Lecture 7
Inheritance and Polymorphism

Outline

• Inheritance overview
• Implementing inheritance
  - adding new methods to subclass
  - overriding methods
  - partially-overriding methods
• Inheritance and polymorphism
• Accessing instance variables
• Abstract methods and classes

Recall: Interfaces and Polymorphism

• Interfaces are contracts that classes agree to
  - If a class decides to implement given interface, it must define all methods declared in interface; compiler will raise errors otherwise

• Polymorphism: A way of coding generically: reference instances of related classes as one generic type
  - VIolin, Trumpet, Drums all implement Playable interface with single play() method
  - how can we make use of this to code method conduct that will polymorphically take any instrument of type Playable?

public class Conductor {
    //previous code elided
    public void conduct(Playable instrument) {
        instrument.play();
    }
}

// in Orchestra class
Playable violin = new Violin();
Playable trumpet = new Trumpet();
conductor.conduct(violin);
What are the similarities between a convertible and a sedan?

What are the differences?

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, door, brake pedals, fuel systems, etc.
Can we model this in code?

- In many cases, objects can be very closely related to each other: in life and in code
  - 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?

```
Convertible
  - turnOnEngine()
  - turnOffEngine()
  - drive()
  - putTopDown()
  - putTopUp()

Sedan
  - turnOnEngine()
  - turnOffEngine()
  - drive()
  - parkInCompactSpace()
```

- We could build an interface to model their similarities
  - build a Car interface with the following methods:
    - turnOnEngine()
    - turnOffEngine()
    - drive()
    - etc.
  - Remember: interfaces only "declare" methods
  - each class that implements Car will need to "define" Car's methods
  - a lot of these method definitions would be the same across classes
    - Convertible and Sedan would have the same definition for drive(), startEngine(), turnOffEngine(), etc.
- Is there a better way that allows us to reuse code, i.e., avoid duplication?

Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - overriding methods
  - partially-override methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes
In OOP, inheritance is a way of modeling very similar classes, and facilitating code reuse.

Inheritance models an "is-a" relationship.

- A Poodle "is-a" Dog, a Dog "is-a" Mammal
- "Inherits from" = "is-a"
  - A Poodle inherits from Dog
  - Dog inherits from Mammal

This relationship is not bidirectional.

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

We will cover exactly what is inherited in Java class hierarchy shortly...
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
    - e.g., no Labradoodle as it would inherit from Poodle and Labrador
    - other languages, like C++, allow for multiple inheritance, but too easy to mess up

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 (i.e., code) 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 – stay tuned)
Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - overruling methods
  - partially-overriding methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes

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 extends its superclass by:
  - adding new methods:
    - the subclass should define specialized methods. Not all Animals can swim, but Fish can
  - overriding inherited methods:
    - a Lion 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 all methods (more on this later?)

Modeling Inheritance Example (1/3)

- Let’s model a Van, a CS15Mobile (Sedan), and a Convertible class with inheritance!
Andries van Dam © 2021 9/30/21

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 inherit from Car, FourWheelTransportation, and GasFueledTransportation

TopHat Question 1
Which of these is an invalid superclass/subclass model:

A. 
B. 
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
  
  public class Car {
  private Engine myEngine;
  public Car() {
    this.myEngine = new Engine();
    this.myEngine.start();
    this.myEngine.start();
    this.myEngine.steamClean();
    public void drive() {
      // code elided
    }
    // more methods elided
  }
}
Modeling Inheritance Example (3/3)

- Step 2 - define a subclass
- Remember 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

public class Convertible extends Car {
  //code elided for now
}

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() {
    this.top = new ConvertibleTop();
  }
  public void putTopDown() {
    //code using this.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
}
```
You can add specialized functionality to a subclass by defining methods. These methods can only be inherited if a class extends this subclass. For example, a `Convertible` may decide that a `Car`'s `drive()` method just doesn't cut it. A `Convertible` drives much faster than a regular car.

A subclass can override a parent class’s method and redefine it. For example, `Porsche` inherits `Car`'s methods and doesn't inherit `Convertible`'s methods.

The `@Override` annotation is used to indicate that a method is overriding an inherited method from the superclass.

Here is the code for `Car` and `Convertible`:

```java
public class Car {
    private Engine myEngine;
    // other variables elided

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

    public void drive() {
        this.goFortyMPH();
    }

    public void goFortyMPH() {
        // code elided
    }
    // more methods elided
}
```

```java
public class Convertible extends Car {
    public Convertible() {
    }

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

    public void goSixtyMPH() {
        // code elided
    }
}
```
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 takes in no parameters; the return type must also match.

*return type must be the same or subtype of superclass’s method’s return type, e.g., if the superclass method returns a Car, the subclass method cannot return a Car or a subclass of Car.

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

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();
    }
    public void goSixtyMPH() {
        //code elided
    }
}
```

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 Convertible extends Car {
    public Convertible() {
    }
    @Override
    public void drive() {
        this.goSixtyMPH();
    }
    public void 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.
  
  Car:
  ```java
  public class Car {
    public void drive() {
      Go 40mph
      Add dot to map
    }
  }
  ```

  CS15Mobile:
  ```java
  public class CS15Mobile extends Car {
    public CS15Mobile() {
      // code elided
    }
    @Override
    public void drive() {
      super.drive();
      this.addDotToMap();
      // code elided
    }
  }
  ```

Partially overriding methods (3/6)

- Just like previous example, use `@Override` to tell compiler we’re about to override an inherited method.
  
  Declare the `drive()` method, making sure that the method signature and return type match that of superclass’s `drive` method.

  ```java
  public class CS15Mobile extends Car {
    public CS15Mobile() {
      // code elided
    }
    @Override
    public void drive() {
      super.drive();
      this.addDotToMap();
      // code elided
    }
  }
  ```

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 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();
      // code elided
    }
  }
  ```
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();
    }

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

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();
        this.addDotToMap();
        this.turnOffEngine();
    }
}
```

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
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 (2/3)

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

Method Resolution (3/3)

Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - overriding methods
  - partially overriding methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes
Let’s use the car inheritance relationship in an actual program

Remember the race program from last lecture?

Silly Premise
- the department received a $mystical$ donation and can now afford to give all TAs cars! (we wish)
- Harriet and Lila 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

Inheritance Example

What classes will we need for this lecture’s program?
- old: App, Racer
- new: Car, Convertible, CS15Mobile, Van

Rather than using any instances of type Transporter, Lila and Harriet are limited to only using instances of type Car
- 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 and Polymorphism (2/3)

- Can we refer to CS15Mobile as its more generic parent, Car?
- Declaring CS15Mobile as of type Car follows the same process as declaring a Bike as of type Transporter
- 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 passed into useTransportation?
  - we can only pass in Car and subclasses of Car

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

Is this legal?

```java
Car convertible = new Convertible();
this.lila.useTransportation(convertible);
Convertible convertible = new Convertible();
this.lila.useTransportation(convertible);
Car bike = new Bike();
this.lila.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
  ```java
  public class Racer {
    //previous code elided
    public void useTransportation(Car myCar) {
      myCar.drive();
    }
  }
  ```
- 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.

Inheritance and Polymorphism (2/2)

- 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
  - treats multiple classes as their generic type while still allowing specific method implementations for specific subclasses to be executed
- Maximum flexibility: polymorphism + inheritance and/or interfaces

Polymorphism Review

- Polymorphism allows programmers to refer to instances of a subclass or a class which implements an interface as of type <superclass> or as of type <interface>, respectively:
  - relaxation of strict type checking, particularly useful in parameter passing:
    - e.g. `drive(Car myCar){…}` can take in any kind of Car that is an instance of a subclass of Car and `Racer(Transporter myTransportation){…}` can take in any instance of a class that implements the Transporter interface
- Advantages:
  - makes code generic and extensible
  - 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:
  - sacrifice specificity for generality
  - can only call methods specified in superinterface or interface, i.e., no putTopDown()
In the following code, the \textit{Survivor} subclass extends the \textit{RealityTV} superclass. \textit{RealityTV} defines a \texttt{play()} method, and \textit{Survivor} overrides that method.

\begin{verbatim}
RealityTV episode = new Survivor(); episode.play();
\end{verbatim}

Whose \texttt{playEpisode()} method is being called?

A. \textit{RealityTV}  
B. \textit{Survivor}

---

### Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - overriding methods
  - partially-overriding methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes

---

### Accessing Superclass Instance Variables (1/3)

- Can \texttt{Convertible} access \texttt{engine}?
- \texttt{private} instance variables or \texttt{private} methods of a superclass are not directly inherited by its subclasses
- \texttt{superclass} protects them from manipulation by its own subclasses
- \texttt{Convertible} cannot directly access any of \texttt{Car}'s private instance variables
- In fact, \texttt{Convertible} is completely unaware that \texttt{engine} exists!

Encapsulation for safety!
- Programmers typically don't have access to superclass' code — know what methods are available (i.e., their declarations) but not how they're implemented

---

