Inheritance and Polymorphism

Spot the Similarities

- What are the similarities between a convertible and a sedan?
- What are the differences?

Convertibles vs. Sedans

- Convertible
  - Might have only 2 seats
  - Top down
- Sedan
  - 5 seats

Digging deeper into the similarities

- A convertible and a sedan are extremely similar
- Not only do they share a lot of the same capabilities, they perform these actions in the same way
  - Both cars drive and brake the same way
  - Let’s assume they have the same engine, chassis, door, brake pedals, fuel systems, etc.

Can we model this in code?

- In some cases, objects can be very closely related to each other
  - Convertibles and sedans drive the same way
  - Flip phones and smartphones call the same way
  - Brown students and Harvard students study the same way
- Imagine we have an Convertible and a Sedan class
  - Can we enumerate their similarities in one place?
  - How do we portray their relationship through code?

Interfaces

- We could build an interface to model their similarities
  - Build a Car interface with the following methods:
    - turnOnEngine()
    - turnOffEngine()
    - drive()
- Remember: interfaces only declare methods
  - Each class will need to implement the method in its own way
  - Thinking ahead: a lot of these method implementations would be the same across classes
    - Convertible and Sedan would have the same definition for drive()
  - startEngine, shiftToDrive, etc
- Is there a better way where we can reuse the code?
In OOP, inheritance is a way of modeling very similar classes. Inheritance models an “is-a” relationship:

- A sedan is a car
- A poodle is a dog
- A dog is a mammal

Remember: Interfaces model an “acts-as” relationship.

You’ve probably seen inheritance before!

- Taxonomy from biology class
- In biology, any level has all of the guaranteed capabilities of the levels above it but is more specialized
- A dog inherits the capabilities of its “parent,” so it knows what a mammal knows how to do (and more)

This is an inheritance diagram:

- Each box represents a class
- A Poodle “is-a” Dog, a Dog “is-a” Mammal
- Transitively, a Poodle is a Mammal

Inheritance and Interfaces both legislate class’s behavior, although in very different ways:

- Interfaces allow the compiler to enforce method implementation
- Inheritance assumes the compiler that all subclasses of a superclass will have the superclass’s public capabilities without having to separately code – methods are inherited

Benefit of inheritance:

- Code reuse
  - If drive() is defined in Car, Convertible doesn’t need to redefine it (Code is inherited)
- Only need to implement what is different, i.e. what makes Convertible special

Superclass/parent/base: A class that is inherited from

Subclass/child/derived: A class that inherits from another

“A Poodle is a Dog”
- Poodle is the subclass
- Dog is the superclass

A class can be both a superclass and a subclass:
- Ex. Dog

You can only inherit from one superclass:
- Other languages, like C++, allow for multiple inheritance, but too easy to mess up

Object provides very few methods – not particularly useful

A subclass inherits all of its parent’s public capabilities:
- If Car defines drive(), Convertible inherits drive() from Car and drives the same way. This holds true for all of Convertible’s subclasses as well

Inheritance and Interfaces both legislate class’s behavior, although in very different ways:
- Interfaces allow the compiler to enforce method implementation
- An implementing class will have all capabilities outlined in an interface

Inheritance assumes the compiler that all subclasses of a superclass will have the superclass’s public capabilities without having to separately code – methods are inherited
- A Convertible knows how to drive and drives the same way as Car because of inherited code

Benefit of inheritance:

- Code reuse
  - If drive() is defined in Car, Convertible doesn’t need to redefine it (Code is inherited)
  - Only need to implement what is different, i.e. what makes Convertible special

Which of the following is a superclass/parent of the rest?

A. Lions
B. Tigers
C. Cats
D. Leopards
Superclasses vs Subclasses

- A **superclass** factors out commonalities among its **subclasses**
  - describes everything that all subclasses have in common
  - Dog defines things common to all Dogs
- A **subclass** differentiates/specializes its **superclass** by:
  - adding new methods:
    - the subclass should define specialized methods. All Animals cannot swim, but Fish can
  - overriding inherited methods:
    - a Bear class might override its inherited sleep method so that it hibernates rather than sleeping as most other Animals do
  - defining "abstract" methods:
    - the superclass declares but does not define (more on this later!)

Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Let's model a Van, a CS15Mobile (Sedan), and a Convertible class with inheritance!

Modeling Inheritance

- Let's model a Van, a CS15Mobile (Sedan), and a Convertible class with inheritance!

Step 1: Define the superclass

- Defining Car is just like defining any other class

```java
public class Car {
  private Engine _engine;
  //other variables elided
  public Car() {
    _engine = new Engine();
  }
  public void turnOnEngine() {
    _engine.start();
  }
  public void turnOffEngine() {
    _engine.shutOff();
  }
  public void cleanEngine() {
    _engine.steamClean();
  }
  public void drive() {
    //code elided
  }
  //more methods elided
}
```

Step 2: Define a subclass

- Notice the extends keyword
  - extends means "is a subclass of" or "inheriting from"
  - extends lets the compiler know that Convertible is inheriting from Car
  - Whenever you create a class that inherits from a superclass, must include "extends <superclass name>" in class declaration

```java
public class Convertible extends Car {
  //code elided for now
```

Model Inheritance

- You can create any number of subclasses
  - CS15Mobile, Van, Convertible, SUV...could all extend from Car
  - These classes will inherit public capabilities from Car
  - Each subclass can only inherit from one superclass
  - Convertible cannot extend Car, FourWheeledTransportation, and GasFueledTransportation
  - Contrast with interfaces: you can implement as many interfaces as you want
Clicker Question 2
All of the following are appropriate ways to model superclasses and subclasses EXCEPT:

A. C.
B. Clicker Question 2

Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Adding new methods (1/3)

- Our `Convertible` class does more than a generic `Car` class
- Let's add a `putTopDown()` method
  - It's as easy as that!

Adding new methods (2/3)

- Now, let's make a `CS15Mobile` class that inherits from `Car`
- Can `CS15Mobile` use `putTopDown()`?
  - No. That method is defined in `Convertible`, so only `Convertible` and its subclasses can use it

Adding new methods (3/3)

- You can add specialized functionality to a subclass by defining methods
- These methods can only be inherited if a class extends this subclass

What can subclasses access? (1/4)

- Remember: a subclass inherits any `public` methods or `public` instance variables (typically none) from its superclass
- Before adding any code to `Convertible` class, what does `Convertible` already know how to do?
  - It can do anything a `Car` can do!
    - `turnOnEngine()`
    - `turnOffEngine()`
    - `drive()`
What can subclasses access? (2/4)

- **Will** Convertible **have access to** \_engine?
- **Private** variables or **private** methods of a superclass are not **directly inherited** by its subclasses
  - superclass protects them from even its subclasses
- Convertible **cannot directly access** any of Car's **private variables**
- In fact, Convertible is completely unaware that a variable named \_engine exists! **Encapsulation for safety!**

```java
public class Car {
    private Engine _engine;
    // other variables elided
    public Car() {
        _engine = new Engine();
    }
    public void turnOnEngine() {
        _engine.start();
    }
    public void turnOffEngine() {
        _engine.shutOff();
    }
    public void drive();
    // code elided
    // more methods elided
}
```

What can subclasses access? (3/4)

- But that’s not the whole story...
- Every instance of a subclass is also an instance of its superclass — every instance of Convertible is also a Car
- You can’t access \_engine **directly** by Convertible’s specialized methods
- But you can use methods defined in your parent, which have access to the variable

```java
public class Convertible extends Car {
    // constructor elided
    public void cleanCar() {
        _engine.teamClean();
    }
}
```

Let’s examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

What can subclasses access? (4/4)

- What if superclass’s designer wants to allow subclasses access to some of its instance variables?
  - Can give indirect access by defining public accessor and mutator methods for private instance variables
  - We’ll see how to do this later in the lecture...

```java
public class Car {
    private Engine _engine;
    // other variables elided
    public Car() {
        _engine = new Engine();
    }
    public void drive();
    // code elided
    // more methods elided
}
```

Overriding methods (1/4)

