

## Responsible CS (1/2)

OpenAi's powerful text generating model : GPT-2

- GPT-2 supposedly generates human-like text
  - Full model considered “too dangerous,” not released yet
  - Smaller models released to select groups
  - Harvard researchers working on fake text detection technology specifically trained on GPT-2



0/83

Sources:  
<https://www.bbc.com/news/technology-49446729>  
<https://news.harvard.edu/gazette/story/2018/01/researchers-develop-a-machine-learning-algorithm-to-extract-facts-from-social-media-sources/>  
<https://doi.org/10.1142/S0219876219500111>

## Responsible CS (2/2)

- Consequences
    - researchers can use the model to better classify which articles are real vs. fake
    - other parties can use the technologies to generate fake news that can confuse the public, and influence their decisions

## Was OpenAI's decision to release its model ethical?

1 / 83

## TopHat Question: Piazza

A screenshot of the Piazza platform interface. At the top, there's a navigation bar with links for 'Home', 'Help', 'Project', 'Exam', 'Logistics', 'Other', 'Analytics', 'Helpdesk', 'FAQs', 'Cartoon', 'Discussions', 'Tests', 'Data Structures and Algorithms', and 'Jobs'. Below the navigation bar is a search bar containing the text 'Search or add a post...'. To the right of the search bar is a button labeled 'New Post'. The main area shows a list of posts with titles like 'How do you calculate the probability of a sequence?', 'What is the difference between a stack and a queue?', and 'What is the difference between a stack and a queue?'. Each post has a 'View' link next to it.

2 / 83

## TopHat Question: TA Hours

Are you attending TA Hours?

- A. Usually (once to multiple times a week)
- B. Sometimes (twice a month)
- C. Not Yet

3 / 83

---

---

---

---

---

---

---

## Top Hat Question: Lectures

In general, how is the pace of in-class lectures?

- A. Moves too slowly
- B. About right
- C. Moves too quickly



4 / 83

---

---

---

---

---

---

---

## TopHat Question: Lecture Capture

Are you reviewing the lecture captures on the website?

- A. Usually (once to multiple times a week)
- B. Sometimes (twice a month)
- C. Not Yet

5 / 83

---

---

---

---

---

---

---

# Lecture 6

## Inheritance and Polymorphism



Andreas van Den Eijnden © 2010 SOA/2010

6 / 83

---



---



---



---



---



---



---



---

## Outline

- [Inheritance](#)
- [Overriding Methods](#)
- [Indirect Inheritance](#)
- [Abstract Classes](#)



Andreas van Den Eijnden © 2010 SOA/2010

7 / 83

---



---



---



---



---



---



---



---

## Similarities? Differences?



Andreas van Den Eijnden © 2010 SOA/2010

8 / 83

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

---



---



---



---



---



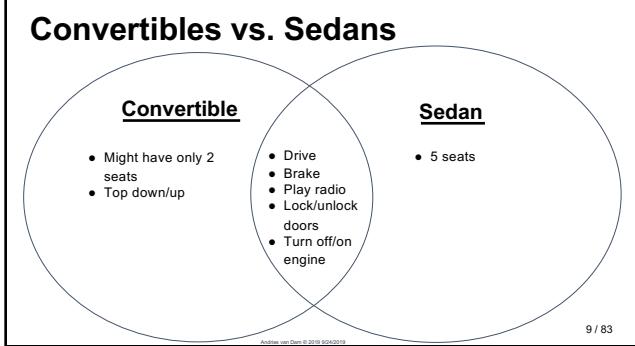
---



---



---




---

---

---

---

---

---

---

---

---

---

## Digging deeper into the similarities

A convertible and a sedan are extremely similar

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

Andries van Den Eijnden © 2010 SOA/2010

10 / 83

---

---

---

---

---

---

---

---

---

---

## Can we model this in code?

- In many cases, objects can be very closely related to each other
  - convertibles and sedans drive the same way
  - flip phones and smartphones call the same way
  - Brown students and Harvard students study the same way
- Imagine we have an **Convertible** and a **Sedan** class
  - can we put their similarities in one place?
  - how do we portray that relationship with code?

<b>Convertible</b> <ul style="list-style-type: none"> <li>putTopDown()</li> <li>putTopUp()</li> <li><b>turnOnEngine()</b></li> <li><b>turnOffEngine()</b></li> <li><b>drive()</b></li> </ul>
<b>Sedan</b> <ul style="list-style-type: none"> <li>parkInCompactSpace()</li> <li><b>turnOnEngine()</b></li> <li><b>turnOffEngine()</b></li> <li><b>drive()</b></li> </ul>

Andries van Den Eijnden © 2010 SOA/2010

11 / 83

---

---

---

---

---

---

---

---

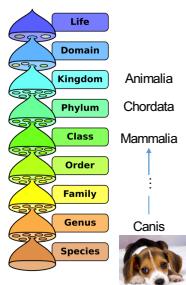
---

---

## 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 extends `Car` will need to “implement” `Car`’s methods
    - a lot of these method implementations would be the same across classes
      - `Convertible` and `Sedan` would have the same definition for `drive()`, `startEngine()`, `shiftToDrive()`, etc.
  - Is there a better way where we can reuse the code?

