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 chooses 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 the conduct() method so it can polymorphically take in any instrument of type Playable?

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

// in Orchestra class
Conductor conductor = new Conductor();
Playable violin = new Violin();
Playable trumpet = new Trumpet();
conductor.conduct(violin);
Similarities? Differences?

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

Convertibles vs. Sedans

<table>
<thead>
<tr>
<th>Convertible</th>
<th>Sedan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Might have only 2 seats</td>
<td>5 seats</td>
</tr>
<tr>
<td>Top down/up</td>
<td></td>
</tr>
</tbody>
</table>

- 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, doors, 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?

<table>
<thead>
<tr>
<th>Convertible</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>turnOnEngine()</td>
<td></td>
</tr>
<tr>
<td>turnOffEngine()</td>
<td></td>
</tr>
<tr>
<td>drive()</td>
<td></td>
</tr>
<tr>
<td>putTopDown()</td>
<td></td>
</tr>
<tr>
<td>putTopUp()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sedan</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>turnOnEngine()</td>
<td></td>
</tr>
<tr>
<td>turnOffEngine()</td>
<td></td>
</tr>
<tr>
<td>drive()</td>
<td></td>
</tr>
<tr>
<td>parkInCompactSpace()</td>
<td></td>
</tr>
</tbody>
</table>

- 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, i.e., code, 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 overriding methods
- Inheritance and polymorphism
- Accessing instance variables
- Abstract methods and classes
Inheritance

- In OOP, inheritance is a way of modeling very similar classes and facilitating code reuse.
- 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: any level has all of the capabilities of the levels above it but is more specialized than its higher levels.
  - a dog inherits the capabilities of its "parent," so it knows what a mammal knows how to do, plus more.
  - we will cover exactly what is inherited in Java class hierarchy shortly…

Modeling Inheritance (1/3)

- This is an inheritance diagram:
  - each box represents a class.
  - A Poodle "is-a" Dog, a Dog "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 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 its parent's public capabilities.
  - Car defines drive() and Convertible inherits drive() from Car, driving the same way and using Car's code. This holds true for all of Convertible's subclasses as well.
- Inheritance and interfaces both legislate class' behavior, although very different ways.
  - Interface: does not define methods, so all implementing classes **must specify all capabilities** outlined in 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
  - overriding 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
- 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 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 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!
Modeling Inheritance Reminders

- You can create any number of subclasses
  - CSIMobile, 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, FourWheeledTransportation, and GasFueledTransportation

TopHat Question 1

Which of these is an invalid superclass/subclass model?

A. 
B. 
C. 
D. None of the above

Join Code: 504547

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() {
        this.engine = new Engine();
    }
    public void turnOnEngine() {
        this.engine.start();
    }
    public void turnOffEngine() {
        this.engine.shutOff();
    }
    public void cleanEngine() {
        this.engine.steamClean();
    }
    public void drive() {
        // code elided
    }
    // more methods elided
}
```
Modeling Inheritance Example (3/3)

- Step 2 – define a subclass
- Use 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, the class declaration must include:
    ```java
    extends <superclass name>
    ```

```
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)

```
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

```
public class CS15Mobile extends Car {
    public CS15Mobile()
    {
        //other methods elided
    }
}
```
### Adding new methods (3/3)

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

```
// Defines Car's methods and doesn't inherit Convertible's new methods
Car

// Inherits Car's methods and doesn't inherit Convertible's methods
Convertible

// Inherits and adds to Convertible's methods, which includes Car's methods
Porsche
```

### 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
class Car {
    private Engine engine;
    // other variables elided

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

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

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

### Overriding methods (2/4)

- `@Override` should look familiar!
- saw it when we implemented an interface method.
- Include `@Override` right before declaring method we want to override.
- `@Override` is an annotation — in a subclass it signals to compiler (and to anyone reading your code) that you’re overriding an inherited method of the superclass.

```java
class Convertible extends Car {

    public Convertible() {
    }

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

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

Overriding methods (4/4)

- Fill in body of overridden 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 magically – stay tuned).

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.
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:**
```
void drive:
Go 40mph
```

**CS15Mobile:**
```
void drive:
Go 40mph
Add dot to map
```

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.

