

PLEASE COMPLETE THE SOCRATIVE Pulse Survey

URL:

gosocrative.com

ROOM NAME:

JAVAGOLD

Week 3 Overview

Monday

Inheritance

Tuesday

Polymorphism

Wednesday

Managing
Inheritance

Thursday

Unit Testing

Friday

Review

Objectives

- **Review**
- **Inheritance**
 - Constructors
 - Super
 - Method overriding
 - Composition

Review: Classes

- Classes are useful for breaking up a project into a sum of individual parts.
 - Data (variables) for that class are encapsulated as well.
- What kind classes would be in a movie collection application?

Review: Case Scenario 1

- John is a teacher with many courses. John is trying to automate the calculation of each student's final grade in each classroom for each course based on their homework and test scores.

What are some examples of the kind of classes that would be needed?

Review: Case Scenario 2

- Becky works as a restaurant manager and needs a way to take new orders, calculate the total revenue, track food inventory, and store the salary of her current employees.

What are some examples of the kind of classes that would be needed?

Review: Class Identify 1

- There are 5 important things to identify in a class:
 - Instance variables
 - Constructors
 - Getter
 - Setter
 - Methods

```
public class Car {  
    private String color = "green";  
    private int numofDoors = 4;  
    private int fuelRemaining = 5;  
    private int totalFuelCapacity = 10;  
  
    public Car(String color, int numberOfDoors) {  
        this.setColor(color);  
        this.setNumofDoors(numberOfDoors);  
    }  
  
    public void goForward() {  
        System.out.println("going forward");  
    }  
  
    public double fuelRemaining() {  
        return fuelRemaining/totalFuelCapacity * 100.0;  
    }  
  
    public String getColor() {  
        return color;  
    }  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
}
```

Review: Class Identify 2

- There are 5 important things to identify in a class:
 - Instance variables
 - Constructors
 - Getter
 - Setter
 - Methods

```
package com.techelevator;

public class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width){
        this.length = length;
        this.width = width;
    }
    public int getLength() {
        return length;
    }
    public void setLength(int length) {
        this.length = Math.abs(length);
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = Math.abs(width);
    }
    public int getArea() {
        return this.length * this.width;
    }
}
```


Review: Methods

- Methods perform tasks to be reused.
- They can be defined and 'called' (executed)
 - The method definition states what will happen when the method is called.
 - A method is **called by name** with the same **number and type of inputs** in the method definition
 - Saving the **return value** is optional

```
// Calling a method
double tempInC = convertFahrenheitToCelsius(98.6);
convertFahrenheitToCelsius(98.6);

// Method definition
double convertFahrenheitToCelsius(double tempInF) {
    tempInC = (tempInF - 32.0) * (5.0/9.0);
    System.out.println(tempInF + ", " + tempInC);
    return tempInC;
}
```

Parts of a Method

```
returnType name( parameters ) {  
    code...  
    return statement;  
}
```

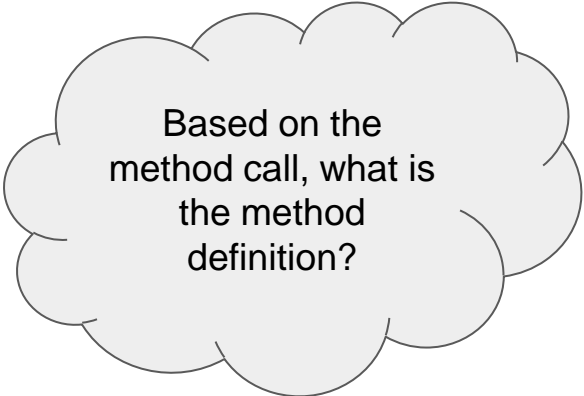
Method Signature	defines the method
Code Block	Code to execute when the method is used. Defined by { } following the method signature.
Return statement	Statement of code that tells the method what value to return. <i>This value must be the same data type as a methods return type.</i>

Example: calling a method by name

```
int sum = addNumbers(42, myIntVariable);
```

Example: method definition

```
int addNumbers(int x, int y) {  
    // code  
}
```



Based on the method call, what is the method definition?

Review: Method Definitions

- Determine the method signature and return type from the following method calls:
- `int zipCode = getZipCodeFromCityState("Cleveland", "Ohio");`
- `String newString = "Hello World!".substring(0, 3);`
- `boolean isWinningBid = placeBid(new Bid(100));`
- `BigDecimal cube = CalculatorClass.cube(3.3);`
- `Bid highestBid = getHighestBid(new Bid[]{ new Bid(100), new Bid(200) });`

Objectives

- Review
- **Inheritance**
 - Constructors
 - Super
 - Method overriding
 - Composition

Object Oriented Programming: Key Ideas

There are three underlying OOP principles:

- ~~Encapsulation~~
- Inheritance
- Polymorphism

By the end of today, you should be able to define **Inheritance**.

