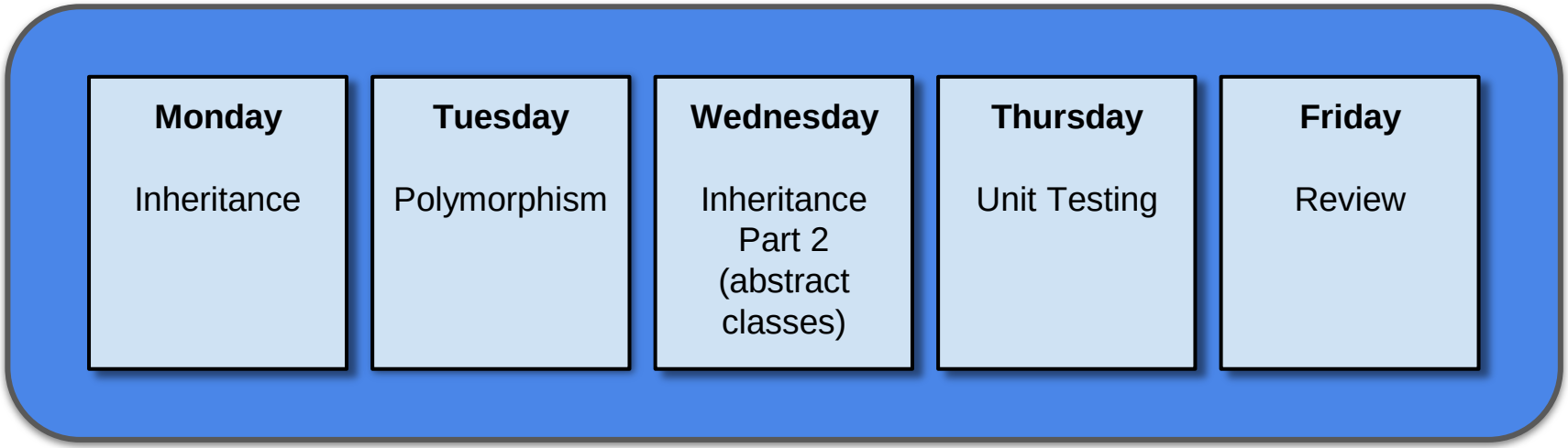
Inheritance

Module 1: 11

Week 3 Overview

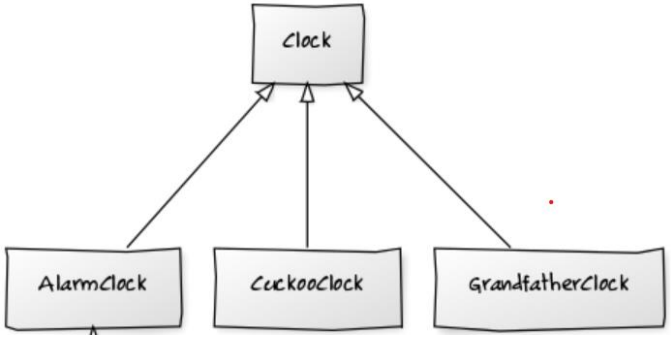


Objectives

1. Defining Inheritance
2. Specialization and is-a
3. Implementing inheritance
4. Polymorphism with inheritance
5. BigDecimal

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.

