TopHat Question: Piazza

Are you reading Piazza posts and posting questions to Piazza?
A. Reading
B. Posting
C. Reading and Posting
D. Not using

TopHat Question: TA Hours

Are you attending TA Hours?
A. Usually (once to multiple times a week)
B. Sometimes (twice a month)
C. Not Yet

TopHat Question: Conceptual Hours

Are you attending Conceptual Hours?
A. Usually (once to multiple times a week)
B. Sometimes (twice a month)
C. Not Yet
Top Hat Question: Lectures
In general, how is the pace of in-class lectures?
A. Moves too slowly
B. About right
C. Moves too quickly

TopHat Question: Lecture Capture
Are you reviewing the lecture captures on the website?
A. Usually (once to multiple times a week)
B. Sometimes (twice a month)
C. Not Yet

Top Hat Question: Interfaces
How comfortable are you with Interfaces?
A. Comfortable
B. Somewhat comfortable
C. Not comfortable
Top Hat Question: Polymorphism

How comfortable are you with polymorphism?

A. Comfortable
B. Somewhat comfortable
C. Not comfortable

Top Hat Question: Declared vs Actual Type

How clear is the distinction between Declared vs Actual type to you?

A. Comfortable
B. Somewhat comfortable
C. Not comfortable

Recall: Lucy and Angel’s Race

- Lucy and Angel are racing from their dorms to the CIT
  - whoever gets there first, wins!
  - catch: they don’t get to choose their method of transportation

- Design a program that
  - assigns mode of transportation to each racer
  - starts the race

- Initial design: racer classes that will tell Lucy and Angel to use their type of transportation
  - CarRacer has a useCar() method
  - BikeRacer has a useBike() method
Things to think about

- Did we need two different Racer classes?
  - Could we tell our Racer to use different modes of transportation?
  - Solution 1: a method in Racer for every form of transportation (useCar, useBike, etc.)
    - Inefficient: we would have to create a new method for future classes made
  - Solution 2: use an abstract blueprint to model similar behavior
    - This is where Interfaces and Polymorphism comes in
      - Interfaces group similar capabilities/function of different classes together
      - Model "acts-as" relationship

Recall: Interfaces

- Interfaces are contracts that classes agree to
- If classes choose to implement given interface, it must define all methods declared in interface
  - If classes don’t implement one of interface’s methods, the compiler raises errors
- Interfaces don’t define their methods – classes that implement them do
  - Interfaces only care about the fact that the methods get defined - not how – implementation-agnostic
  - General tip: methods that interface declares should model functionality all implementing classes share

Recall: What does an Interface look like?

```java
public interface Transporter {
    public void move();
}
```
Recall: Using an Interface

```java
public class Car implements Transporter {
    public Car() {
        // constructor
    }
    public void drive() {
        // code for driving car
    }
    @Override
    public void move() {
        this.drive();
    }
}
```

Recall: Polymorphism

- A way of coding *generically*
  - way of referencing many related objects as one generic type
  - cars and bikes can both move() → refer to them as Transporter objects
- How do we write one generic `useTransportation(…)` method?
  ```java
  public class Racer {
      // previous code elided
      public void useTransportation(Transporter transportation) {
          transportation.move();
      }
  }
  ```

Recall: Actual vs. Declared Type

- Can treat Car/Bike objects as Transporter objects
- Car is the actual type
  - Java compiler will look in this class for the definition of any method called on transportation
- Transporter is the declared type
  - compiler will limit any caller so it can only call methods on instances that are declared as Transporter objects AND are defined in that interface
- If Car defines `playRadio()` method, is this correct?
  ```java
  Transporter transportation = new Car();
  transportation.playRadio();
  ```

  Note: The `playRadio()` method is not declared in Transporter Interface. Therefore, compiler does not recognize it as a valid method call.
Method Resolution Uses Actual Type

TopHat Question

How does it work?
Motivations for Polymorphism

- Many different kinds of transportation but only care about their shared capability
  - i.e., how they move
- Polymorphism let programmers sacrifice specificity for generality
  - treat any number of classes as their most general form
    
    ```java
    Transporter angelsCar = new Car();
    ```
  - can only use methods declared in generic form
    - i.e. angelsCar can only call `move()
- You get to decide when that sacrifice is ok!

Why use interfaces?