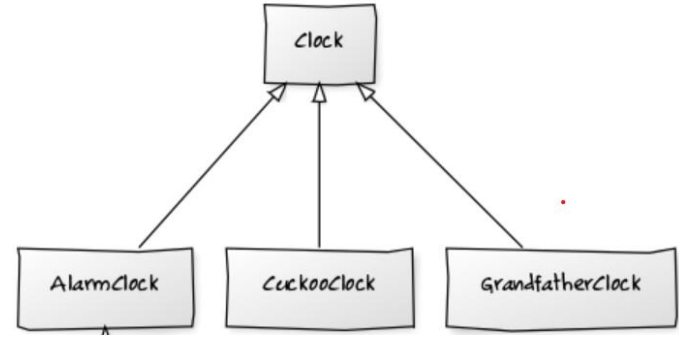
Inheritance

Enables a class to take on the properties and methods defined in another class. A subclass will inherit visible properties and methods from the superclass while adding members of its own.

A **superclass** is the *base class* whose members are being passed down. (parent)

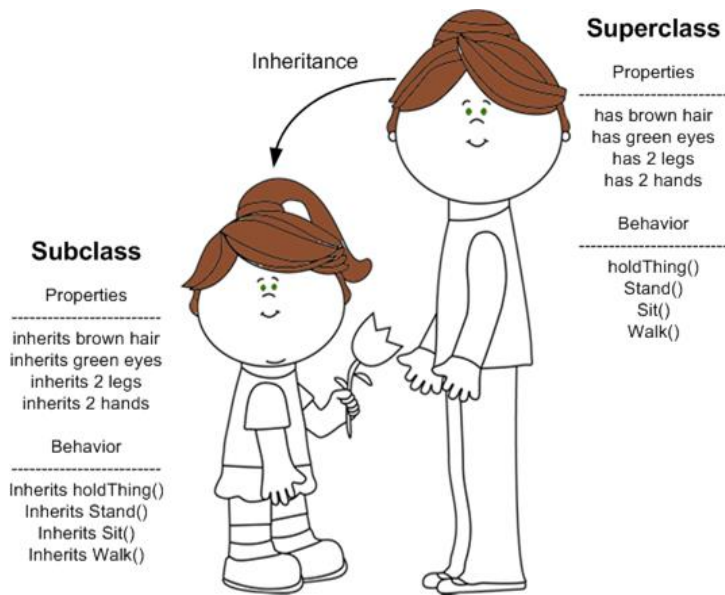
A **subclass** is the derived class acquiring the properties and behaviors from another class.(child)

ALL classes in Java have ONE and ONLY ONE **direct** superclass (one parent), which is called Single Inheritance.



Note: If a class does not have an explicit superclass, then it is implicitly a subclass of `Object` (the start of all classes in Java)

Inheritance



- A subclass inherits all visible (**non-private**) properties and behaviors (methods) from the superclass.
- A subclass DOES NOT inherit **private** properties or methods from the superclass.
- **Constructors** are NOT inherited.
- The subclass will then pass these traits through each subsequent generation, if it becomes a superclass to its own subclasses.
- A Superclass can have multiple subclasses, however, a subclass may only have 1 superclass.

