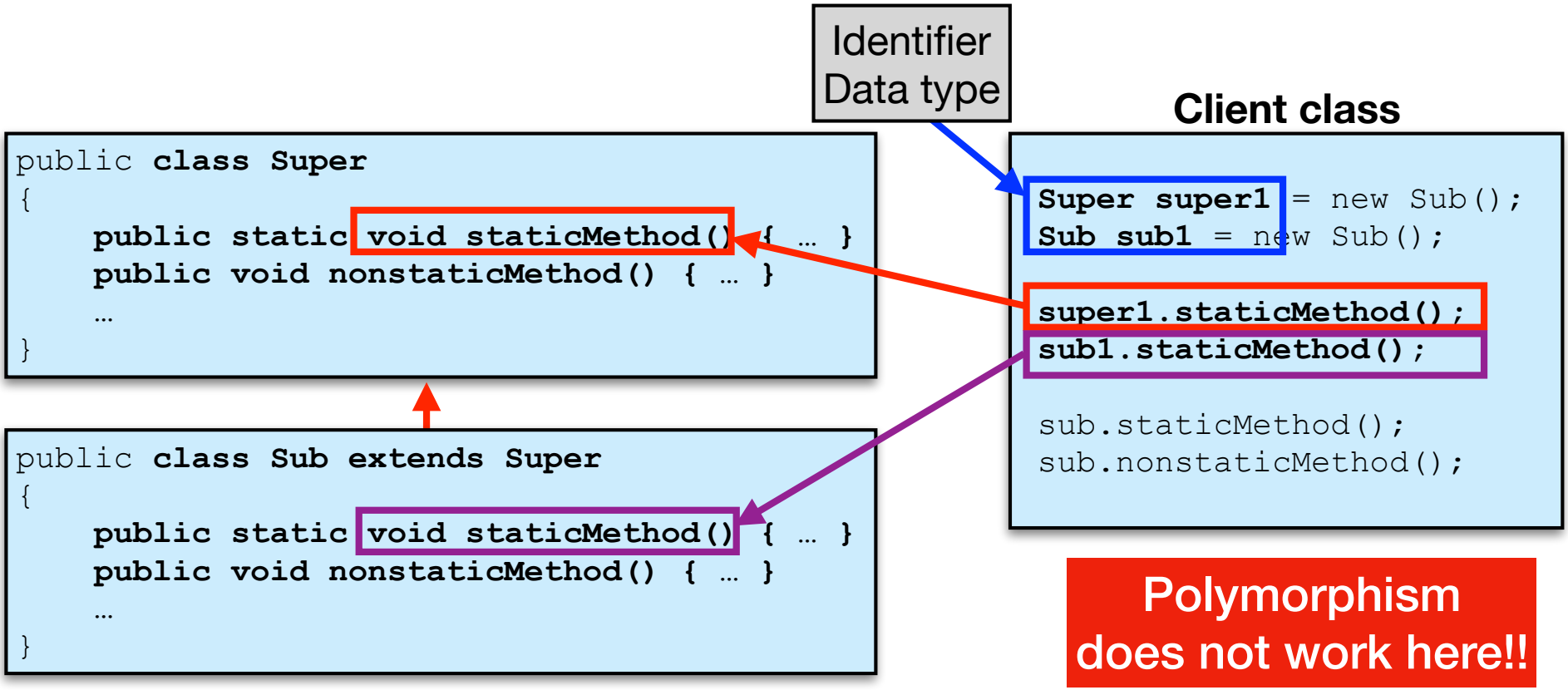
- For example, subclasses **Lion** and **Horse** each extend the **Mammal** class.

- The identifier is **Mammal**, but ("under the hood") **eats()** still executes the correct **Lion** version.