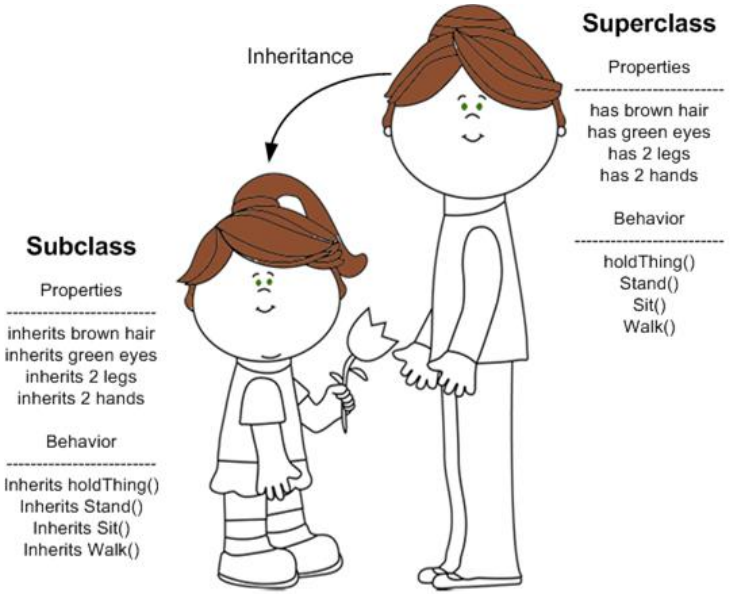


A **superclass** is the *base class* whose members are being passed down. (parent)
A **subclass** is the derived class acquiring the properties and behaviors from another class.(child)

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.