- Contractual enforcement
  - will guarantee that class has certain capabilities
    
    ```java
    Car implements Transporter, therefore it must know how to move()
    ```

- Polymorphism
  - can have implementation-agnostic classes and methods
    - know that these capability exists, don't care how they're implemented
    - allows for more generic programming
      - can easily extend this program to use any form of transportation, with minimal changes to existing code
    - an extremely powerful tool for extensible programming
  - use when we only care about the most basic functionality
  - if you want to use functionality specialized to a class DO NOT USE

Why is this important?

- Without Polymorphism: more classes → more specialized methods
  - useRollerblades(), useBike(), etc)
- With Polymorphism:
  - as long as the new classes implement `Transporter`, `Racer` doesn't care what transportation it has been given
  - don't need to change `Racer`!
    - less work for you!
    - just add more transportation classes that implement `Transporter`
  - works because Java relaxes its strict type checking
- Are interfaces the only way to use Polymorphism?
Lecture 6
Inheritance and Polymorphism

Outline
- Inheritance
- Overriding Methods
- Indirect Inheritance
- Abstract Classes

Similarities? Differences?
- 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/up

**Sedan**
- 5 seats
- Drive
- Brake
- Play radio
- Lock/unlock doors
- Turn off/on engine

### 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 many 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 a **Convertible** and a **Sedan** class
  - can we put their similarities in one place?
  - how do we portray that relationship with code?

<table>
<thead>
<tr>
<th>Convertible</th>
<th>Sedan</th>
</tr>
</thead>
<tbody>
<tr>
<td>putTopDown()</td>
<td>parkInCompactSpace()</td>
</tr>
<tr>
<td>putTopUp()</td>
<td>turnOnEngine()</td>
</tr>
<tr>
<td>turnOffEngine()</td>
<td>turnOffEngine()</td>
</tr>
<tr>
<td>drive()</td>
<td>drive()</td>
</tr>
</tbody>
</table>
We could build an interface to model their similarities by building a `Car` interface with the following methods:

- `turnOnEngine()`
- `turnOffEngine()`
- `drive()`

Remember: interfaces only "declare" methods. Each class that extends `Car` will need to "implement" `Car`'s methods. A lot of these method implementations would be the same across classes. For example, `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 `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)
- We will cover exactly what is inherited in Java class hierarchy shortly.

Let's examine inheritance further:

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods
4. Accessing instance variables
Modeling Inheritance (1/3)

- 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
- "Inherits from" = "is-a"
  - Poodle inherits from Dog
  - Dog inherits from Mammal
    - for simplicity, we're simplifying the taxonomy here a bit
- This relationship is not bidirectional
  - a Poodle is a Dog, but not every Dog is a Poodle (could be a Labrador, a German Shepherd, etc.)

Modeling Inheritance (2/3)

- **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

Modeling Inheritance (3/3)

- **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
  - e.g., Dog
  - You can only inherit from one superclass
    - no Labradoodles as it would inherit from Poodle and Labrador
  - other languages, like C++, allow for multiple inheritance, but too easy to mess up
TopHat Question 1

Which of the following would be a superclass of the rest?

A. Cat  
B. Panda  
C. Mammal  
D. Dog  
E. None of the Above

Motivations for Inheritance

- A subclass inherits all of its parent's public capabilities
  - if Car defines drive(), Convertible inherits drive() from Car and drives the same way, using Car’s code. This holds true for all of Convertible’s subclasses as well
- Inheritance and interfaces both legislate class’ behavior, although in very different ways
  - an implementing class must specify all capabilities outlined in an interface
  - inheritance assures that all subclasses of a superclass will have the superclass’ public capabilities automatically – no need to re-specify
    - a Convertible knows how to drive and drives the same way as Car because of inherited code

Benefits 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 – do this by adding methods (or modifying inherited methods)
### 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!)