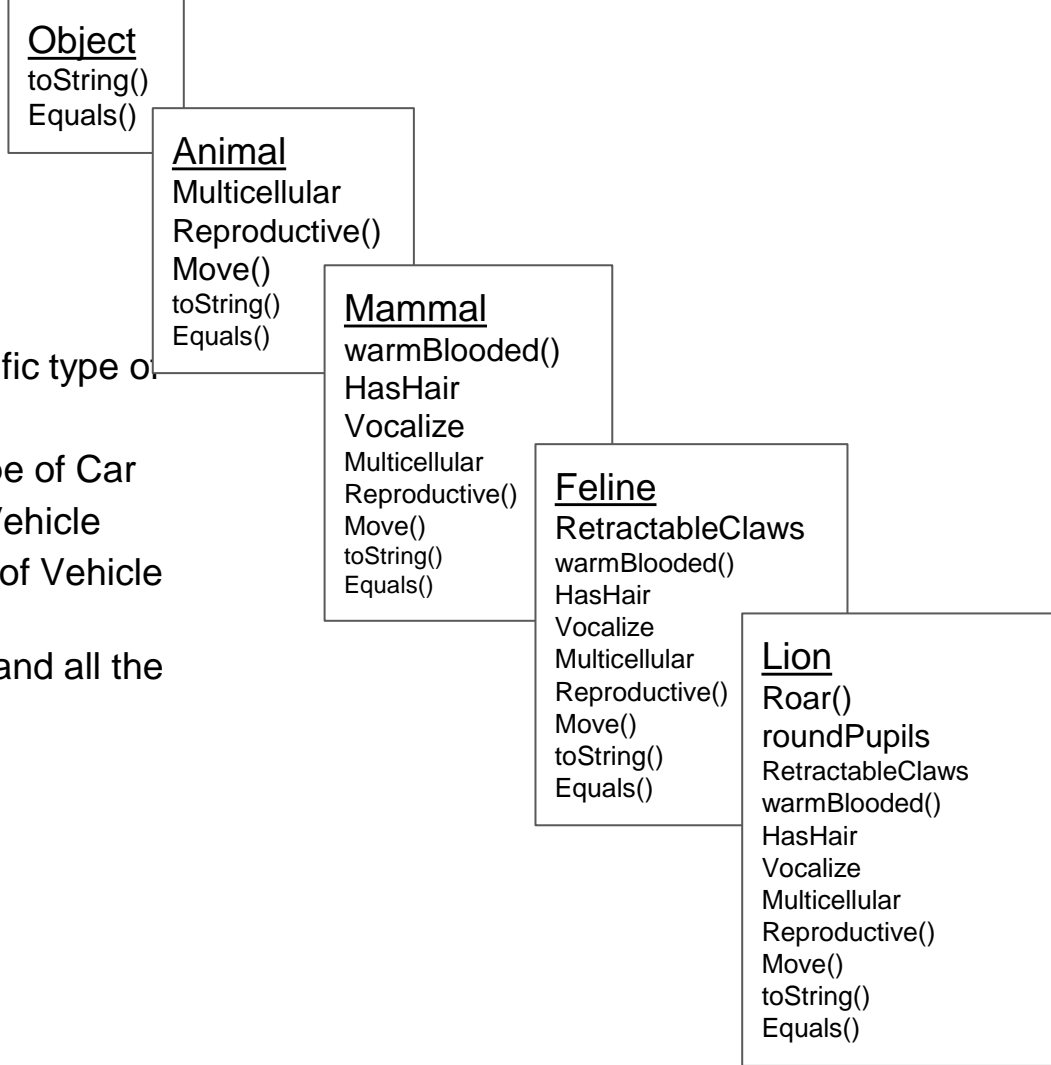
IS-A Relationship

Subclasses are specializations of their base(super) class

- A Graphing Calculator IS-A more specific type of Calculator
- A Honda Accord IS-A more specific type of Car
- A Car is a more specific type of Land Vehicle
- A Land Vehicle is a more specific type of Vehicle

We can say that a subclass IS-A of its superclass and all the superclasses above it.

- Lion IS-A Feline
- Lion IS-A Mammal
- Lion IS-A Animal
- Lion IS-A Object (in Java)



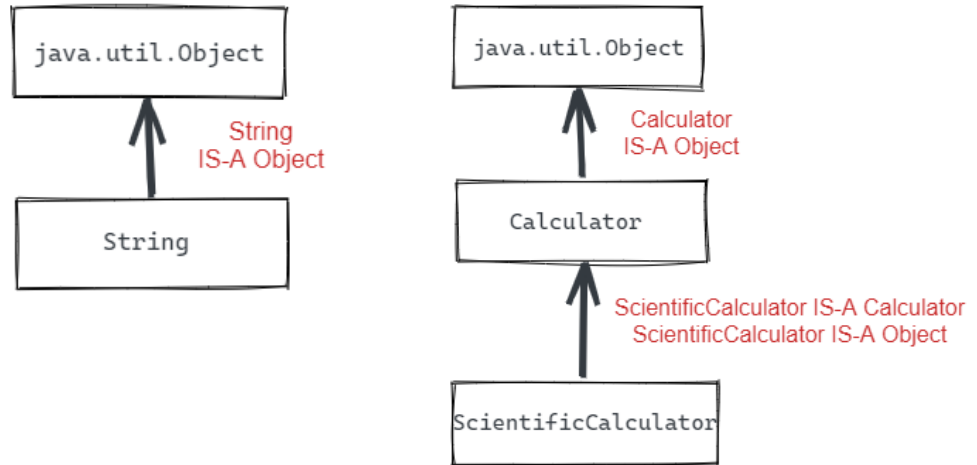
Object

In Java, all Objects (Reference Types) are subclasses of the class **java.lang.Object**. Object is the only class in Java that does not have a superclass.