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

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.

```
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }
    @Override
    public void drive(){
        // super refers to parent class
        super.drive();
        this.addDotToMap();
    }
    public void 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
- 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 uses it
  - otherwise, checks superclass up one level... etc.
  - if a class has no superclass, then compiler throws an error

```java
public class CS15Mobile extends Car {
    public CS15Mobile() {
        //code elided
    }
    @Override
    public void drive() {
        super.turnOnEngine();
        super.drive();
        this.addDotToMap();
        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 uses it
    - otherwise, checks superclass up one level... etc.
    - if a class has no superclass, then compiler throws an error
```

```
partially overrides" the Car's drive method; it drives the way its superclass does, then does something specialized.
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. Compiler won't let you give a command for which there is no method!

When we call `drive()` on a `Porsche`, Java compiler uses the `drive()` method defined in `Porsche`
When we call `topDown()` on a `Porsche`, Java compiler uses the `topDown()` method defined in `Convertible`

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
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)
  - Lexi and Cannon 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, Lexi and Cannon 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?
  - Car is the LCD!
Inheritance and Polymorphism (2/3)

- Can we refer to CS15Mobile as its more generic parent, Car?
- Declaring CS15Mobile as 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

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

Inheritance and Polymorphism (3/3)

- What would happen if we made Car the type of the parameter passed into useTransportation?
  - Can only pass in Car and subclasses of Car, i.e., anything that is-a Car

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

Is this legal?

```java
Car convertible = new Convertible();
this.lexi.useTransportation(convertible);
Convertible convertible = new Convertible();
this.lexi.useTransportation(convertible);
Car bike = new Bike();
this.lexi.useTransportation(bike);
```

Bike is not a subclass of Car (the two classes have no relationship), 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 type <superclass> or as 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 `Race(Transporter myTransportation)` can take in any instance of a class that implements the Transporter interface
- Advantages
  - makes code generic and extensible
  - treats 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
  - sacrifices specificity for generality
    - can only call methods specified in superclass or interface, i.e., `putTopDown()`
In the following code, the `HungerGames` subclass extends the `SurvivalGame` superclass. `SurvivalGame` defines a `play()` method, and `HungerGames` overrides that method.

```java
SurvivalGame game = new HungerGames();
game.play();
```

Whose `play()` method is being called?

A. `SurvivalGame`
B. `HungerGames`
Accessing Superclass Instance Variables (2/3)

- But that's not the whole story...
- While every instance of a subclass of \texttt{Car} is a \texttt{Car}, it can't access engine directly by \texttt{Convertible}'s specialized methods.
- Instead parent can make a method available for us by its subclasses (\texttt{cleanEngine}())

```
public class \texttt{Convertible} extends \texttt{Car} {
    public void cleanCar() {
        this.cleanEngine();
    }
}
```

Accessing Superclass Instance Variables (3/3)

- What if \texttt{superclass}'s designer wants to allow \texttt{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 \texttt{Car} (or more typically, we can't even access \texttt{Car} to modify it)
- \texttt{Car} can provide controlled indirect access by defining public accessor and mutator methods for private instance variables, a familiar pattern!

```
public class \texttt{Car} {
    private \texttt{Engine} engine;
    //other instance variables elided
    //constructor elided
    public void cleanEngine() {
        this.engine.steamClean();
    }
}
```

Defining Accessors and Mutators in Superclass

- Assume \texttt{Car} also has \texttt{Radio; Radio} class defines \texttt{setFavorite()} method
- \texttt{Car} can provide access to \texttt{radio} via \texttt{getRadio()} and \texttt{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 \texttt{set} and \texttt{get}
- they should be provided very sparingly
- \texttt{setter should error-check} received parameter(s) so it returns some control, e.g., don't allow negative values

```
public class \texttt{Car} {
    \texttt{private \texttt{Radio} radio;}
    \texttt{//other initialization}
    public \texttt{Radio} getRadio() {
        \texttt{return this.radio;}
    }
    public void setRadio(\texttt{Radio} myRadio) {
        \texttt{this.radio = myRadio;}
    }
}
```
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

Methods that define the subclass
\[ \text{Subclass} \]

Methods that are inherited from the superclass
\[ \text{Superclass} \]

Instance variables are inherited, but their properties are "pseudo-inherited"; they can't be directly accessed by the subclass.

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

What is the value of `radio` 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 {
    public Convertible() {
        //code elided
    }

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

```java
public class Car {
    private Radio radio;
    //constructor initializing radio and //other code elided

    public Radio getRadio() {
        return this.radio;
    }
}
```

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`.

```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, 92.5);
    }
}
```

```
public class Car {
    private Radio radio;
    //constructor initializing radio and //other code elided

    public Radio getRadio() {
        return this.radio;
    }
}
```

**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 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).

```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, 92.5);
    }
}
```
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 subclass only if class provides setters/getters (see diagram in slide 56).