### Modeling Inheritance Example (1/3)

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

```
Car
  ↓
Van  CS15Mobile  Convertible
```

### Modeling Inheritance Reminders

- You can create any number of subclasses
  - *CS15Mobile*, *Van*, *Convertible*, *SUV*...could all inherit from *Car*
  - These classes will inherit public capabilities (i.e., code) from *Car*
- Each subclass can only inherit from one superclass
  - *Convertible* cannot extend *Car*, *FourWheeledTransportation*, and *GasFueledTransportation*
- Now, let’s continue with our example!
TopHat Question 2

Which of these is an invalid superclass/subclass model:

A. ![Diagram A]
B. ![Diagram B]
C. ![Diagram C]
D. None of the above

---

Modeling Inheritance Example (2/3)

- **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
}
```

---

Modeling Inheritance Example (3/3)

- **Step 2 – define a subclass**
- Notice the `extends` keyword
  - `extends` means "is a subclass of" or "inherits 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
};
```
Let's examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods
4. Accessing Instance Variables

Adding new methods (1/3)
- We don't need to (re)declare any inherited methods
- Our Convertible class does more than a generic Car class
- Let’s add a putTopDown() method and an instance variable _top (initialized in constructor)

```java
public class Convertible extends Car {
    private ConvertibleTop _top;
    public Convertible() {
        _top = new ConvertibleTop();
    }
    public void putTopDown() {
        //code with _top elided
    }
}
```

Adding new methods (2/3)
- Now, let’s make a new CS15Mobile class that also inherits from Car
- Can CS15Mobile putTopDown()?
  - Nope. That method is defined in Convertible, so only Convertible and Convertible's subclasses can use it

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

Adding new methods (3/3)
- We don’t need to (re)declare any inherited methods
- Our Convertible class does more than a generic Car class
- Let’s add a putTopDown() method and an instance variable _top (initialized in constructor)
You can add specialized functionality to a subclass by defining methods. These methods can only be inherited if a class extends this subclass.

Let's examine inheritance further:

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods
4. Accessing Instance Variables

- A `Convertible` may decide a `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 Car {
    private Engine _engine;
    // other variables elided
    public Car() {
        _engine = new Engine();
    }
    public void drive() {
        this.goFortyMPH();
    }
    public void goFortyMPH() {
        // 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)

- We override methods by re-declaring and re-defining them
- Be careful—in declaration, the method signature (name of method and list of parameters) and return type 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() {
  }
  @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 method
- When a Convertible is told to drive, it will execute this code instead of the code in its superclass’s drive method (Java compiler does this automatically—stay tuned)

```java
public class Convertible extends Car {
  public Convertible() {
  }
  @Override
  public void drive() {
    this.goSixtyMPH();
  }
}
```
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.

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

Just like previous example, use @Override to tell compiler we're about to override a method. Declare the drive() method, making sure that the method signature and return type match that of superclass's drive method.
Partially overriding methods (4/6)

- When a `CS15Mobile` drives, it first does what every `Car` does: goes 40 mph.
- First thing to do in `CS15Mobile`'s drive method therefore is “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.addDotToMap();
    }

    public void addDotToMap() {
        // code elided
    }
}
```

Partially overriding methods (5/6)

- After doing everything a `Car` does to drive, the `CS15Mobile` needs to add a dot 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.addDotToMap();
        super.drive();
        super.drive();
        this.addDotToMap();
        this.turnOffEngine();
    }

    public void addDotToMap() {
        // code elided
    }
}
```

Partially overriding methods (6/6)

- 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; parent’s methods are inherited by the subclass
  - 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()`?