The only things in the language that are not descendents of java.lang.Object are the primitives: long, int, double, boolean, etc.

Even if no superclass is specified, all classes still *implicitly extend* from java.lang.Object, and inherit a set of common methods, such as:

- .toString()
- .equals()
- .hashCode()



Implementing Inheritance

A class **extends** a superclass

```
public class Car extends Vehicle {  
  
}
```

Once extended, the subclass (**Car**) will inherit all non-private properties and methods from the superclass (**Vehicle**).

Inheritance Example

We use the **super** keyword to refer to the parent's members and variables.

Vehicle has defined several methods and fields. In this example, Vehicle serves as the parent class.

```
package te.mobility;

public class Vehicle {

    private int numberOfWheels;
    private double engineSize;
    private String bodyColor;

    public int getNumberOfWheels() {
        return numberOfWheels;
    }
    public void setNumberOfWheels(int numberOfWheels) {
        this.numberOfWheels = numberOfWheels;
    }
}
```

Car is a child class of Vehicle. Note how it is able to call Vehicle's methods. The extends syntax is used to create the "is-a" relationship.

```
package te.mobility;

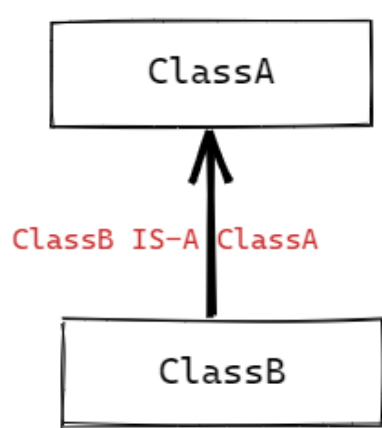
public class Car extends Vehicle {

    public void report() {
        System.out.println(super.getNumberOfWheels());
        // 0, inherited from parent class which will have the
        // default value for integers.

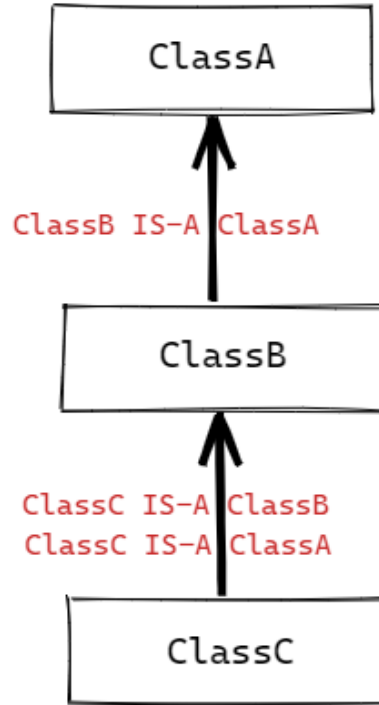
        super.setNumberOfWheels(4);
        // we are calling the setter defined on its parent

        System.out.println(super.getNumberOfWheels());
        // 4
    }
}
```

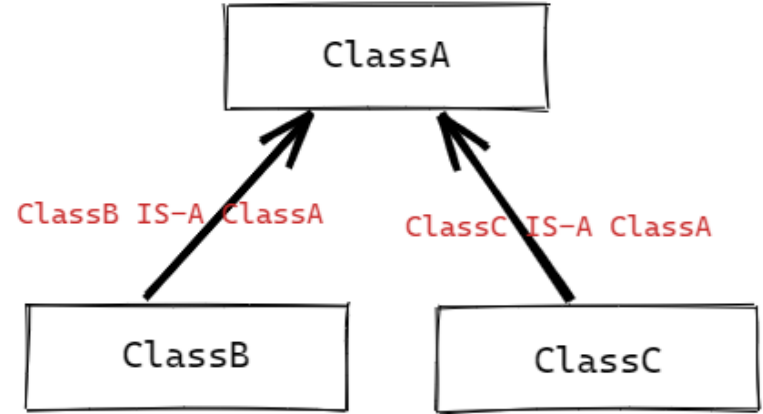
Types of Inheritance in Java



1) Single



2) Multilevel



2) Hierarchical

```
public class Calculator {  
  
    private double total;  
  
    public Calculator(double startingTotal) {  
        this.total = startingTotal;  
    }  
  
    public void add(double amount) {}  
  
    public void subtract(double amount) {}  
  
    public void multiple(double amount) {}  
  
    public void divide(double amount) {}  
  
    public double getTotal() {  
        return this.total;  
    }  
  
}
```