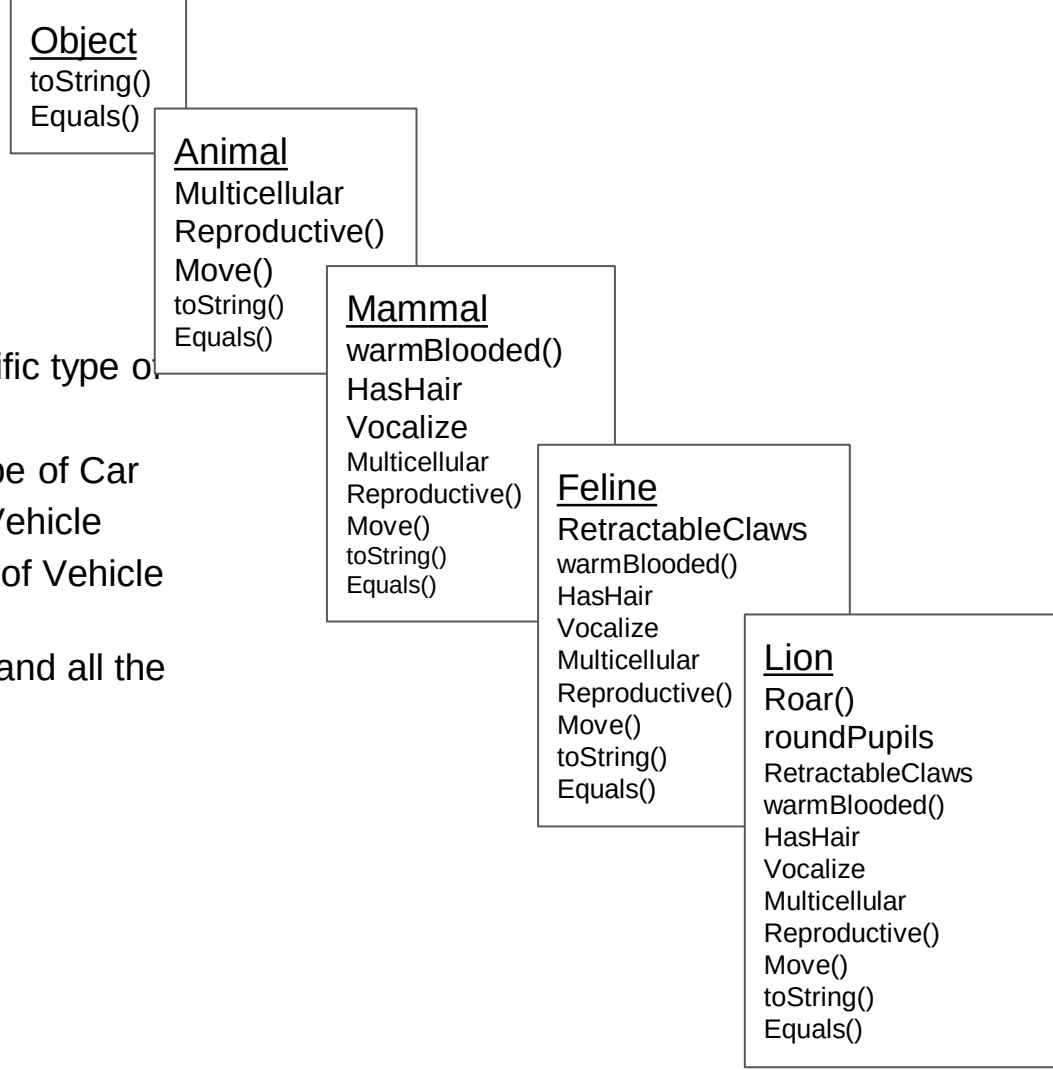
IS-A Relationship

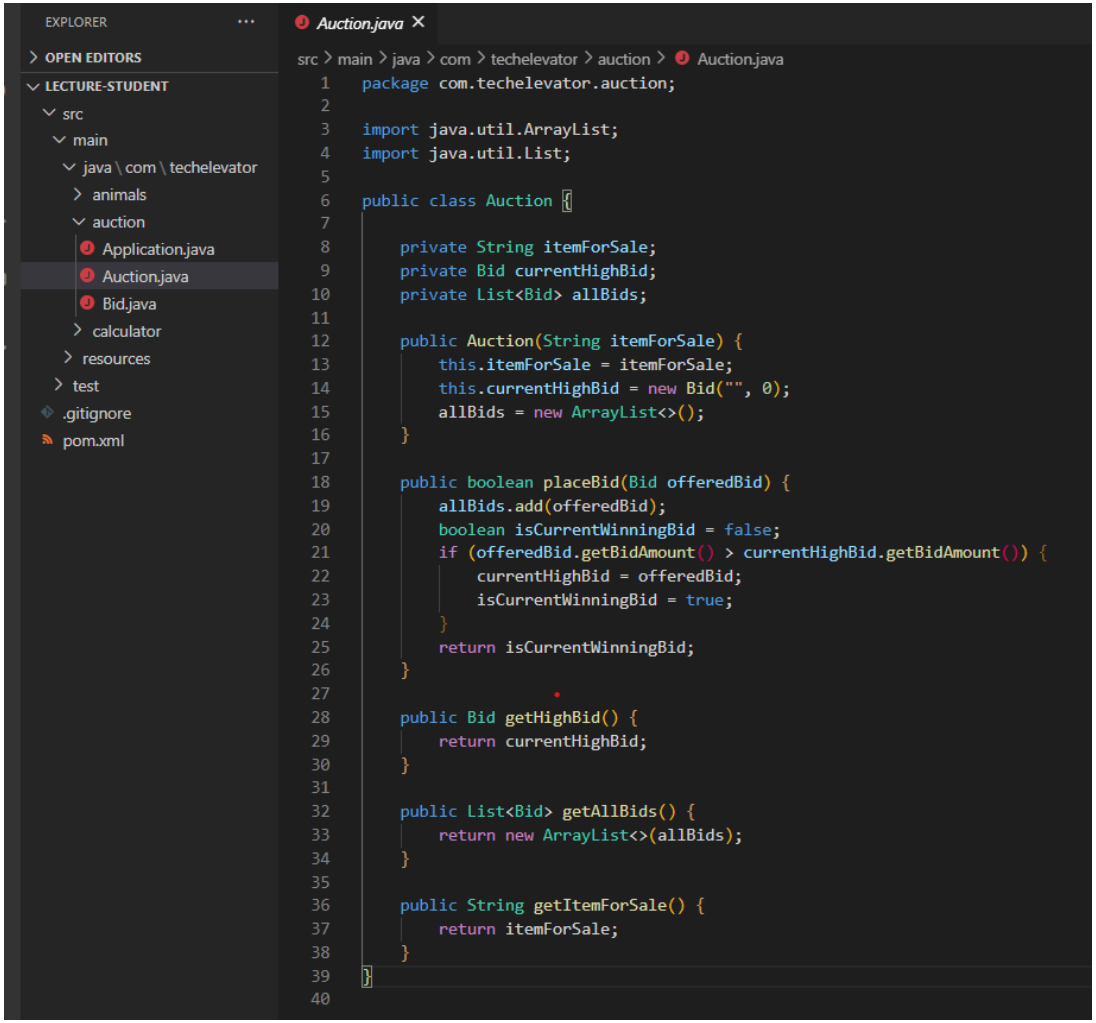
Subclasses are specializations of their base(super) class

- 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)





An auction includes

Private:
ItemForSale
currentHighBid
allBids

Public:
placeBid()
getHighBid()
getAllBids()
getItemForSale()

Implementing Inheritance

A class ***extends*** a superclass

```
public class Feline extends Mammal {

}
```

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

A common naming convention is name a subclass as SubClassNameSuperClassName

```
public class FelineMammal extends Mammal {
```

This is only convention so not required, and in recent years has become less common.