```java
public class Car {
    private Engine myEngine;
    // other variables elided
    public Car() {
        this.myEngine = new Engine();
    }
    public void turnOnEngine() {
        this.myEngine.start();
    }
    public void turnOffEngine() {
        this.myEngine.shutOff();
    }
    public void drive() {
        // code elided
    }
    // more methods elided
}
```
Accessing Superclass Instance Variables (2/3)

- 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

```
public class Convertible extends Car {
    public void cleanCar() {
        this.myEngine.steamClean();
        //additional code
    }
}
```

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

Accessing Superclass Instance Variables (3/3)

- 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, a familiar pattern!

```
public class Car {
    private Engine myEngine;
    //other instance variables elided
    public void cleanEngine() {
        this.myEngine.steamClean();
    }
    //other methods
}
```

```
public class Convertible extends Car {
    public void cleanCar() {
        this.cleanEngine();
        //additional code
    }
}
```

```
public class Convertible extends Car {
    public void cleanCar() {
        this.myEngine.steamClean();
        //additional code
    }
}
```

---

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
  - `set`/`get` should error-check received parameter(s) so it retains some control, e.g., don’t allow negative values

```
public class Car {
    private Radio myRadio;
    private Engine myEngine;
    //other instance variables
    public Car() {
        this.myRadio = new Radio();
        this.myEngine = new Engine();
    }
    //other methods
    public Radio getRadio() {
        return this.myRadio;
    }
    public void setRadio(Radio radio) {
        this.myRadio = radio;
    }
    //other methods
}
```

```
public class Convertible extends Car {
    public void setRadio(Radio radio) {
        this.myRadio = radio;
    }
    //other methods
}
```
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

Review of Inheritance and Indirect (“pseudo”) Inheritance of Instance Variables

Convertible can get a reference to myRadio by calling

\texttt{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 \texttt{Convertible} is instantiated

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

The next line of code calls \texttt{setRadioPresets()}

Let’s step into \texttt{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.

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

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

Making Sure Superclass’s 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`’s instance variables are initialized in its constructor.
  - but we don’t instantiate a `Car` when we instantiate a `Convertible`!
- When we instantiate `Convertible`, how can we make sure `Car`’s instance variables are initialized too via an explicit call?
  - want to call `Car`’s constructor without making an instance of a `Car` via `new`.

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

`super()` is a special keyword in Java that allows a subclass to invoke the constructor of its superclass. It is used to make sure the superclass’s instance variables are initialized when a subclass instance is created.

```
super(): Invoking Superclass’s Constructor (1/4)
```

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

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

### Example Code:

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

### Question:

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?

### Solution:

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`.
What if we don’t call super()?

- 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

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, Convertible’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 Passengers

Notice how we only need to pass driver to super()

We can add additional parameters in the constructor that only the subclasses will use
Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - overriding methods
  - partially-overriding methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes

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 defined – thus use semicolon after declaration instead of curly braces

```java
public abstract class Car {
    private Racer driver;
    public Car (Racer driver) {
        this.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 return a Seat in a specific logical position

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

```java
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)

- All concrete subclasses of `Car` override by providing a concrete implementation for `Car`'s abstract `loadPassengers()` method
- As usual, method signature and return type must match the one that `Car` declared

```java
public class Convertible extends Car {
    @Override
    public void loadPassengers()
    {
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getShotgun());
    }
}
```

```java
public class CS15Mobile extends Car {
    @Override
    public void loadPassengers()
    {
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getShotgun());
        this.passenger2.sit(seatGen.getBackLeft());
        this.passenger3.sit(seatGen.getBackCenter());
    }
}
```