superclass of ScientificCalculator

Inheritance is transitive. All public methods/properties, except the constructor, are passed from superclass to subclass, and to all further subclasses in the hierarchy.

```
public class ScientificCalculator extends Calculator {  
  
    public ScientificCalculator() {  
        super(0);  
    }  
  
    public void addExponent(int exponent) {}  
    public void log(int base) {}  
  
    add()  
    subtract()  
    multiply()  
    divide()  
  
}
```

Inherited from Calculator

subclass of Calculator
superclass of
TrigonometricCalculator

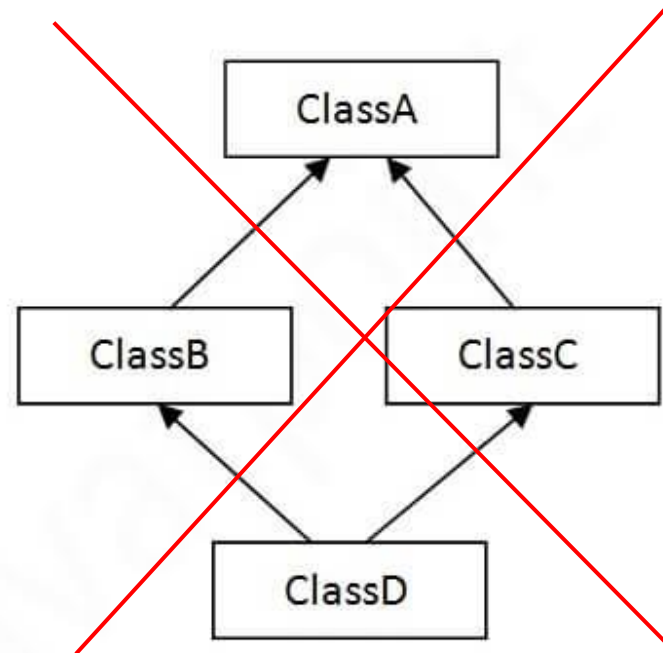
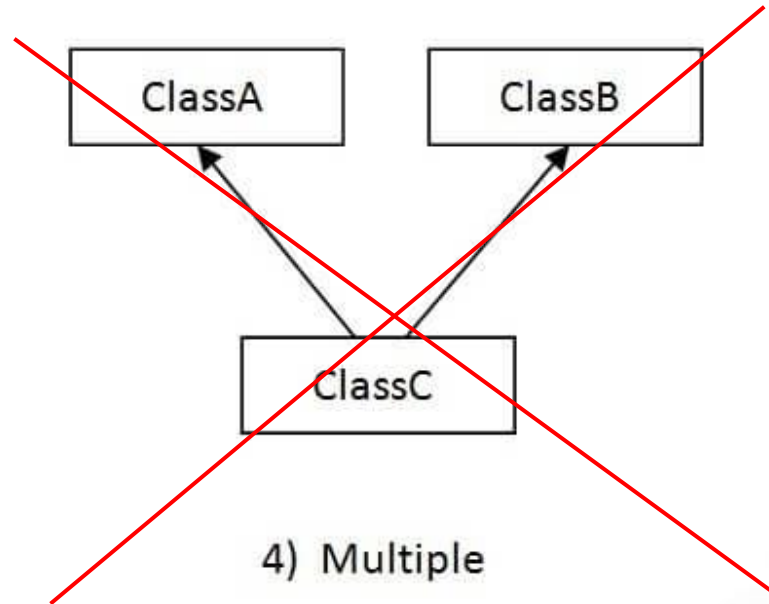
```
public class TrigonometricCalculator extends ScientificCalculator {  
  
    public void sine() {}  
    public void cosine() {}  
    public void tangent() {}  
  
    addExponent()  
    log()  
  
    add()  
    subtract()  
    multiply()  
    divide()  
  
}
```

Inherited from ScientificCalculator

Inherited from Calculator

subclass of ScientificCalculator
ancestor of Calculator

Java Does Not Support Multiple Inheritance



5) Hybrid

Constructors on Parent Classes: Example

We have declared a constructor for Vehicle:

```
package te.mobility;

public class Vehicle {

    private int numberOfWheels;
    private double engineSize;
    private String bodyColor;

    public Vehicle(int numberOfWheels, double engineSize, String bodyColor) {
        this.numberOfWheels = numberOfWheels;
        this.engineSize = engineSize;
        this.bodyColor = bodyColor;
    }
}
```