- A Convertible **may decide** Car’s drive() method just **doesn’t cut it**
  - A Convertible drives much faster than a regular car
- Can **override a parent class’s method** and **redefine it**

```java
public class Convertible extends Car {
    public Convertible() {
    }
    @Override
    public void drive() {
        this.goSixtyMPH();
    }
    public void goSixtyMPH() {
        // code elided
    }
    // more methods elided
}
```

Overriding methods (2/4)

- @Override should look familiar!
  - Saw it when we implemented an interface method
- We include @Override **right before** we declare method we mean to override
- @Override **is an annotation** — signals to compiler (and to anyone reading your code) that you’re overriding a method of the superclass

```java
public class Convertible extends Car {
    public Convertible() {
    }
    @Override
    public void drive() {
        this.goSixtyMPH();
    }
}
```
Overriding methods (3/4)

- Here's where we re-declare a method we want to override.
- Be careful – method signature (name of method, and list of parameters) must match that of the superclass's method exactly!
  - Or else Java will create a new, additional method instead of overriding.
- `drive()` is the method signature, indicating that name of method is `drive` and it takes in no parameters.

```java
public class Convertible extends Car {
    public Convertible() {
        //code elided
    }

    @Override
    public void drive() {
        this.goSixtyMPH();
    }
}
```

Overriding methods (4/4)

- Fill in body of method with whatever we want a `Convertible` to do when it is told to drive.
- In this case, we're fully overriding the `drive()` method.
- When a `Convertible` is told to drive, it will execute this code instead of the code in its superclass's `drive` method (Java does this automagically - stay tuned).

```java
public class Convertible extends Car {
    public Convertible() {
        //code elided
    }

    @Override
    public void drive() {
        super.drive();
        this.addPinToMap();
    }
}
```

Partially overriding methods (1/6)

- Let's say we want to keep track of `CS15Mobile`'s route.
- `CS15Mobile` drives at the same speed as a `Car`, but it adds dots to a map.

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive() {
        this.goSixtyMPH();
        //code elided
    }
}
```

Partially overriding methods (2/6)

- We need a `CS15Mobile` to start driving normally, and then start adding dots.
- To do this, we partially override the `drive()` method.
  - Partially accept the inheritance relationship.

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive() {
        super.drive();
        this.addPinToMap();
    }
}
```

Partially overriding methods (3/6)

- Just like previous example, use `@Override` to tell compiler we're about to override a method.
- Declare the `drive()` method, making sure that method signature matches that of superclass's `drive` method.

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive() {
        super.drive();
        this.addPinToMap();
    }
}
```

Partially overriding methods (4/6)

- When a `CS15Mobile` drives, it first does what every `Car` does: goes 40mph.
- First thing to do in `CS15Mobile`'s `drive` method is say “drive as if I were just a `Car`, and nothing more.”
- Keyword `super` used to invoke original inherited method from parent: in this case, drive as implemented in parent `Car`.

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive() {
        super.drive();
        this.addPinToMap();
    }
}
```
After doing everything a Car does to drive, the CS15Mobile needs to add a pin to the map!

In this example, the CS15Mobile "partially overrides" the Car's drive method: it drives the way its superclass does, then does something specialized.

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive() {
        super.drive();
        this.addPinToMap();
    }
}
```

If we think our CS15Mobile should move a little more, we can call super.drive() multiple times.

While you can use super to call other methods in the parent class, it's strongly discouraged:
- Use the this keyword instead
- Except when you are calling the parent's method within the child's method of the same name
  - This is partial overriding
  - What would happen if we said this.drive() instead of super.drive()?  

When we call drive() on some instance of Convertible, how does Java know which version of the method to call?

- Starts by looking at the instance's class, regardless of where class is in the inheritance hierarchy
  - If method is defined in the instance's class, Java calls it
  - Otherwise, it checks the superclass
    - If method is explicitly defined in superclass, Java calls it
    - Otherwise, checks the superclass up one level... etc.

Car drive()
Convertible drive()
Porsche drive()

Essentially, Java "walks up the class inheritance tree" from subclass to superclass until it either:
- finds the method, and calls it
- doesn't find the method, and generates a compile-time error.

You can't send a message for which there is no method!

Let's use the car inheritance relationship in an actual program.

Remember the race program from last lecture?

Silly Premise:
- The department received a ~mysterious~ donation and can now afford to give all TAs cars! (we wish)
- Sophia and Dan want to race from their dorms to the CIT in their brand new cars
  - Whoever gets there first, wins!
  - You get to choose which car they get to use
Last lecture’s final design

- Transportation classes that implement the `Transporter` interface
- A `Racer` class that has a `useTransportation(Transporter transport)` method
- A `Race` class that contains the transportation classes and the Racers

A refresher on polymorphism (1/2)