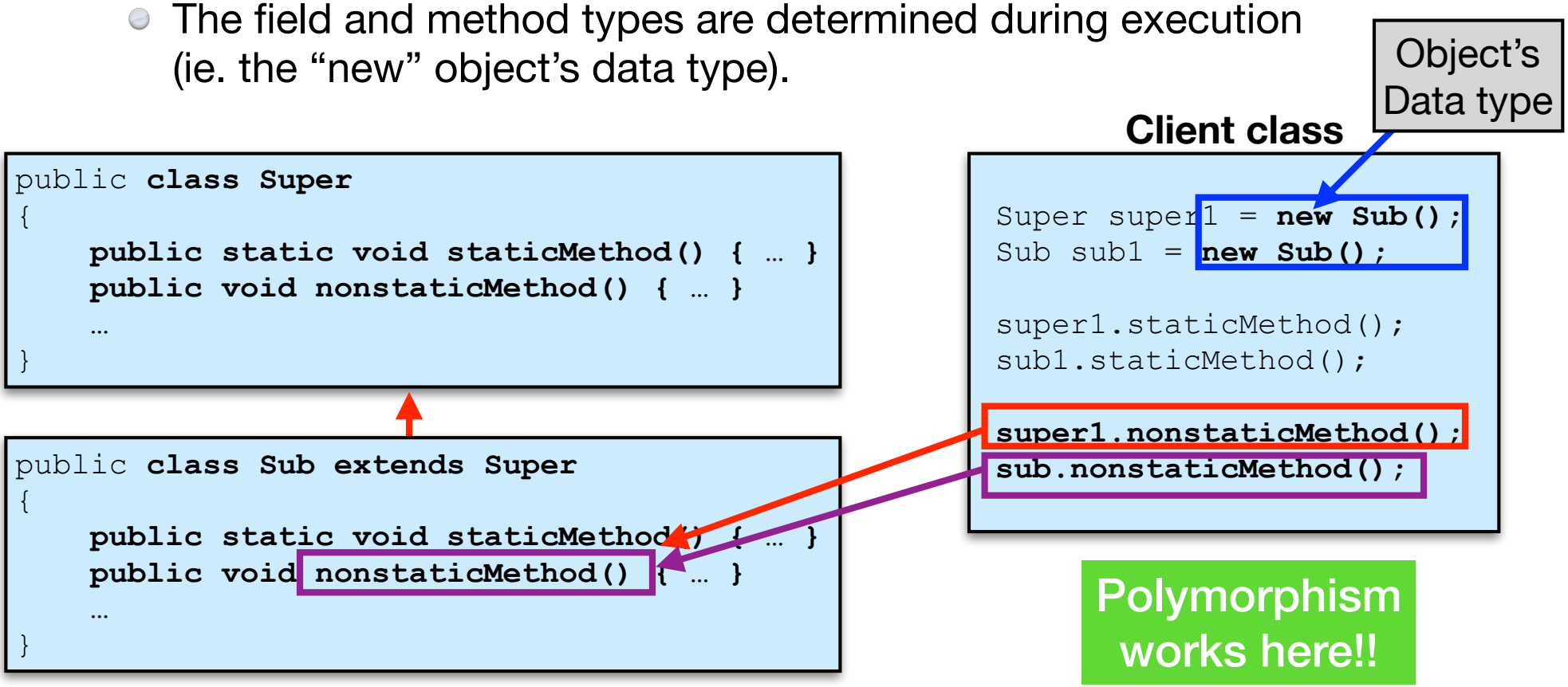
Non-static vs. Static Methods

- **Static methods** are "bound" to the identifier's data type (ie. class).
  - The fields and methods are determined at compile time.



Non-static vs. Static Methods

- **Static methods** are "bound" to the identifier's data type (ie. class).
  - The fields and method types are determined at compile time.
- **Non-static methods** are "bound" to the object's data type.
  - The field and method types are determined during execution (ie. the "new" object's data type).