Constructors on Parent Classes: Example

Note how the child class, Truck will now have to implement a constructor with a `super(...)` call.

```
public class Truck extends Vehicle {  
  
    public Truck(int numberOfWheels, double engineSize, String bodyColor) {  
        super(numberOfWheels, engineSize, bodyColor);  
    }  
    ...  
}
```

```
public class Vehicle {  
    ...  
    public Vehicle(int numberOfWheels, double engineSize, String bodyColor) {  
        this.numberOfWheels = numberOfWheels;  
        this.engineSize = engineSize;  
        this.bodyColor = bodyColor;  
    }  
    ...  
}
```

The `super(...)` call is basically a call to the parent constructor, providing any required parameters

Constructors on Parent Classes: Example

In the Garage orchestrator class note how we are able to instantiate a new Truck with the constructor.

```
package te.main;

import te.mobility.Truck;

public class Garage {

    public static void main(String args[]) {

        Truck cargoTruck = new Truck(10,
14.8, "red");
    }
}
```

Inheritance Example

Here we define another child class of Vehicle called Truck.

```
package te.mobility;

public class Truck extends Vehicle {

    public void report() {
        super.setNumberOfWheels(10);
        // we are calling the setter defined on
    }

    public void coupleCargoContainer() {
        System.out.println("...convoy!");
        super.setNumberOfWheels(18);
    }

}
```

its parent

Let's create another child class of Vehicle, this time Truck.

The Truck class has its own unique method, it has a method called coupleCargoContainer() which is unique to the Truck class, and not part of the Vehicle or Car class.

Inheritance Example

```
package te.main;

import te.mobility.Car;
import te.mobility.Truck;

public class Garage {

    public static void main(String args[]) {

        Car myCar = new Car();

        System.out.println(myCar.getNumberOfWheels());

        Truck myTruck = new Truck();

        // This is an invalid call:
        //myCar.coupleCargoContainer();

    }
}
```

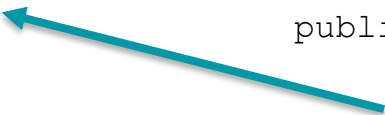
Suppose there is a class called Garage with a main method that will instantiating new cars and trucks..

The highlighted code will not compile since coupleCargoContainer() is unique to the Truck class.

Method Overriding

Inherited methods can be **Overridden** to provide functionality that is specific to the subclass. The **super** keyword can be used in the subclass to invoke the super class's version of an overridden method. To Override a superclass method, a method with an identical method signature is added to the subclass.

```
public class Account {  
    ...  
    public void deposit(int  
        amount) {  
        balance += amount;  
    }  
}  
  
public class CheckingAccount extends Account  
{  
    ...  
    @Override  
    public void deposit(int amount) {  
        amount += depositFee;  
        super.deposit(amount);  
    }  
}
```



Alternatives to Inheritance for Code Reuse

I need to use code from another class... I must use inheritance!
You can also use composition.

```
package te.main;

public class Engine {
    int numberOfCylinders;
    // more code
}
```

```
package te.main;

public class Car {
    Engine engine;

    public Car(){
        this.engine = new Engine();
    }
}
```

```
package te.main;

public class Car {
    Engine engine;

    public Car(Engine engine){
        this.engine = engine;
    }
}
```

- For the example above, the Engine class does not have a 'is-a' relationship with the Car class.
- Engine has a 'has-a' relationship with the Car class.
- The other alternative to inheritance is to create an instance variable of Engine and either:
 - 1. Instantiate the Engine class within the Car class
 - 2. Pass an object of the Engine class into the constructor

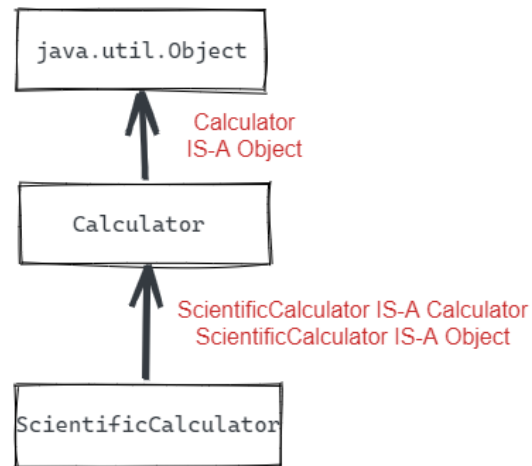
Casting to a superclass (upcasting)

Objects can be cast to any superclass type in their hierarchy. Casting to a superclass is called **Upclassing**.