```java
public class Racer {
    //previous code elided
    public void useTransportation(Transporter transport) {
        transport.move();
    }
}

public class Race {
    private Racer_sophia;
    //previous code elided
    public void startRace() {
        _sophia.useTransportation(new Bike());
    }
}
```

A refresher on polymorphism (2/2)

- With last lecture’s example, we used polymorphism to pass in different types of transportation to the `Racer` class
- We chose to see `Bike` as a more generic `Transporter` object
- We can only call methods that `Transporter` declares
  - We sacrifice specificity for generality

Inheritance Example

```java
Transporter bike = new Bike();
Transporter car = new Car();
```

Inheritance and Polymorphism (1/3)

- What is the "lowest common denominator" between `Convertible`, `CS15Mobile`, and `Van`?
  
  ![Diagram](image)

Inheritance and Polymorphism (2/3)

- Can we refer to `CS15Mobile` as its more generic parent, `Car`?
- Declaring `CS15Mobile` as a `Car` follows the same process as declaring a `Bike` as a `Transporter` object
  
  ```java
  Transporter bike = new Bike();
  Car car = new CS15Mobile();
  ```

- `Transporter` and `Car` are the declared types
- `Bike` and `CS15Mobile` are the actual types
Inheritance and Polymorphism (3/3)

- What would happen if we made `Car` the type of the parameter?
  - We can only pass in `Car` and subclasses of `Car`.

```java
public class Racer {
    public void useTransportation(Car myCar) {
        //code elided
    }
}
```

- Is this legal?

```java
Car convertible = new Car();
_sophia.useTransportation(convertible);
Car cs15Mobile = new CS15Mobile();
_sophia.useTransportation(cs15Mobile);
Car bike = new Bike();
_sophia.useTransportation(bike);
```

- Bike is not a subclass of Car, so you cannot treat an instance of Bike as a Car.

--

Inheritance and Polymorphism

- Let’s define `useTransportation()`.
- What method should we call on `myCar`?
  - Every `Car` knows how to drive, which means we can guarantee that every subclass of `Car` also knows how to drive.

```java
public class Racer {
    //previous code elided
    public void useTransportation(Car myCar) {
        myCar.drive();
    }
}
```

- That’s all we needed to do!
- Our inheritance structure looks really similar to our interfaces structure.
  - Therefore, we only need to change 2 lines in `Racer` in order to use any of our new cars!
  - But remember, what’s happening behind the curtain is very different: method resolution “climbs up the hierarchy” for inheritance.
  - Polymorphism is an incredibly powerful tool.
  - Allows for generic programming.
  - Treat multiple classes as their generic type while still allowing specific method implementations to be executed.
  - Polymorphism + Inheritance is strong generic coding.

--

Clicker Question 3

In the following code, the Elephant subclass extends the Animal superclass, both of which contain and define an `eat()` method:

```java
Animal horton = new Elephant();
horton.eat();
```

Whose `eat` method is being called?

A. Animal  
B. Elephant  
C. CS15Mobile  
D. None of the above

--

Indirectly Accessing private Instance Variables

- Remember from earlier that private variables are not directly inherited by subclasses.
- Consider Car’s private instance variable, `_radio`.
- A subclass of `Car`, like Convertible, cannot access _radio by name, has no knowledge of it.