Interfaces

- An **interface** in Java is like an abstract class, except:
  - it has no constructors.
  - all of its methods are implicitly **public abstract**
  - all of its fields are implicitly **public static final**

```
public interface Herd {
    …
    void setHerdSize(int num);
    int getHerdSize();
    …
}
```

Implied **public abstract** methods

Interfaces (cont)

- A **concrete class** that implements an **interface** must implement every method mentioned in the **interface**.

- Like abstract methods, **interfaces** provide the compiler an opportunity for additional error checking.

```
public interface Herd {
    …
    void setHerdSize(int num);
    int getHerdSize();
    …
}
```

Class **Horse** must implement all the abstract methods mentioned in **Herd**

```
public class Horse implements Herd {
    …
    public void setHerdSize(int num)
    { … }
    public int getHerdSize()
    { … }
    …
}
```

Interfaces (cont)

- **Abstract** classes can also implement **interfaces**.
- An **abstract** class is not obligated to implement the **abstract** methods of the **interface**, but they are allowed to implement.
- **Concrete** classes must implement all abstract methods inherited from their superclass (***extends***) and their interfaces (***implements***).

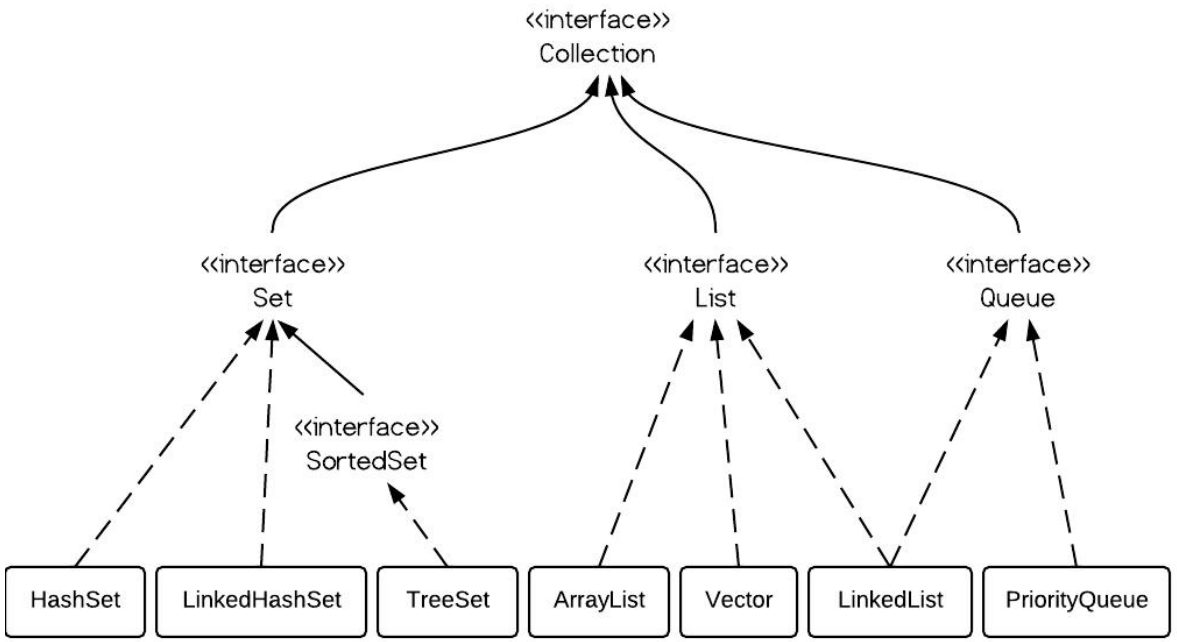
```
public interface Herd {
    …
    void setHerdSize(int num);
    int getHerdSize();
    …
}
```

Abstract classes may implement the method or let it pass to derived class.

```
public abstract class Mammal implements Herd {
    …
    public void setHerdSize(int num){ … }
    // int getHerdSize() passed to derived class
    …
}
```