We need an auction in which:

1) The seller has set a minimum sale price in advance (the 'reserve' price)
2) If the final bid does not reach the reserve price the item remains unsold.

Since the reserve price needs set in advance, that sounds like a good job for the Constructor.

Auction

```
public boolean placeBid(Bid offeredBid) {
    allBids.add(offeredBid);
    boolean isCurrentWinningBid = false;
    if (offeredBid.getBidAmount() > currentHighBid.getBidAmount()) {
        currentHighBid = offeredBid;
        isCurrentWinningBid = true;
    }
    return isCurrentWinningBid;
}
```

Reserve auctions need to make sure that when placing a bid:
offeredBid.getBidAmount() > currentHighBid.getBidAmount()
and
offeredBid.getBidAmount() >= reservePrice

Constructors

1. Constructors are not inherited. If the superclass has a constructor with arguments, then the subclass must invoke the superclass's constructor to provide the values.
2. The ***super*** keyword can be used to invoke the superclass's constructor

```
public class Coin {

            private int value;

            public Coin(int value) {
                        this.value =
value;
      }
}
```

```
public class Quarter extends Coin {

      public Quarter() {

                  super(25);

      }

}
```

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 {

     ...

     @Override
     public void deposit(int
     amount) {
          amount += depositFee;
          super.deposit(amount);
     }

}
```

We need an additional auction in which:

1) an additional set price (the 'buyout' price) that any bidder can accept at any time during the auction, thereby immediately ending the auction and winning the item.

1) If no bidder chooses to utilize the buyout option before the end of bidding the, highest bidder wins.

Since the buyout price needs set in advance, that sounds like a good job for the Constructor.

Auction

```
public boolean placeBid(Bid offeredBid) {
    allBids.add(offeredBid);
    boolean isCurrentWinningBid = false;
    if (offeredBid.getBidAmount() > currentHighB
        currentHighBid = offeredBid;
        isCurrentWinningBid = true;
    }
    return isCurrentWinningBid;
}
```

Buyout auctions need to make sure that when placing a bid:
They determine
if the currentHighBid.getBidAmount() < buyoutPrice
And then if
offeredBid.getBidAmount() >= buyoutPrice
reassign offeredBid bid amount
Otherwise if offeredBid.getBidAmount() >= currentHighBid.getBidAmount() **proceed as normal.**

```
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 dou
                return this.total;
        }

}
```

superclass of ScientificCalculator

```
public class ScientificCalculator extends Calculator {

        public ScientificCalculator() {
                super(0);
        }

        public void addExponent(int exponnent) {}
        public void log(int base) {}

        add()
        subtract()
        multiply()
        divide()
}
```

Inherited from Calculator

subclass of Calculator
superclass of
TrignometricCalculator

```
public class TrigonometricCalculator extends ScientificCalculator {

        public void sine() {}
        public void cosine() {}
        public void tangent() {}

