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
  - way of referencing instances of related classes as one generic type
    - `Cars` and `Bikes` can both `move()` → refer to them as "of type `Transporter`"
    - if you have `Transporter myBike = new Bike();` then `myBike` will be of actual type `Bike` and of declared type `Transporter`
    - therefore `myBike`, and similarly instances of `Car`, `PogoStick`, etc., can all be passed into the following method, as long as the classes implement `Transporter`

```
public class Racer {

    //previous code elided
    public void useTransportation(Transporter transportation) {
        transportation.move();
    }
}
```


0

Lecture 6

Inheritance and Polymorphism

1

Outline

- Inheritance
- Overriding Methods
- Indirect Inheritance
- Abstract Classes

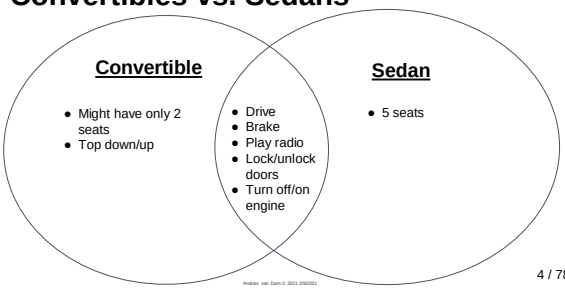
2

Similarities? Differences?

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

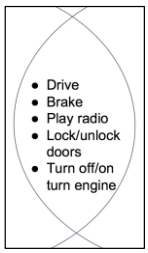
3

Convertibles vs. Sedans



4

Digging deeper into the similarities

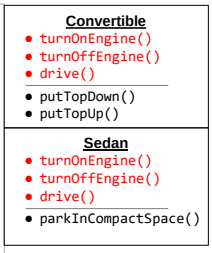


- A convertible and a sedan are extremely similar
- Not only do they share a lot of the same capabilities, they perform these actions in the same way
  - both cars drive and brake the same way
    - let's assume they have the same engine, door, brake pedals, fuel systems, etc.

5

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?



6

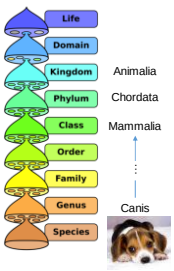
Interfaces

- 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 `implement`s `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?

7

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
  - 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…

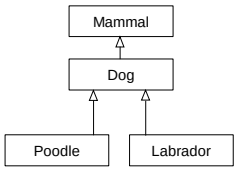


8

Let's examine inheritance further

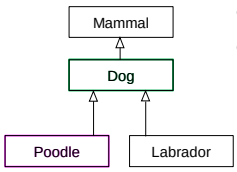
1. Model inheritance relationship
2. Adding new methods
3. Overriding methods
4. Accessing Instance Variables

Modeling Inheritance (1/3)



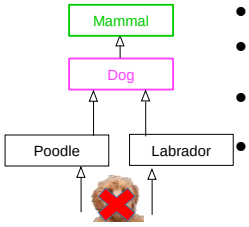
- This is an inheritance diagram
  - each box represents a class
- A Poodle "is-a" Dog, a Dog "is-a" Mammal
  - transitively, a Poodle is a Mammal
- **"Inherits from"** = "is-a"
  - Poodle inherits from Dog
  - Dog inherits from Mammal
    - for simplicity, we're simplifying the taxonomy here a bit
- 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 Labradoodle as it would inherit from Poodle **and** Labrador
  - other languages, like C++, allow for multiple inheritance, but too easy to mess up

12

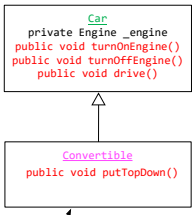
Motivations for Inheritance

- 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

13

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)



Note that we don't list the parent's methods again here – they are implicitly inherited!

14

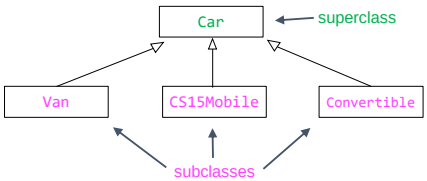
Superclasses vs. Subclasses

- A superclass factors out commonalities among its subclasses
  - describes everything that all subclasses have in common
  - `Dog` defines things common to all `Dog`s
- A subclass extends 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 all methods (more on this later!)

15

Modeling Inheritance Example (1/3)

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



16

Modeling Inheritance Reminders

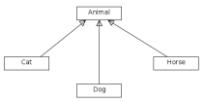
- You can create any number of subclasses
  - `CS15Mobile`, `Van`, `Convertible`, `SUV`...could all inherit from `Car`
  - these classes will inherit public capabilities (i.e., code) from `Car`
- Each subclass can only inherit from one superclass
  - `Convertible` cannot inherit from `Car`, `FourWheeledTransportation`, and `GasFueledTransportation`

17

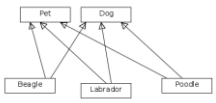
Lecture Question 1

Which of these is an invalid superclass/subclass model:

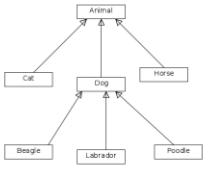
A.