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

    @Override
    public void drive() {
        super.turnOnEngine();
        super.drive();
        this.addDotToMap();
        super.drive();
        super.drive();
        super.drive();
        this.addDotToMap();
        this.turnOffEngine();
    }
}
```
Method Resolution (1/3)

- When we call `drive()` on some instance of `Convertible`, how does the compiler 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 compiler calls it
  - otherwise, it checks the superclass
    - if method is explicitly defined in superclass, compiler calls it
    - otherwise, checks the superclass up one level... etc.
  - if a class has no superclass, then compiler throws an error

Method Resolution (2/3)

- Essentially, the Java compiler “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 give a command for which there is no method!

Method Resolution (3/3)

- When we call `drive()` on a `Porsche`, Java compiler executes the `drive()` method defined in `Porsche`
- When we call `topDown()` on a `Porsche`, Java compiler executes the `topDown()` method defined in `Convertible`
**Inheritance Example**

- 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)
- Lucy and Angel 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 `Racer`s

---

**A refresher on polymorphism (1/2)**

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

- With last lecture's example, we used polymorphism to pass in different types of transportation to the `useTransportation` method of the `Racer` class
A refresher on polymorphism (2/2)

- A list of transporters can include cars, bikes, planes... but the only method we can call on each transporter is the move method defined by the `Transporter` interface.
- We can only call methods that `Transporter` declares.
  - we sacrifice specificity for generality.
- Why is this useful?
  - allows us to interact with more objects generally
  - i.e., a list of Transporters
    - can have a list of Cars and Bikes

Inheritance Example

- What classes will we need for this lecture’s program?
  - old: App, Racer
  - new: Car, Convertible, CS15Mobile, Van
- Rather than using any `Transporter`, Lucy and Angel are limited to only using Cars.
  - for now, transportation options have moved from Bike and Car to Convertible, CS15Mobile, and Van.
- How do we modify Racer’s `useTransportation()` method to reflect that?
  - can we use polymorphism here?

Inheritance and Polymorphism (1/3)

- What is the “lowest common denominator” between Convertible, CS15Mobile, and Van?

![Inheritance Diagram](image-url)
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.

Transporter bike = new Bike();
Car car = new CS15Mobile();

Transporter and Car are the declared types.
Bike and CS15Mobile are the actual types.

What would happen if we made Car the type of the parameter passed into useTransportation?

We can only pass in Car and subclasses of Car.

Is this legal?

Car convertible = new Convertible();  
_lucy.useTransportation(convertible);

Convertible convertible = new Convertible();  
_lucy.useTransportation(convertible);

Car bike = new Bike();  
_lucy.useTransportation(bike);

Bike is not a subclass of Car, so you cannot treat an instance of Bike as a Car.
Inheritance and Polymorphism (1/2)

- Let's define
  public class Racer {
    // previous code elided
    public void useTransportation(Car myCar) {
      myCar.drive();
    }
  }

- What method should we call on myCar?
  o every Car knows how to drive, which means we can guarantee that every subclass of Car also knows how to drive

Inheritance and Polymorphism (2/2)

- That's all we needed to do!
- Our inheritance structure looks really similar to our interfaces structure
  o therefore, we only need to change 2 lines in Racer in order to use any of our new Cars!
  o 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
  o allows for generic programming
  o treats multiple classes as their generic type while still allowing specific method implementations for specific subclasses to be executed

- Polymorphism + Inheritance is good coding practice

Polymorphism Review

- Polymorphism allows programmers to reference instances of a subclass as their superclass or as instances of an interface they implement
  o relaxation of strict type checking, especially useful in parameter passing
  o e.g. Drive(car: myCar: : ) can take in any kind of Car that is an instance of a subtype of Car and Race(Transporter myTransportation : : ) can take in any instance of a class that implements the Transporter interface

- Advantages
  o makes code generic and extensible
  o treat multiple classes as their generic (declared) type while still allowing instances of specific subclasses to execute their specific method implementations through method resolution based on the actual type

- Disadvantages
  o sacrifice specificity for generality
  o can only call methods specified in superclass or interface, i.e., no putDownTop()
In the following code, the `Salesman` subclass extends the `Employee` superclass. `Employee` contains and defines a `work()` method, and `Salesman` overrides that method.