        addExponent()
        log()
```

Inherited from ScientificCalculator

```
add()
        subtract()
        multiply()
        divide()
}
```

Inherited from Calculator

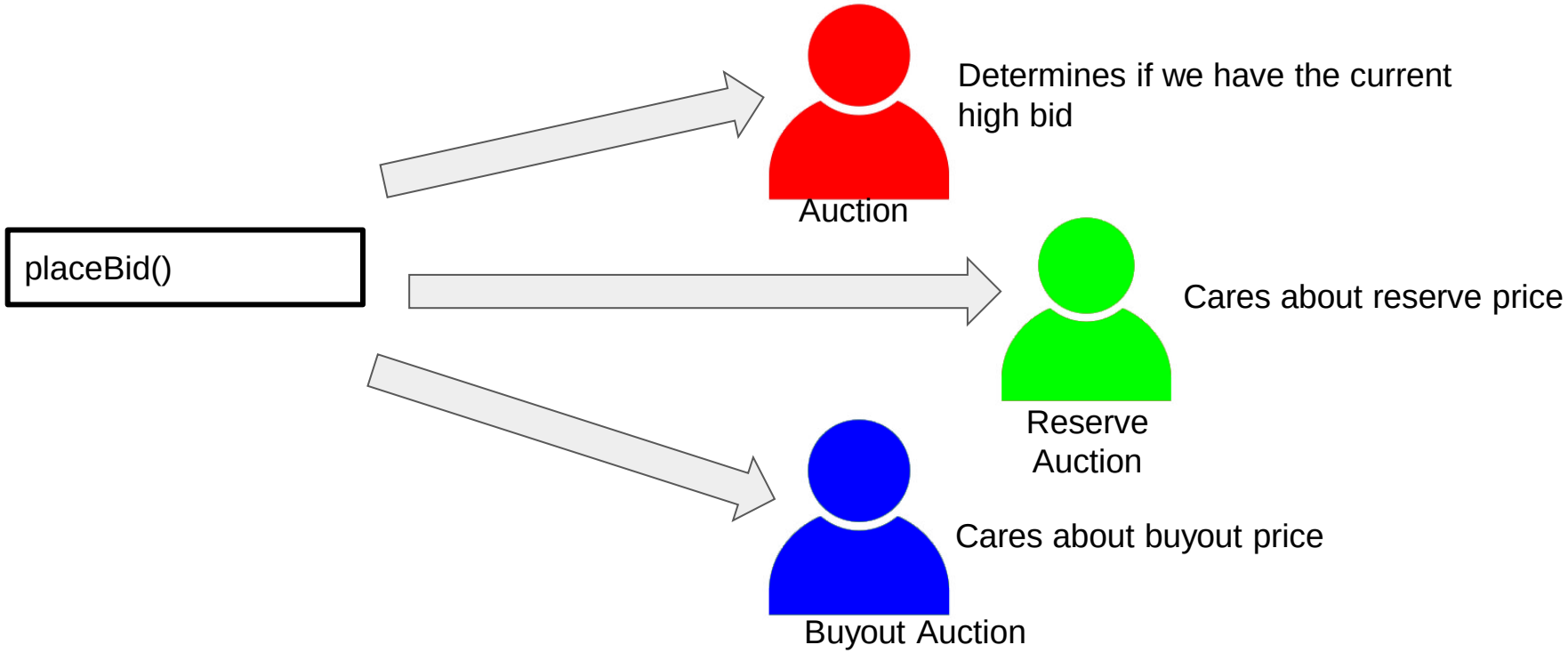
subclass of ScientificCalculator
ancestor of Calculator

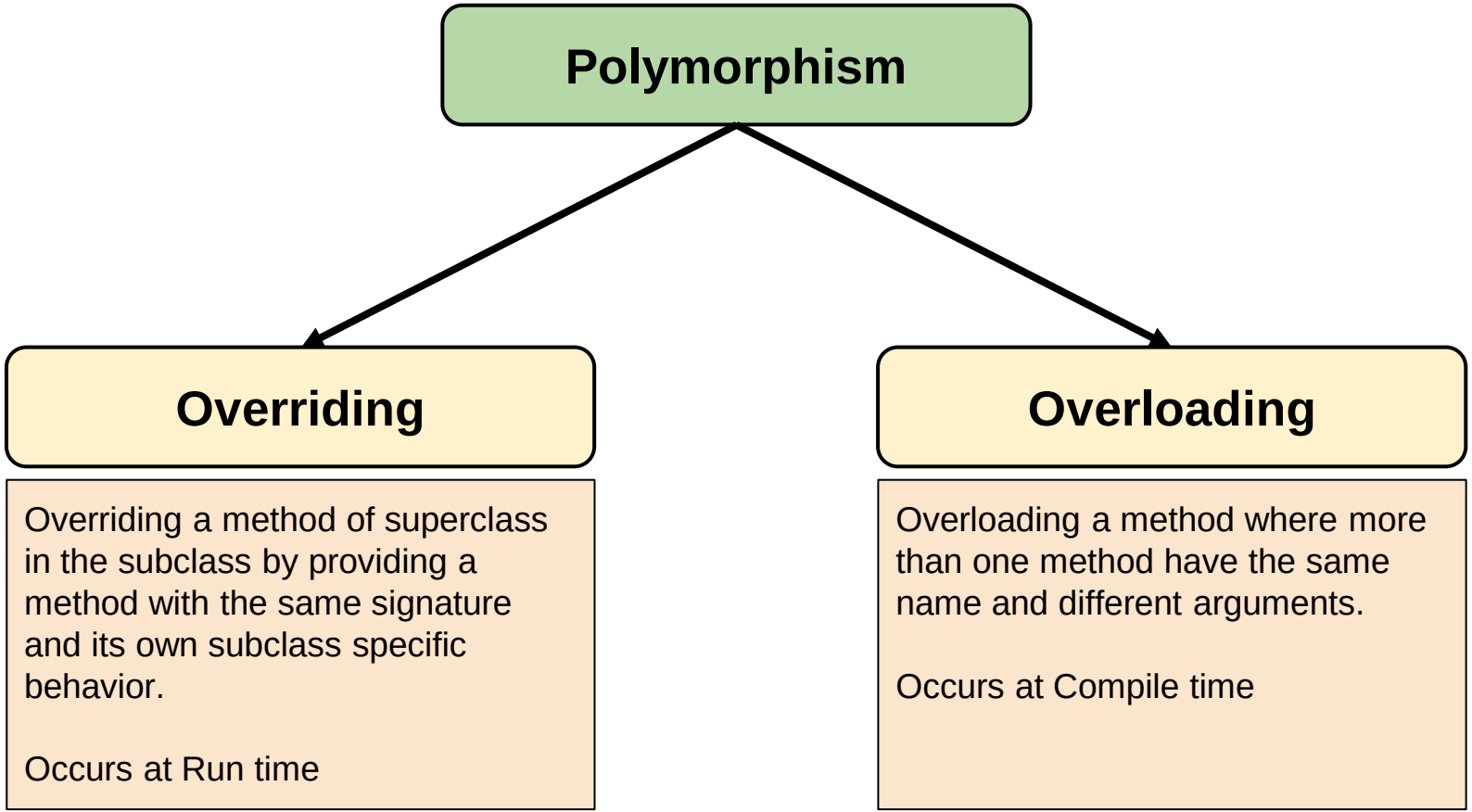
Inheritance is transitive. All public methods/properties, except the constructor, are passed from superclass to subclass, and to all further subclasses in the hierarchy.

Polymorphism with Inheritance

Polymorphism

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




Polymorphism
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

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

Animal

makeSound()

Dog
extends Animal

makeSound()

Cat
extends Animal

makeSound()

Cow
extends Animal

makeSound()

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

returns 4

returns 2

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

for (Vehicle v : vehicles) {

          v.getNumberOfWheel

}
```