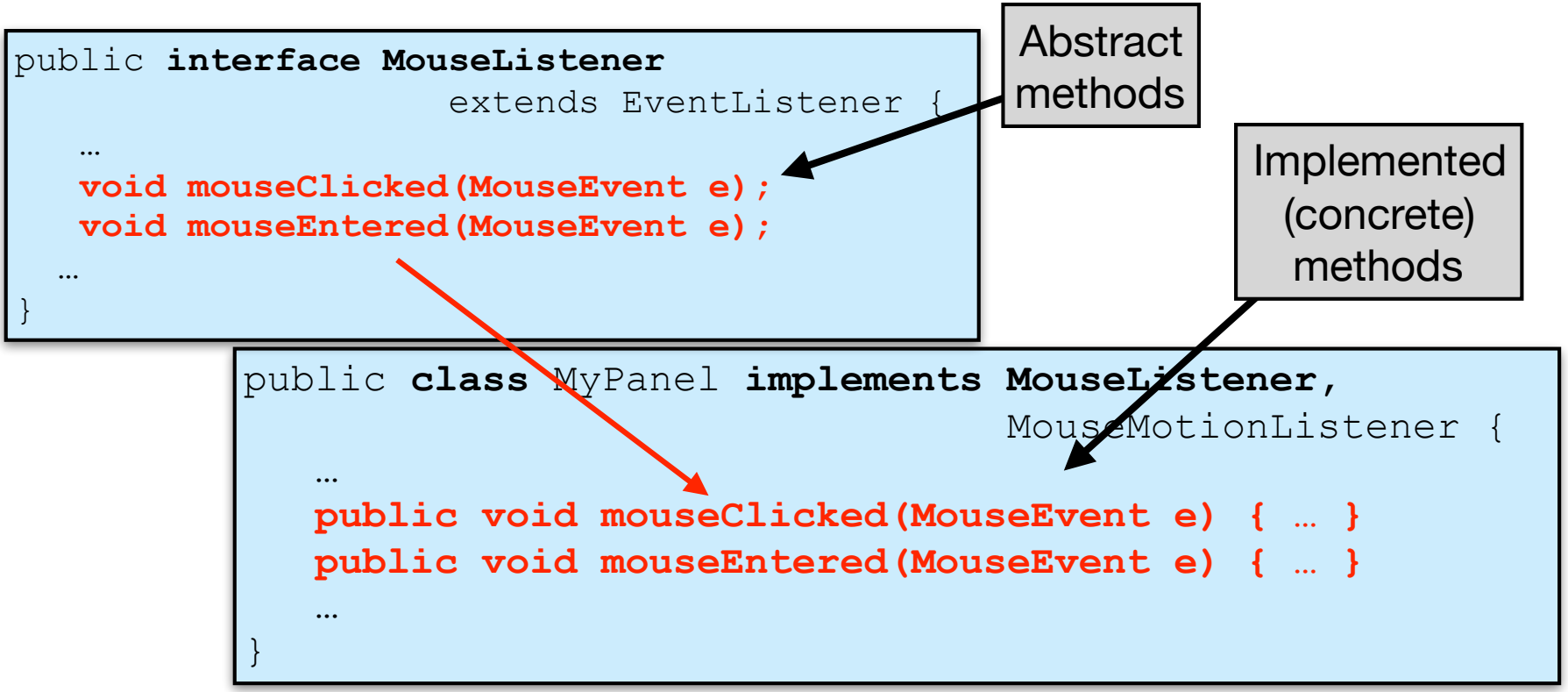
Interfaces (cont)

- **Interfaces** cannot extend classes nor implement other interfaces.
- **Interfaces** are not in the hierarchy of classes.
- On the other hand, **interfaces** can extend other **interfaces** creating their own hierarchy.



