Inheritance

Mason Vail
Boise State University Computer Science

Pillars of Object-Oriented Programming

- Encapsulation
- **Inheritance**
- Polymorphism
- Abstraction (sometimes)

Why Inherit?

- Code Reuse - build on code that has already been developed and tested
- Minimize Code Duplication - consolidate duplicate code into a common ancestor
- Set up for Polymorphism - its own topic

Common, Highly-Interchangeable Names

Old Class

- Parent class
- Superclass
- Base class

New Class

- Child class
- Subclass
- Derived class

Explicit Inheritance

```
public class NewClass extends OldClass {

   //without writing a single new line of
   //code, NewClass starts with all
   //properties and methods of OldClass

}
```

Implied Object Inheritance

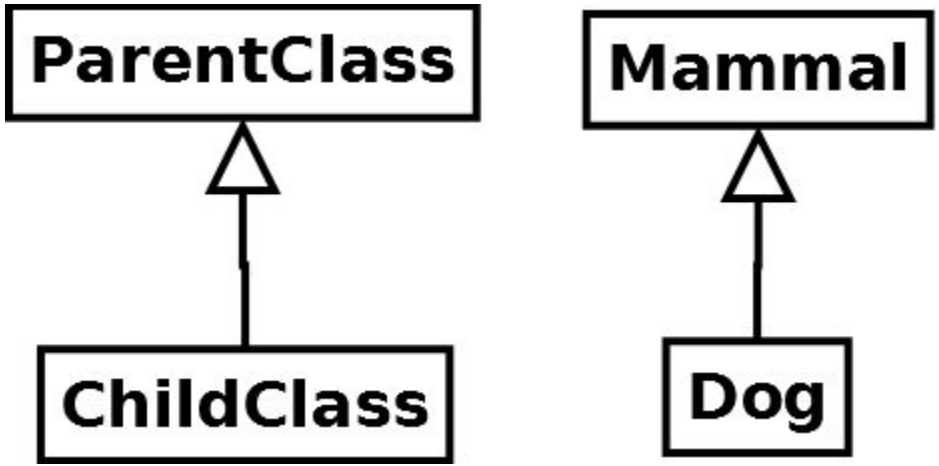
If no parent is explicitly declared, Java automatically extends the Object class. Whether directly or indirectly, all classes are descendents of Object.

implied

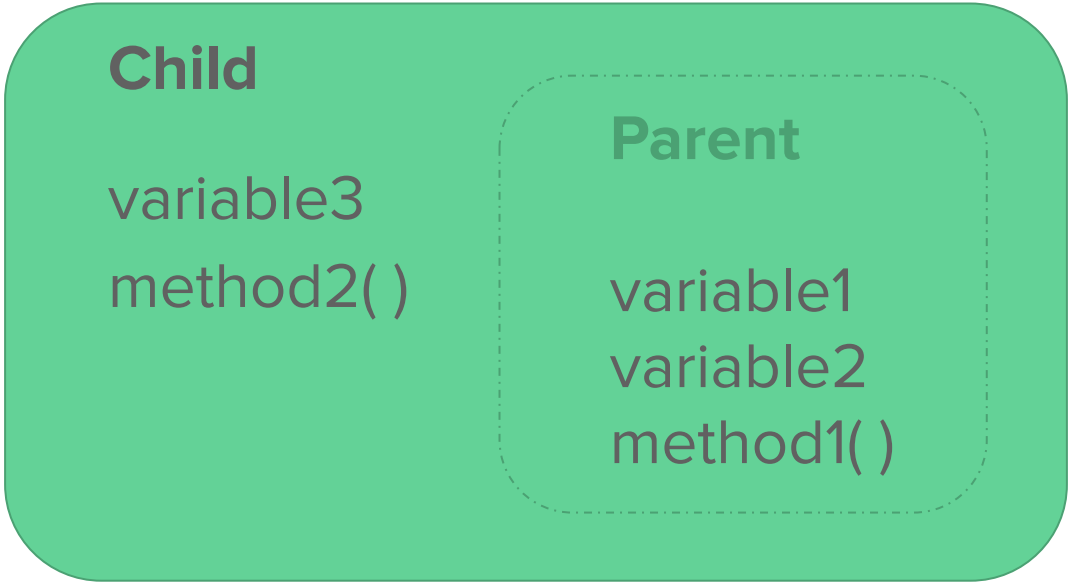
```
public class NewClass extends Object { }
```

This is why every class has toString(), equals(), and other universal methods.