uses subclass Override

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()

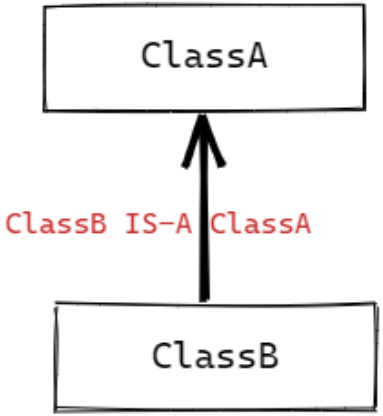
BigDecimal is immutable.

```
BigDecimal amountOne = new BigDecimal(100.50);
BigDecimal amountTwo = new BigDecimal(200.25);
BigDecimal combinedAmount = amountOne.add(amountTwo);
```

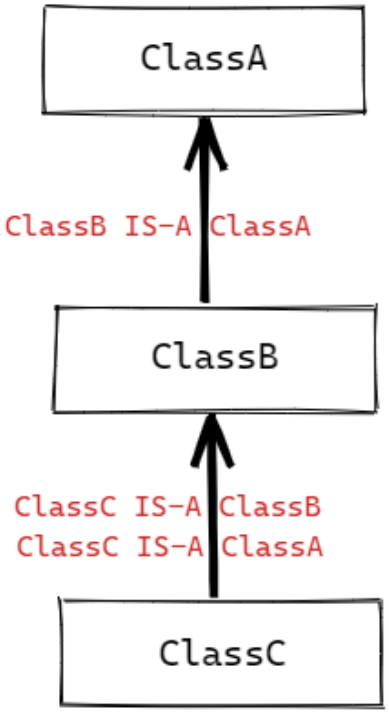
In the above code, when add is called the value of amountOne is not changed, it remains 100.50. Instead a new BigDecimal is returned with the sum (300.75). This is due to BigDecimal being immutable, and is the same as when you use a String function like substring() or toUpperCase()