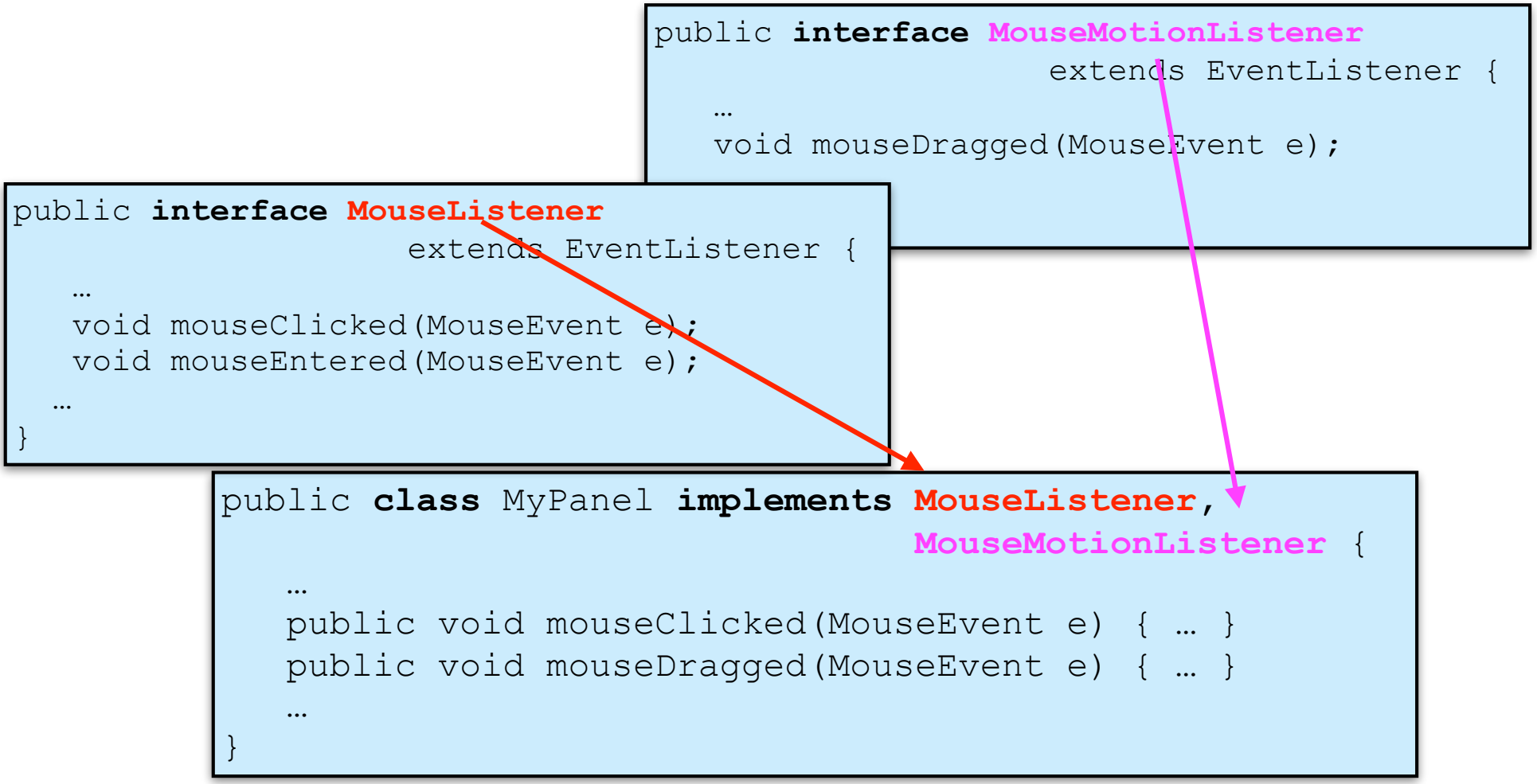
Interfaces (cont)

- **Concrete classes** must implement all the **abstract** methods of the **interface**.
- **Abstract classes** do not have to implement the **abstract** methods of the **interface**.