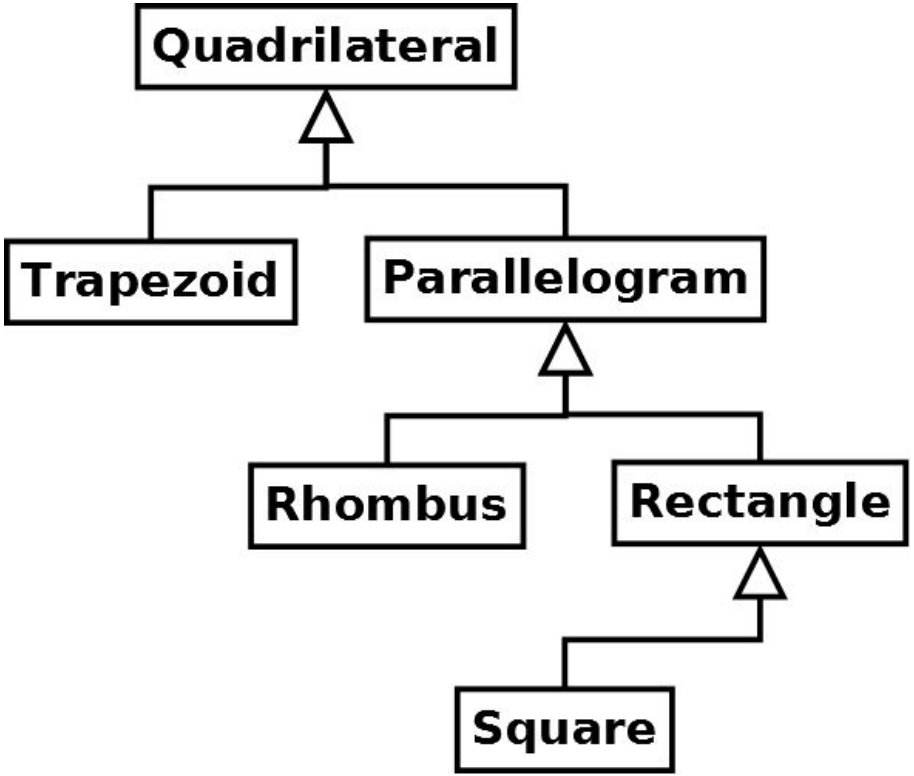
“Is-a” Model of Inheritance Relationships



“Extends” Model of Inheritance Relationships



Inheritance Hierarchies



Encapsulation: public and private Visibility

Inherited **public** data and methods are accessible, by name, in the child class.

private really means private. Although the child does inherit all private data and methods, they are accessible by name only within the class where they were defined: the parent class.

Encapsulation: protected Visibility

If data or methods are declared with **protected** visibility in the parent class, they will be accessible by name in child classes.

While protected preserves encapsulation better than public, protected data and methods will also be accessible by any other classes in the same package as the parent class, as well.

private is still the best protection and accessor and mutator methods inherited from the parent may be sufficient.

Constructors

Constructors are not inherited. Every class must provide its own constructors.

The child can use its parent's constructor code with the **super** keyword in the first line of its own constructor.

This is a very good practice. It allows the class where variables were defined to be responsible for their initialization.

Constructors

```
public class Parent {
    private int variable1;

    public Parent ( int var1 ) {
     variable1 = var1;
    }
}
```

```
public class Child extends Parent {
    private int variable2;

    public Child ( int var1,
                   int var2 ) {
     super ( var1 );
     variable2 = var2;
    }
}
```

Overriding Methods

One of the most common ways to specialize a child class is to override one or more inherited methods - replacing the inherited method.

The inherited method is still present and may be accessed via the **super** keyword within the child class, but it is hidden behind the new version.

Whenever you have written a toString() method in a class, you have been overriding the toString() your class inherited from Object.

Overriding Methods

```
public class Override {

    public static void main ( String[ ] args ) {
     Override obj = new Override( );
     System.out.println ( obj.toString( ) );
    }

    public String toString( ) {
     return "The inherited toString() returns: " + super.toString();
    }
}
```

main() calls the overridden toString(), which internally calls the inherited toString(), resulting in:

```
    The inherited toString() returns: Override@15db9742
```

Inheritance and Interfaces

Interfaces can also inherit from other interfaces.

Unlike classes, interfaces can inherit from multiple parent interfaces, combining all inherited method signatures into one interface. The child interface can then add more methods.

```
public interface NewInterface
    extends OldInterface1, OldInterface2 { }
```

Abstract Classes

An Abstract Class falls between a fully-defined, instantiable, concrete Class and a fully abstract Interface.

Like a class, it can declare variables and methods with bodies.

Like an interface, it can declare abstract methods, without bodies.

Abstract Classes

```
public abstract class AbstractClass {
    private int variable1;    //instance variable inherited by children

    public AbstractClass ( int var1 ) {    //constructor, used via super()
        variable1 = var1;
    }

    public int getVariable1( ) {  //fully-defined method
        return variable1;
    }

    public abstract void doSomething(); //abstract method, must be overridden
}
```

Limiting Inheritance

The **final** keyword can lock down all or part of a class.

Adding **final** to a method signature will prevent it from being overridden in a child class.

Adding **final** to a class heading will prevent the class from being extended. The Math class, for example, is final and cannot be extended.

Note that **final** is incompatible with keyword **abstract**.

Inheritance

Mason Vail
Boise State University Computer Science