```java
public class Car {
    private Radio _myRadio;
    public Car() {
        _myRadio = new Radio();
    }
    //more code elided
}
```
Defining Accessors and Mutators in Superclass

```java
public class Car {
    private Radio _myRadio;
    public Car() {
        _myRadio = new Radio();
    }
    public Radio getRadio() {
        return _myRadio;
    }
    public void setRadio(Radio radio) {
        _myRadio = radio;
    }
}
```

Calling Accessors/Mutators From Subclass

```java
public class Convertible extends Car {
    public Convertible() {
    }
    public void setRadioPresets() {
        this.getRadio().setFavorite(1, 95.5);
        this.getRadio().setFavorite(2, 92.3);
    }
}
```

Let's step through some code

- Somewhere in our code, a `Convertible` is instantiated.

  ```java
  //somewhere in the program
  Convertible convertible = new Convertible();
  convertible.setRadioPresets();
  ```

  - The next line of code calls `setRadioPresets()`.
  - Let's step into `setRadioPresets()`.

Indirect Inheritance

- Can be thought of as "indirect inheritance".
  - Accessor/mutator methods are the "gateway" through which a subclass can change properties indirectly inherited (pseudo-inherited) from its parent class.

Making Sure Superclass’s Instance Variables are Initialized

- `Convertible` may declare its own instance variables, which it initializes in its constructor.
- `Car`’s instance variables are initialized in the `Car` constructor.
- When we instantiate `Convertible`, how can we make sure `Car`’s instance variables are initialized too?
super(): Invoking Superclass’s Constructor (1/4)

- Car’s instance variables (like _radio) are initialized in Car’s constructor.
- To make sure that _radio is initialized whenever we instantiate a Convertible, we need to call the superclass’s constructor.
- The syntax for doing this is “super().
- Here super() is parent’s constructor; before it was the parent itself (verb vs. noun).

public class Convertible extends Car {
    private ConvertibleTop _top;
    public Convertible() {
        super();
        _top = new ConvertibleTop();
    }
    public void setRadioPresets() {
        this.getRadio().setFavorite(1, 95.5);
        this.getRadio().setFavorite(2, 92.3);
    }
}

What if we don’t call super()?

- What if we don’t call super()
- If you don’t explicitly call super() first thing in your constructor, Java automatically calls it for you, passing in no arguments
- But if superclass’s constructor requires a parameter, you’ll get an error!
- In this case, we get a compiler error stating that there is no constructor “public Car()”, since it was declared with a parameter.

super(): Invoking Superclass’s Constructor (2/4)

- We call super() from the subclass’s constructor to make sure the superclass’s instance variables are initialized properly
- Even though we aren’t instantiating an instance of the superclass, we need to construct the superclass to initialize its instance variables
- Can only make this call once, and it must be the very first line in the subclass’s constructor.

public class Car {
    private Racer _driver;
    public Car(Racer driver) {
        _driver = driver;
    }
    public Racer getRacer() {
        return _driver;
    }
}

Constructor Parameters

- Does CS15Mobile need to have the same number of parameters as Car?
- Nope!
  - As long as Car’s parameters are among the passed parameters, CS15Mobile’s constructor can take in anything else it wants.
  - Let’s modify all the subclasses of Car to take in a number of Passengers.
### Constructor Parameters

```java
public class Passenger extends Car {
  private Passenger _p1;
  public convertible(Racer driver, Passenger p) {
    super(driver);
    _p1 = p;
  }
  // code with passengers elided
}
```

- Notice how we only need to pass driver to super()
- We can add additional parameters in the constructor that only the subclasses will use

```java
public class CS15Mobile(Racer driver, Passenger p1, Passenger p2, Passenger p3, Passenger p4) {
  _p1 = p1;
  _p2 = p2;
  _p3 = p3;
  _p4 = p4;
}
```

### abstract Methods and Classes (1/6)
- What if we wanted to seat all of the passengers in the car?
- CS15Mobile, Convertible, and Van all have different numbers of seats
  - They will all have different implementations of the same method