B.



C.



D. None of the above

18

Modeling Inheritance Example (2/3)

- Step 1 – define the superclass
  - defining `Car` is just like defining any other class

```
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
}
```


19

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

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

20

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)

```
public class Convertible extends Car {

    private ConvertibleTop _top;
    public Convertible(){
        _top = new ConvertibleTop();
    }

    public void putTopDown(){
        //code using _top elided
    }

}
```


Adding new methods (2/3)

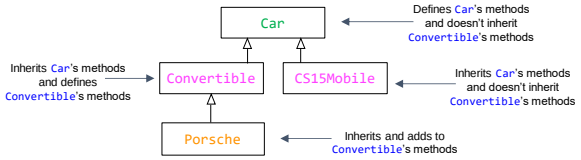
- Now, let's make a new `CS15Mobile` class that also inherits from `Car`
- Can `CS15Mobile` `putTopDown()`?
  - Nope. That method is defined in `Convertible`, so only `Convertible` and `Convertible`'s subclasses can use it

```
public class Convertible extends Car {
    private ConvertibleTop _top;
    public Convertible(){
        _top = new ConvertibleTop();
    }

    public void putTopDown(){
        //code with _top elided
    }
}
```

```
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
- These methods can only be inherited if a class extends this subclass



24

Let's examine inheritance further

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

25

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

```
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
}
```


26

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– in a subclass it signals to compiler (and to anyone reading your code) that you're overriding an inherited method of the superclass

```
public 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

```
public class Convertible extends Car {

    public Convertible() {

    }

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

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

Code from previous slide

*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 should return a car or a subclass of `car`

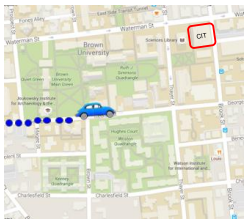
Overriding methods (4/4)

- 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 automagically - stay tuned)

```
public class Convertible extends Car {

    public Convertible() {

    }

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

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

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



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:
    void drive:
        Go 40mph

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


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

```
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 (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`

```
public class CS15Mobile extends Car {

    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive(){
        // super is parent class
        super.drive();
        this.addDotToMap();
    }

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


33

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

```
public class CS15Mobile extends Car {

    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive(){

        super.drive();
        this.addDotToMap();
    }

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


34

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()`?

`StackOverflowError`

```
public class CS15Mobile extends Car {

    public CS15Mobile() {
        //code elided
    }

    @Override
    public void drive(){
        this.turnOnEngine();
        this .drive();
        this.addDotToMap();
        super.drive();
        super.drive();
        this.addDotToMap();
        this.turnOffEngine();
    }
}
```

bad form!

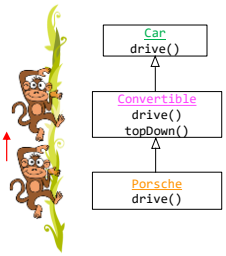
35

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

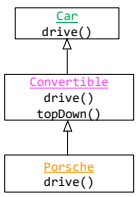
Method Resolution (2/3)

- 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)
  - Marina and Anna 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

A refresher on polymorphism (1/2)

```
public class Race {

    private Racer _marina;
    //other code elided

    public void startRace() {
        _marina.useTransportation(new Bike());
    }
}
```

```
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
- `Racer` class has `useTransportation(Transporter transport)`
- `Race` contains the transportation classes and the `Racer`s

A refresher on polymorphism (2/2)

- A list of transporters can include cars, bikes, planes that implement the transporter interface. But the only method we can call on each such instance 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 instances, generally
  - i.e., arguments of formal type `Transporter`
    - can't have a method with a parameter of both types `Car` and `Bike`

```
Transporter bike = new Bike();

Transporter car = new Car();
```

Lecture Question 2

Given the following interface, class, and `Car` and `Bike` classes from last lecture:

```
public interface Transporter {
    public void move();
}
```

```
public class Scooter {
    public void move(){
        //code to move
    }
}
```

Consider we have an instance `racer` of class `Racer`, which of the following is not a valid use of its `useTransportation()` method?

Recall: The `useTransportation()` method in the Racer class

```
public class Racer {
public void useTransportation(Transporter transportation) {
    transportation.move();
}
}
```

A.
```
Transporter bike = new Bike();
racer.useTransportation(bike);
```

B.
```
Car willsCar = new Car();
racer.useTransportation(willsCar);
```

C.
```
Bike bike = new Bike();
racer.useTransportation(bike);
```

D.
```
Scooter scoot = new Scooter();
racer.useTransportation(scoot);
```


42

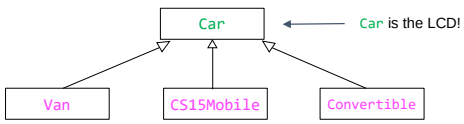
Inheritance Example

- 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`, Marina and Anna 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?