```
Employee jim = new Salesman();
jim.work();
```

Whose `work()` method is being called?

A. Employee  
B. jim  
C. Dwight  
D. Salesman

Let’s examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods
4. Accessing Instance Variables

Accessing Superclass Instance Variables (1/3)

- Can `Convertible` access `_engine`?
- `private` instance variables or `private` methods of a superclass are not directly inherited by its subclasses
  - superclasses protects them from manipulation by its own subclasses
- `Convertible` cannot directly access any of `Car`’s private instance variables
- In fact, `Convertible` is completely unaware that `_engine` exists!
  - Encapsulation for safety!
  - Programmers typically don’t have access to superclass’ code – know what methods are available but not how they’re implemented
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`

But you can’t access `_engine` directly by `Convertible`’s specialized methods

Instead parent can make a method available for us by its subclasses (`cleanEngine()`)

What if superclass’s designer wants to allow subclasses access (in a safe way) to some of its instance variables directly for their own needs?
For example, different subclasses might each want to do something different to an engine, but we don’t want to factor out and put each specialized method into the superclass `Car` (or more typically, we can’t even access `Car` to modify it)

Car can provide controlled indirect access by defining public accessor and mutator methods for private instance variables

Assume `Car` also has `_myRadio`; `Radio` class defines `setFavorite()` method

Car can provide access to `_myRadio` via `getRadio()` and `setRadio(...)` methods

Important to consider this design decision in your own programs — which properties will need to be directly accessible to other classes?
- Don’t always need both `set` and `get`
- They should be provided very sparingly
- Setter should error-check received parameter(s) so it retains some control, e.g., don’t allow negative values

Defining Accessors and Mutators in Superclass

Assume `Car` also has `_myRadio`; `Radio` class defines `setFavorite()` method

Car can provide access to `_myRadio` via `getRadio()` and `setRadio(...)` methods

Important to consider this design decision in your own programs — which properties will need to be directly accessible to other classes?
- Don’t always need both `set` and `get`
- They should be provided very sparingly
- Setter should error-check received parameter(s) so it retains some control, e.g., don’t allow negative values
Methods are inherited, potentially (partially) overridden

Additional methods and instance variables are defined to specialize the subclass

Instance variables are also inherited, but only "pseudo-inherited"; i.e., are part of a subclass' set of properties...but they can't be directly accessed by the subclass

Instead, accessor/mutator methods are the proper mechanism with which a subclass can change those properties

This provides the parent with protection against children's potential misbehavior

Calling Accessors/Mutators From Subclass

Convertible can get a reference to _radio by calling this.getRadio()

Subclasses automatically inherit these public accessor and mutator methods

Note that using "double dot" we've chained two methods together

First, getRadio is called, and returns the radio

Next, setFavorite is called on that radio

Let's step through some code

Somewhere in our code, a Convertible is instantiated

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

The next line of code calls setRadioPresets()

Let's step into setRadioPresets()
Someone calls `setRadioPresets()` on a `Convertible`—first line is `this.getRadio()`.

- `getRadio()` returns `_myRadio`
- What is the value of `_myRadio` at this point in the code?
  - Was it initialized when `Convertible` was instantiated?
  - Java will, in fact, call superclass constructor by default, but we don’t want to rely on that.

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

---

Making Sure Superclassʼs Instance Variables are Initialized

- `Convertible` may declare its own instance variables, which are initialized in its constructor, but what about instance variables pseudo-inherited from `Car`?
- `Car`ʼs instance variables are initialized in its constructor
  - But we donʼt instantiate a `Car` when we instantiate a `Convertible`!
- When we instantiate `Convertible`, how can we make sure `Car`ʼs instance variables are initialized too via an explicit call?
  - Want to call `Car`ʼs constructor without making an instance of a `Car` via new

```java
public class Car {
    private Radio _myRadio;
    //constructor initializing _myRadio and //other code elided
    public Radio getRadio()
    {
        return _myRadio;
    }
}
```

---

`super()`: Invoking Superclassʼs Constructor (1/4)

- `Car`ʼs instance variables (like `_radio`) are initialized in `Car`ʼs constructor
- To make sure that `_radio` is initialized whenever we instantiate a `Convertible`, we need to call superclass `Car`ʼs constructor
- The syntax for doing this is “`super()`”
- Here `super()` is the parentʼs constructor; before, in partial overriding when we used `super drive`, “`super`” referred to the parent itself (verb vs. noun distinction)
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.

Note: Our call to `super()` creates one copy of the instance variables, located deep inside the subclass, but accessible to sub class only if class provides setters/getters (see diagram in slide 57).

**super(): Invoking Superclass's Constructor (2/4)**

```java
public class Convertible extends Car {
    private ConvertibleTop _top;

    public Convertible() {
        _top = new ConvertibleTop();
        this.setupRadioPresets();
    }

    public void setRadioPresets() {
        this.getRadio().setFavorite(1, 95.5);
        this.getRadio().setFavorite(2, 92.3);
    }
}
```

```
public class Car {
    private Racer _driver;

    public Car() {
        _driver = driver;
    }

    public Racer getRacer() {
        return _driver;
    }
}
```

**super(): Invoking Superclass's Constructor (3/4)**

What if the superclass's constructor takes in a parameter?

We've modified Car's constructor to take in a Racer as a parameter.

How do we invoke this constructor correctly from the subclass?

```java
public class Convertible extends Car {
    private ConvertibleTop _top;

    public Convertible(Racer driver) {
        super(driver);
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getRacer().stepOnIt();
    }
}
```

Racer driver

---

**super(): Invoking Superclass's Constructor (4/4)**

In this case, need the Convertible's constructor to also take in a Racer.

This way, Convertible can pass on the instance of Racer it receives to Car's constructor.

The Racer is passed as an argument to `super()` -- now Racer's constructor will initialize Car's driver to the instance of Racer that was passed to the Convertible.