Interfaces (cont)

- **Concrete and abstract classes** can implement more than one **interface**.



Interfaces (cont)

- Like an **abstract** class, an **interface** supplies a secondary data type to objects of a class that implements that **interface**.

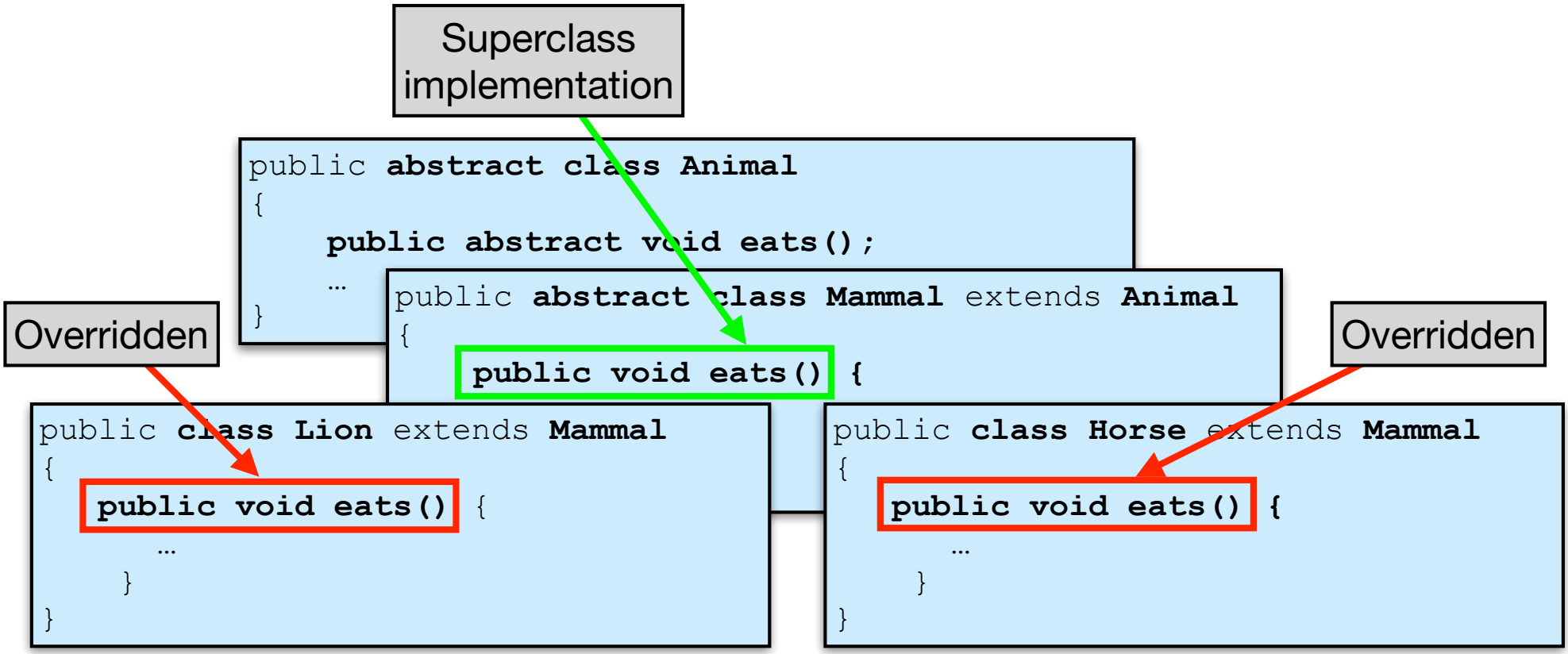
- You can declare variables and parameters of an **interface** type.

```
Herd flicker = new Horse();
```

- **Polymorphism** fully applies to objects disguised as **interface** types.

Overriding Methods

- To **override** a method is to redefine (reimplement) a superclass' method in a subclass using the same signature.



Questions?