43

Inheritance and Polymorphism (1/3)

- What is the "lowest common denominator" between `Convertible`, `CS15Mobile`, and `Van`?



44

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

```
Transporter bike = new Bike();

Car car = new CS15Mobile();
```


45

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`

```
public class Racer {

    //previous code elided

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

46

Is this legal?

```
Car convertible = new Convertible();
_marina.useTransportation(convertible);
```

```
Convertible convertible = new Convertible();
_marina.useTransportation(convertible);
```

```
Car bike = new Bike();
_marina.useTransportation(bike);
```

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

47

Inheritance and Polymorphism (1/2)

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

```
public class Racer {

    //previous code elided

    public void useTransportation(Car myCar) {
        myCar.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 `Car`s!
  - 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 `Race(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 superclass or interface, i.e., no `putTopDown()`

Lecture Question 3

In the following code, the `LoveIsland` subclass extends the `RealityTV` superclass. `RealityTV` contains and defines a `play()` method, and `LoveIsland` **overrides** that method.

```
RealityTV episode = new LoveIsland();
episode.play();
```

Whose `playEpisode()` method is being called?

A. RealityTV
B. season
C. episode
D. LoveIsland

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
  - superclass 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 (i.e., their declarations) but not how they're implemented

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

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 {
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
        //additional code
    }
}
```

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

```
public class Car {
    private Engine _engine;
    //other instance variables elided

    //constructor elided
    public void cleanEngine() {
        _engine.steamClean();
    }
}
```

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

54

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

55

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`ter should **error-check** received parameter(s) so it retains some control, e.g., don't allow negative values

```
public class Car {
    private Radio _myRadio;
    //other instance variables
    public Car() {
        _myRadio = new Radio();
        //other initialization
    }
    //other methods
    public Radio getRadio(){
        return _myRadio;
    }
    public void setRadio(Radio radio){
        _myRadio = radio;
    }
}
```

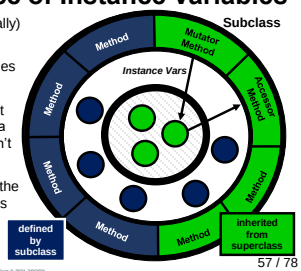
accessor

mutator

56

Review of Inheritance and Indirect ("pseudo") Inheritance of Instance Variables

- 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



57

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`

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

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


58

Let's step through some code

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

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

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

59

Code Step Through

- 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 _myRadio;
   }
}
```


60

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`

61

`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)

```
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);
   }
}
```


62

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

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

```
public class 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);
    }
}
```

Code from previous slide

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)

63

`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?

```
public class Car {

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


64

`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`

```
public class Convertible extends Car {

    private ConvertibleTop _top;

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

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


65

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(…)
        _top = new ConvertibleTop();
    }

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


Constructor Parameters

- Does `CS15Mobile` need to have the same number of parameters as `Car`?
- Nope!
  - as long as `Car`'s parameters are among the passed parameters, `CS15Mobile`'s constructor can take in anything else it wants to do its job
- Let's modify all the subclasses of `Car` to take in a number of `Passenger`s

Constructor Parameters

- 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 Convertible extends Car {
     private Passenger _p1;
     public Convertible(Racer myRacer, Passenger p1) {
         super(driver);
          _p1 = p1;
     }
     //code with passengers elided
}
```

```
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

69

`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 `Car`s 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

70

`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

```
public abstract class Car {

    private Racer _driver;

    public Car(Racer driver) {
        _driver = driver;
    }

    public abstract void loadPassengers();

}
```


71

`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

```
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)

```
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
    }
}
```

- 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

`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.

75

Quick Comparison: Inheritance and Interfaces

Inheritance	Interface
• Each subclass can only inherit from one superclass	• You can implement as many interfaces as you want
• Useful for when classes have more similarities than differences	• Useful for when classes have more differences than similarities
• **is-a** relationship: classes that extend another class ○ i.e. A `Convertible` is-a `Car`	• **acts-as** relationship: classes implementing an interface define its methods
• Can define more methods to use ○ i.e. `Convertible` putting its top down	• Can only use methods declared in the interface

76

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

77

Announcements

- Leap Frog on time deadline: tomorrow 2/10 at 11:59pm
  - Late is Friday 2/12 at 11:59pm
- If you have not received a HW1 or AndyBot grade, email the HTAs ASAP!
- Lab 2 will be going on today and tomorrow during your weekly section time!