```java
public abstract class Car {
  private Racer _driver;
  public Car(Racer driver) {
    _driver = driver;
  }
  public abstract void loadPassengers();
}
```

### abstract Methods and Classes (2/6)
- We declare a method `abstract` in a superclass when the subclasses can’t really re-use any implementation the superclass might provide
- In this case, we know that all `Car` should `loadPassengers`, but each `subclass` will `loadPassengers` very differently
- `abstract` method is declared in superclass, but not defined—up to subclasses farther down hierarchy to provide their own implementations

```java
public class Convertible(Racer driver, Passenger p1) {
  public void sit(Seat seat) {
    // more code elided
  }
}
```

### abstract Methods and Classes (3/6)
- Here, we’ve modified `Car` to make it an `abstract` class: a class with at least one abstract method
- We declare both `Car` and its drive method `abstract`: if one of a class’s methods is abstract, the class itself must also be declared abstract
- An abstract method is only declared by the superclass, not implemented — use `semicolon` after declaration instead of curly braces

```java
public abstract class Car {
  private Racer _driver;
  public Car(Racer driver) {
    _driver = driver;
  }
  public abstract void loadPassengers();
}
```

### abstract Methods and Classes (4/6)
- How do you load passengers?
  - Every Passenger must be told to sit in a specific Seat. In a physical Car,
  - SeatGenerator has methods that returns a Seat in a specific logical position

```java
public class Passenger {
  public Passenger() { // code elided }
  public void sit(Seat seat) { // code elided }
}
```

### abstract Methods and Classes (5/6)
- All concrete subclasses of `Car` override by providing a concrete implementation for `Car`’s abstract `loadPassengers()` method
- As usual, method signature must match the one that `Car` declared

```java
public class Convertible extends Car {
  public void loadPassengers() {
    // method implementation
  }
}
```

```java
public class CS15Mobile extends Car {
  public void loadPassengers() {
    // method implementation
  }
}
```

### abstract Methods and Classes (6/6)
- Van all have different numbers of seats
  - They will all have different implementations of the same method
abstract Methods and Classes (6/6)

- **abstract classes cannot be instantiated!**
  - This makes sense – shouldn’t be able to just instantiate a generic `Car`, since it has no code to `loadPassengers()`.
  - Instead, provide implementation of `loadPassengers()` in concrete `subclass`, and instantiate `subclass`.
- **Subclass at any level in inheritance hierarchy can make abstract method concrete by providing implementation**
  - It’s common to have multiple consecutive levels of abstract classes before reaching a concrete class.
- **Even though an abstract class can’t be instantiated, its constructor must still be invoked via `super()` by a `subclass`**
  - Because only the superclass knows about (and therefore only it can initialize) its own instance variables.

So.. What’s the difference?

- You might be wondering: what’s the difference between abstract classes and interfaces?
  - **abstract Classes**:
    - Can define instance variables
    - Can define a mix of concrete and abstract methods
    - You can only inherit from one class
  - **Interfaces**:
    - Cannot define any instance variables/concrete methods
    - You can implement multiple interfaces

Note: Java, like most programming languages, is evolving. In Java 8, interfaces and abstract classes are even closer in that you can have concrete methods in interfaces. We will not make use of this in CS15.

Summary

- Inheritance models very similar classes
  - Factor out all similar capabilities into a generic superclass
  - Superclasses can
    - Declare and define methods
    - Declare abstract methods
  - Subclasses can
    - Inherit methods from a superclass
    - Define their own specialized methods
    - Completely/partially override an inherited method
- Polymorphism allows programmers to reference instances of a subclass as their superclass.
- Inheritance, Interfaces, and Polymorphism take generic programming to the max – more in later lecture
  - Will use polymorphic inheritance in TA SafeHouse and polymorphic interfaces in lab.

Announcements

- LiteBrite early deadline is **tonight** (9/27) at 11:59pm!
  - Hand in early for a 4% extra credit boost
- On-time deadline for LiteBrite is 9/29 (Thursday) at 11:59pm, late deadline is 10/1 at 10pm
- Sign up for a Design Discussion!
  - Keep an eye out for an email about these
- Department is working on fixing issues with FastX
  - Work over SSH or in the Sunlab if you have FastX problems

LiteBrite early deadline is **tonight** (9/27) at 11:59pm!