Upclassing is widening, so it is implicit.

```
ScientificCalculator sc = new ScientificCalculator();  
Calculator c = sc;  
  
Object obj = c;
```

Casting changes the way we view and use the object, but not the object itself. When an object is cast as another object in its hierarchy, then it can be treated as the object it is cast as, and will only have the methods and properties available to that type.



Casting to a subclass (downcasting)

Objects can be cast to any of their subclass types, called **Downcasting**, provided that internally the Object is already that subclass type. Downcasting is narrowing, so must be explicit.

```
ScientificCalculator sc = new ScientificCalculator();  
Calculator c = sc;
```

```
ScientificCalculator backToSc = (ScientificCalculator) c;
```

If the Object is not internally the subclass type it is being cast as, then it will result in a **ClassCastException** runtime error

```
Calculator c = new Calculator();
```

```
ScientificCalculator sc = (ScientificCalculator) c;
```

```
Object obj = new Scanner();
```

```
String s = (String) obj;
```



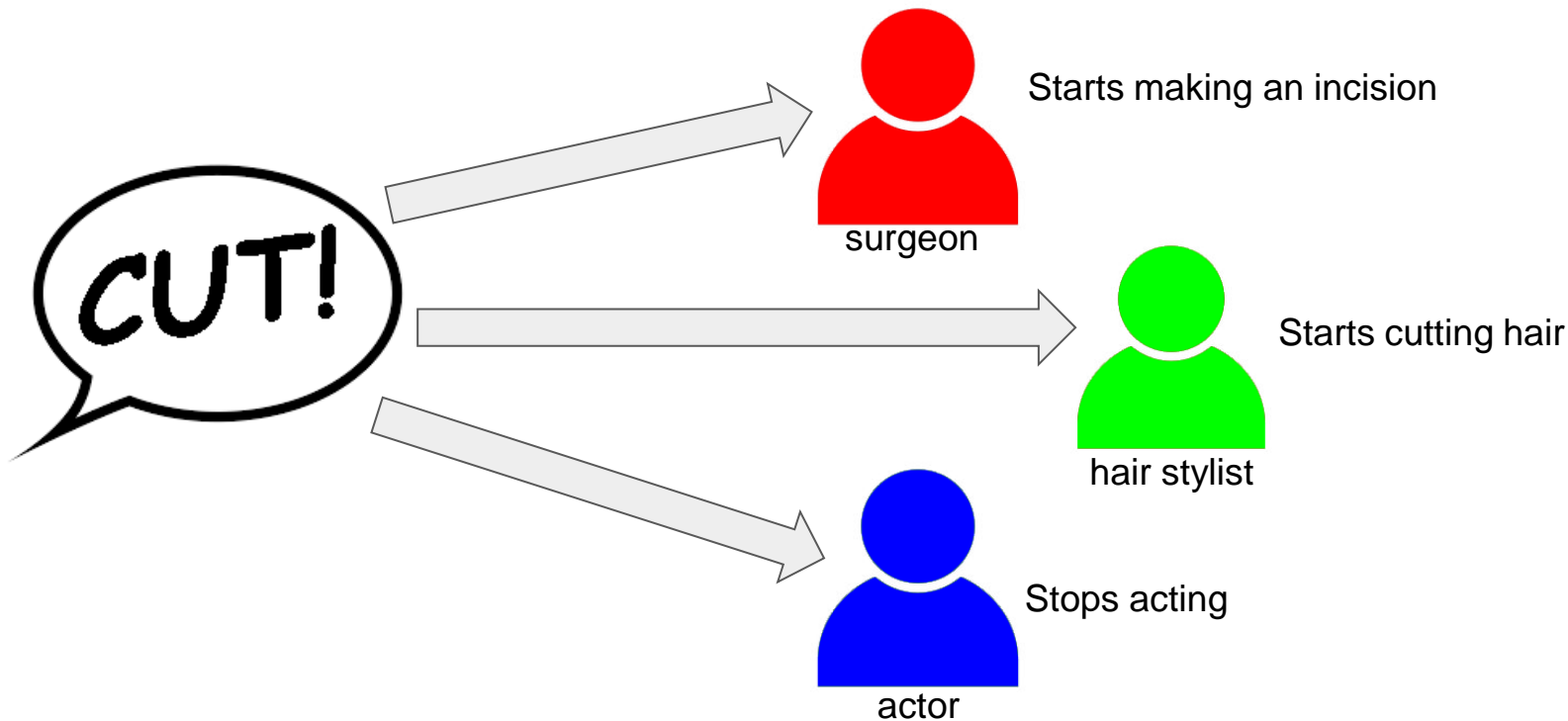
ClassCastException

Summary

- A class can inherit the public and protected variables and methods of another class using the 'extends' keyword.
 - A class can only extend one other class.
 - A subclass is a class that inherits (extends) another class
 - A superclass is a class that is extended.
 - `public class Truck extends Vehicle // Truck = subclass; Vehicle = superclass`
- Inheritance creates an 'is-an' relationship between the subclass and its superclass.
 - `Vehicle truck = new Truck(); // Truck class extends Vehicle class`
- Methods in a superclass can be **Overridden** by its subclass
 - A subclass can call the superclass's method using `super.<method>`
- A subclass's constructor must call the superclass's constructor if one is defined using `super()`

Polymorphism

The ability to treat an object as its superclass (generically) and still get the specific response for the subclass.



Polymorphism

```
graph TD; A[Polymorphism] --> B[Overriding]; A --> C[Overloading];
```

Overriding

Overriding a method of superclass in the subclass by providing a method with the same signature and its own subclass specific behavior.

Occurs at Run time

Overloading

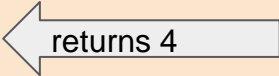
Overloading a method where more than one method have the same name and different arguments.

Occurs at Compile time

```