Bonus slides

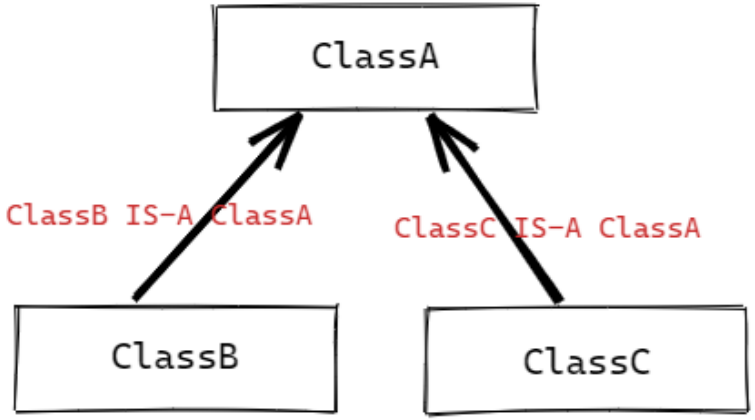
Types of Inheritance in Java



1) Single

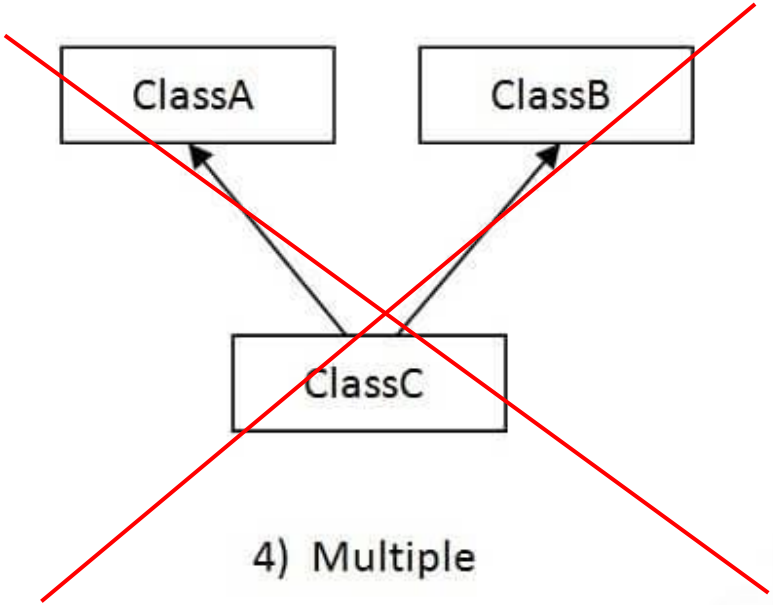


2) Multilevel

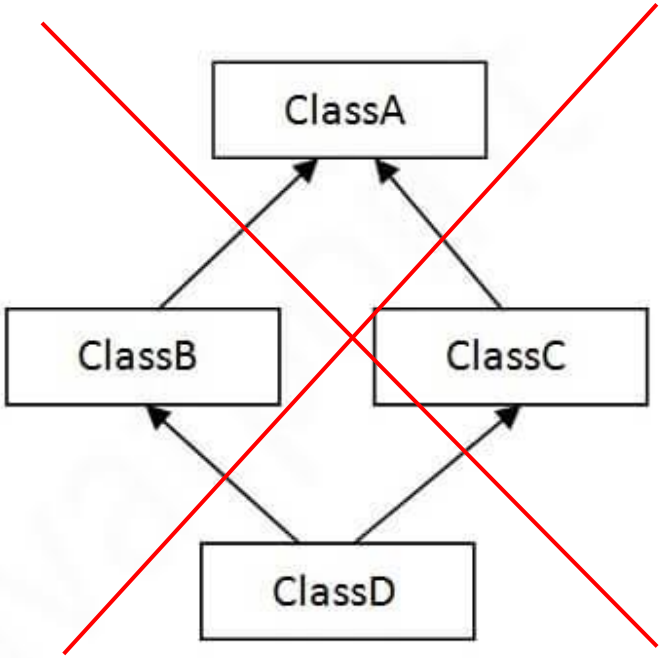


2) Hierarchical

Java Does Not Support Multiple Inheritance



4) Multiple



5) Hybrid

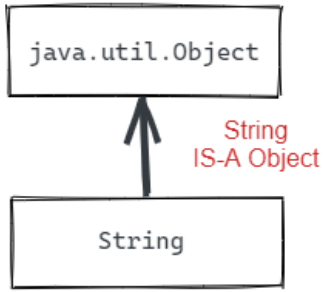
Object

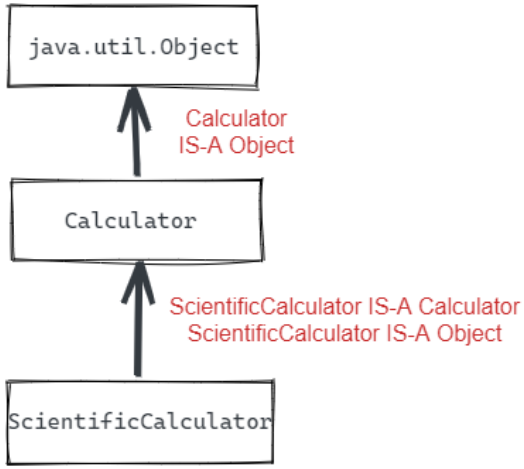
In Java, all Objects (Reference Types) are subclasses of the class **java.lang.Object**. Object is the only class in Java that does not have a superclass.

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()





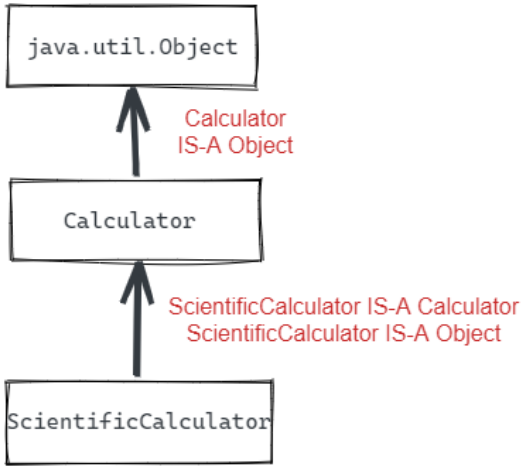
Casting to a superclass (upcasting)

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;
```

Casting

When an object is *downcast* or *upcast* to another class in its hierarchy it will only have access to the properties or methods available on the type it is cast to, and will not have access to any of its own subclass specific methods or properties.

Casting an object to a different type in its hierarchy, only changes how the object is being treated, and does not change the object or what it internally is.

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

In the code above, sc is instantiated as a ScientificCalculator and then upcast to a Calculator and then Object. However, in all cases the object is still internally a ScientificCalculator, even when it is cast and being treated as one of its superclasses.

instanceof

Since *downcasting* can only be done if the object is already internally the type it is being cast to, there is a boolean operator, ***instanceof***, that can check if the object can be downcast to the subclass type.

object instanceof class

```
public void convert(Calculator calculator) {

     if (calculator instanceof ScientificCalculator) {
               ScientificCalculator = (Scientific) calculator
     }

}
```

instanceof should be used when a class is being downcast to a subclass, and it is not known what type the object is internally.

instanceof never needs to be used when upcasting to a superclass, since all subclasses can always be upcast to their superclass.