```java
public class Convertible extends Car {
    private ConvertibleTop _top;

    public Convertible(Racer driver) {
        super(driver);
        _top = new ConvertibleTop();
    }

    public void dragRace() {
        this.getRacer().stepOnIt();
    }
}
```
If you don't explicitly call `super()` first thing in your constructor, Java compiler automatically calls it for you, passing in no arguments.

But if superclass's constructor requires an argument, you'll get an error!

In this case, we get a **compiler error** saying that there is no constructor "public Car()", since it was declared with a parameter

```java
public class Convertible extends Car {
    private ConvertibleTop _top;
    public Convertible(Racer driver) {
        _top = new ConvertibleTop();
    }
    public void dragRace() {
        this.getRacer().stepOnIt();
    }
}
```

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 to do its job.

Let's modify all the subclasses of `Car` to take in a number of `Passenger`

```java
public class Convertible extends Car {
    private Passenger _p1;
    public Convertible(Racer driver, Passenger p1) {
        super(driver);
        _p1 = p1;
    }
    //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 extends Car {
    private Passenger _p1, _p2, _p3, _p4;
    public CS15Mobile(Racer driver, Passenger p1, Passenger p2, Passenger p3, Passenger p4) {
        super(driver);
        _p1 = p1;
        _p2 = p2;
        _p3 = p3;
        _p4 = p4;
    }
    //code with passengers elided
}
```
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

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 – no code-reuse
- In this case, we know that all Cars should loadPassengers, but each subclass will loadPassengers very differently
- abstract method is declared in superclass, but not defined – it is up to subclasses farther down hierarchy to provide their own implementations
- Thus superclass specifies a contractual obligation to its subclasses – just like an interface does to its implementors

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 loadPassengers 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 – thus use semicolon after declaration instead of curly braces
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 st) { //code elided }
}

public class SeatGenerator {
    public SeatGenerator() { //code elided }
    public Seat getShotgun() { //code elided }
    public Seat getBackLeft() { //code elided }
    public Seat getBackCenter() { //code elided }
    public Seat getBackRight() { //code elided }
    public Seat getMiddleLeft() { //code elided }
    public Seat getMiddleRight() { //code elided }
}
```

abstract Methods and Classes (5/6)

```java
public class Convertible extends Car {
    @Override
    public void loadPassengers() {
        SeatGenerator seatGen = new SeatGenerator();
        _passenger1.sit(seatGen.getShotgun());
    }
}

public class CS15Mobile extends Car {
    @Override
    public void loadPassengers() {
        SeatGenerator seatGen = new SeatGenerator();
        _passenger1.sit(seatGen.getShotgun());
        _passenger2.sit(seatGen.getBackLeft());
        _passenger3.sit(seatGen.getBackCenter());
    }
}

public class Van extends Car {
    @Override
    public void loadPassengers() {
        SeatGenerator seatGen = new SeatGenerator();
        _passenger1.sit(seatGen.getMiddleLeft());
        _passenger2.sit(seatGen.getMiddleRight());
        _passenger3.sit(seatGen.getBackLeft());
        //more code elided
    }
}
```

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

- Subclasses at any level in inheritance hierarchy can make an 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
    - think of an interface as a class with only undefined methods (no instance variables)
    - but 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.

Quick Comparison: Inheritance and Interfaces

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each subclass can only inherit from one superclass</td>
<td>You can implement as many interfaces as you want</td>
</tr>
<tr>
<td>Useful for when classes have more similarities than differences</td>
<td>Useful for when classes have more differences than similarities</td>
</tr>
<tr>
<td>is-a relationship: classes that extend another class</td>
<td>acts-as relationship: classes implementing an interface define its methods</td>
</tr>
<tr>
<td>Can define more methods to use</td>
<td>Can only use methods defined in the interface</td>
</tr>
<tr>
<td>o i.e. A Convertible is-a Car</td>
<td></td>
</tr>
<tr>
<td>o i.e. Convertible putting top down</td>
<td></td>
</tr>
</tbody>
</table>

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 polymorphism with inheritance and interfaces in Fruit Ninja
Announcements

- LiteBrite early deadline is tonight at 11:59pm
  - On time is Thursday at 11:59pm, Late is Saturday at 10pm
- If you have not received a HW1 or AndyBot grade, email the HTAs ASAP!
- Lab 2 is out! Make sure to go to the Sunlab for this week’s section!