public class Vehicle {  
  
    public int getNumberOfWheels() {  
        return 0;  
    }  
}  
  
public class Car extends Vehicle {  
  
    @Override  
    public int getNumberOfWheels() {  
        return 4;  
    }  
}  
  
public class Bike extends Vehicle {  
    @Override  
    public int getNumberOfWheels() {  
        return 2;  
    }  
}
```

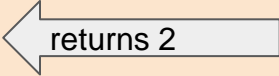
```
Vehicle vehicleOne = new Car();  
Vehicle vehicleTwo = new Bike();
```

```
vehicleOne.getNumberOfWheels();
```



returns 4

```
vehicleTwo.getNumberOfWheels();
```



returns 2

```
List<Vehicle> vehicles = new  
ArrayList<Vehicle>();  
vehicles.add( vehicleOne );  
vehicles.add( vehicleTwo );
```

```
for (Vehicle v : vehicles) {  
    v.getNumberOfWheels();  
}
```



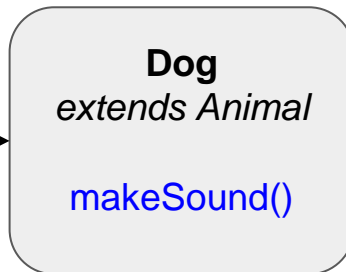
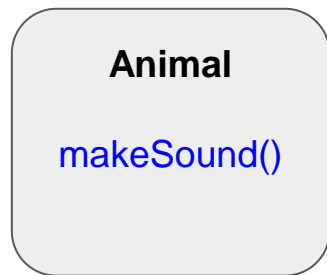
uses subclass Override

Polymorphism with Inheritance

When a *subclass* is **upcast** to its *superclass* the subclass specific *overrides* will still be invoked.

This allows for a subclass to be treated as one of their more generic superclasses and still give responses specific to that subclass.

Polymorphism with Inheritance



Bark

A black speech bubble containing the word "Bark" in bold black text.

Meow

A black speech bubble containing the word "Meow" in bold black text.

Moo

A black speech bubble containing the word "Moo" in bold black text.

BigDecimal

java.Math.BigDecimal

<https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>

Does not have a floating point rounding problem like double and float. Does not truncate like integer. Is commonly used for currency and other calculations that require a high and precise significance of precision.

```
import java.math.BigDecimal;
```

```
BigDecimal amount = new BigDecimal(<value>);    ← CANNOT use a No-Argument Constructor
```

Can't use operators +, -, %, /, *, <, etc. instead use methods

example: amount.add()

BigDecimal is immutable.

```
BigDecimal amountOne = new BigDecimal("100.50");           // Note the quotes
BigDecimal amountTwo = new BigDecimal("200.25");           // Note the quotes
BigDecimal combinedAmount = amountOne.add(amountTwo);
```

In the above code, when add is called the value of amountOne is not changed, it remains 100.50. Instead a new BigDecimal is returned with the sum (300.75). This is due to BigDecimal being immutable, and is the same as when you use a String function like substring() or toUpperCase()