```java
public class Van extends Car {
    @Override
    public void loadPassengers()
    {
        SeatGenerator seatGen = new SeatGenerator();
        this.passenger1.sit(seatGen.getMiddleLeft());
        this.passenger2.sit(seatGen.getMiddleRight());
        this.passenger3.sit(seatGen.getBackLeft());
    }
}
```
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 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
    - has only undefined methods (no instance variables)
    - 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>i.e. A Convertible is a Car</td>
<td>i.e. Convertible putting its top down</td>
</tr>
<tr>
<td>Can define more methods to use</td>
<td>Can only use methods declared in the interface</td>
</tr>
</tbody>
</table>
Summary

- **Inheritance** models very similar classes
  - factor out all similar capabilities into a generic superclass
  - **superclasses**
    - declare and define methods
    - declare abstract methods
  - **subclasses**
    - 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

Announcements

- Tic Tac Toe deadlines
  - Early handin: tomorrow 10/1 (+2 bonus points)
  - On-time handin: Sunday 10/3
  - Late handin: Tuesday 10/5 (-8 for late handin, but 3 late days to use throughout semester)
- Thank you for reading our emails and being flexible with sections this week – TAs will reach out with final & permanent section rooms before next Tuesday
- HTA Hours: Fridays 3 - 4pm in Friedman 101, or email us!
- ~ special surprise ~ at Tuesday’s lecture

Topics in Socially-Responsible Computing

Government and Antitrust
On Big Tech

• Tuesday: internal ethics committees within companies

What is the government’s role here?

Should we break up big tech?

A brief (and incomplete) history of American antitrust

• Theodore Roosevelt, “trust busting”
  - the age of “robber baron” capitalism
  - Standard Oil, Northern Securities Company, railroad monopoly

• Late 1970s through mid-2010s, University of Chicago school of economics
  - philosophy: against government intervention / for free markets / belief that market forces will self-regulate
  - “Reaganomics” / ‘supply side’ / ‘trickle down’ economics in the US

• Now, competition is decreasing, “greater profits are falling in the hands of fewer firms,” increase in wealth inequality (Harvard Business Review)
  - note: associated with Harvard Business School that has preached shareholder value as key

Lina Khan

• Confirmed 69-28 (narrow!) as chair of FTC on June 15, 2021 after Biden nomination

  - NYT in 2019: “with a single scholarly article, Lina Khan, 29, has reframed decades of monopoly law”
  - wrote “Amazon’s Antitrust Paradox” as a Yale Law student in 2017
  - current doctrine underappreciates the risk of predatory pricing and how integration across distinct business lines may prove anticompetitive
  - “these platforms […] control the essential infrastructure on which their rivals depend”

• neo-Brandeis movement (after Justice Louis Brandeis)
  - “hipster antitrust” — reviving the attitudes of 80 yrs ago
  - push antitrust to include
    - income inequality/wage growth/unemployment
    - consumer rights
    - fighting corporate power
Biden WH Antitrust Policy

- "Executive Order on Promoting Competition in the American Economy" (July 9, 2021)
- "We are now forty years into the experiment of letting giant corporations accumulate more and more power…[this was] a failure." — Biden

Key players
- Tim Wu—coined "net neutrality" appointed to National Economic Council, responsible for technology and competition policy
- Lina Khan—chair of FTC
- Jonathan Kanter—nominated for assistant attorney general

Bonus: Biden WH extremely pro-labor, we'll talk about this later in SRC!

More reading that may be of interest!
- POLITICO Morning Tech Newsletter (I get this every morning and skim the headlines, it is great!)
- "Amazon’s Antitrust Antagonist Has a Breakthrough Idea" (2018) — David Streitfeld, New York Times
- "Biden Names Lina Khan, a Big Tech Critic, as F.T.C. Chair" (2021) — Matt Stoller, New York Times
- "The Antitrust Revolution Has Found Its Leader" (2021) — Matt Stoller, BIG by Matt Stoller
- New Brandeis movement — Wikipedia
- "Executive Order on Promoting Competition in the American Economy" (2021) — The White House
- "FTC says Facebook has been a monopoly since at least 2011 in amended antitrust complaint" (2021) — Russell Branden and Makena Kelly, The Verge