---

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?

---

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, `super()`.

- 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.

public class Convertible extends Car {
    private ConvertibleTop top;
    public Convertible(Racer myRacer) {
        //oops, forgot to call super()
        this.top = new ConvertibleTop();
    }
    public void dragRace() {
        this.getRacer().move();
    }
}

Constructor Parameters

- Does CS15Mobile need to have the same number of parameters as Car?
- No!
  - as long as Car’s parameters are among the passed parameters, CS15Mobile’s constructor can take in anything else it needs for its job
- Let’s modify all the subclasses of Car to take in a number of Passengers

public class Convertible extends Car {
    private Passenger p1;
    public Convertible(Racer myRacer, Passenger p1) {
        super(myRacer);
        this.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.

public class CS15Mobile extends Car {
    private Passenger p1, p2, p3, p4;
    public CS15Mobile(Racer myDriver, Passenger p1, Passenger p2, Passenger p3, Passenger p4) {
        super(myDriver);
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
        this.p4 = p4;
    }
    //code with passengers elided
}
Outline

- Inheritance overview
- Implementing inheritance
  - adding new methods to subclass
  - extending methods
  - partially extending 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 class Car {
    private Racer driver;
    public Car(Racer myDriver) {
        this.driver = myDriver;
    }
    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
    }
}

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());
    }
}

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());
    }
}

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());
        //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.
- **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.

---

**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.
Quick Comparison: Inheritance and Interfaces

**Inheritance**
- Each subclass can only inherit from one superclass.
- Useful when classes have more similarities than differences and can share code.
- “is-a” relationship: classes that extend another class.
  - i.e. A Convertible is a Car.
- Can define more methods to specialize.
  - i.e. Convertible putting its top down.

**Interface**
- Classes can implement as many interfaces as you want.
- Useful for when classes have more differences than similarities.
- “acts-as” relationship: classes implementing an interface define its methods.
- Can only use methods declared in the interface.

Announcements

- Tic Tac Toe deadlines:
  - Early handin: today 9/28 (+2 bonus points)
  - On-time handin: Saturday 9/30
  - Late handin: Monday 10/2 (8 for late handin, but 4 late days to use throughout semester)

- SRC Extra Credit Discussion (1 extra point on final grade!)
  - See Ed or website for details
  - Sunday 10/22 at 2pm, 3pm and 4pm

- HTA Hours: Fridays 3 - 4pm in CIT210, or email us!
- ~ special surprise ~ at Tuesday’s lecture
Automation as a force for good

- Take over jobs with dangerous working conditions
- Improve workers' health and safety
- Take over night shifts
- Take over mind-numbing, repetitive jobs
- Work collaboratively with human workers

The flip side of automation...

- Uncertainty about whether it creates as many jobs as it removes
- Can reduce worker welfare if not deployed well
Uncertainty as to whether it creates as many jobs as it removes.

Can reduce worker welfare if not deployed well.

How AI is predicted to enter the workforce:

- **Automating physical labor**
  - Factory automation
  - Self-driving trucks! (est. 3.5 million drivers - US Census)

- **Automating non-physical, routine labor**
  - Bookkeepers
  - Accountants
  - Radiologists
  - Lawyers (est. 62 million jobs - Fed)

- **Automating creative work**
  - Branding
  - Logo design
  - Voice acting
  - ... even art!
  - Even programming!

How can we ensure that automation has good impacts on the labor force?

Support for workers – education & reskilling

**Hard Skills**

**Soft Skills**

Estimated to cost $24,800 per person in the United States! (World Bank, Boston Consulting Group, 2019)
Reskilling Initiatives

Company Specific Programs:
- Ex. Amazon Career Choice Program
- According to BCG ~24% of large companies link reskilling efforts to their corporate strategy

Government Efforts
- 2019 Trump Executive Order addressed AI’s effect on workforce
- Biden has indicated plans to release a similar executive order soon

Ethical limits of AI

Explored this week in lab!

Source: MIT Technology Review

In the limit...
... will anyone need to work?

"Yet there is no country and no people, I think, who can look forward to the age of leisure and of abundance without a dread. For we have been trained too long to strive and not to enjoy."

John Maynard Keynes, Economic Possibilities for our Grandchildren (1930)