12 / 83



## Inheritance

- In OOP, inheritance is a way of modeling very similar classes
  - Inheritance** models an “**is-a**” relationship
    - a **sedan** “**is a**” **car**
    - a **poodle** “**is a**” **dog**
    - a **dog** “**is a**” **mammal**
  - Remember: **Interfaces** model an “**acts-as**” relationship
  - You’ve probably seen inheritance before!
    - taxonomy from biology class
    - in biology, any level has all of the guaranteed capabilities of the levels above it but is more specialized
    - a dog **inherits the capabilities** of its “parent,” so it knows what a mammal knows how to do (and more)
    - we will cover exactly what is inherited in Java class hierarchy shortly...

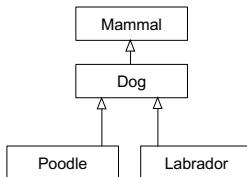
13 /

### **Let's examine inheritance further**

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

14 / 83

## Modeling Inheritance (1/3)

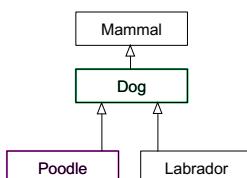


- This is an inheritance diagram
    - each box represents a class
  - A Poodle "is-a" Dog, a Dog "is-a" Mammal
    - transitively, a Poodle is a Mammal
  - "**Inherits from**" = "**is-a**"
    - Poodle inherits from Dog
    - Dog inherits from Mammal
      - for simplicity, we're simplifying the taxonomy 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.)

Andries van Dam © 2019 9/24/2019

15 / 83

## Modeling Inheritance (2/3)

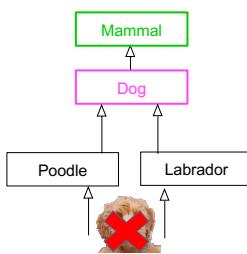


- **Superclass/parent/base:** A class that is inherited from
  - **Subclass/child/derived:** A class that inherits from another
  - “A **Poodle** is a **Dog**”
    - **Poodle** is the **subclass**
    - **Dog** is the **superclass**

Actas VIII Congreso de la AED

16 / 83

## 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

17 / 83

# TopHat Question 1

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

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



18 / 83

---

---

---

---

---

---

---

---

---

---

## Motivations for Inheritance

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

19 / 83

---

---

---

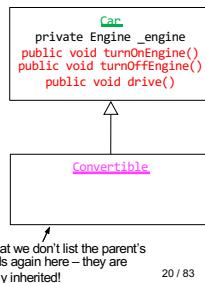
---

---

---

# 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)



20 / 83

---

---

---

---

---

---

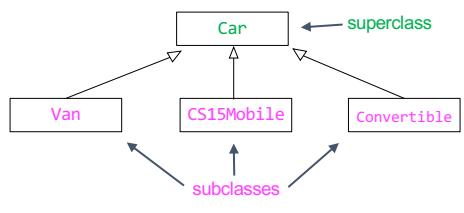
---

---

## **Superclasses vs Subclasses**

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

21 / 83



22 / 83

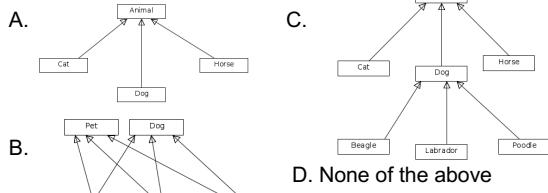
## Modeling Inheritance Reminders

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

23 / 83

## TopHat Question 2

Which of these is an invalid superclass/subclass model:



D. None of the above

24 / 83



## 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.shutdown();
    }
    public void cleanEngine() {
        _engine.steamClean();
    }
    public void drive() {
        //code elided
    }
}
```

25 / 83



## 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  
}
```

26 / 83

## Let's examine inheritance further

1. [Model inheritance relationship](#)
2. **Adding new methods**
3. [Overriding methods](#)
4. [Accessing Instance Variables](#)

27 / 83

---



---



---



---



---



---



---

## 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 with _top elided
    }
}
```

28 / 83

---



---



---



---



---



---



---

## 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
}
```

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

29 / 83

---



---



---



---



---



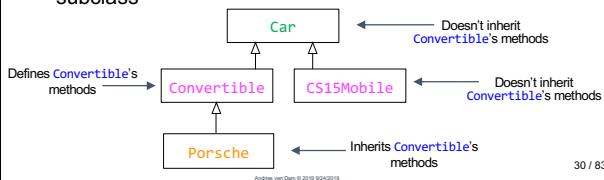
---



---

## 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



30 / 83

---

---

---

---

---

---

---

---

## Let's examine inheritance further

- [Model inheritance relationship](#)
- [Adding new methods](#)
- [Overriding methods](#)
- [Accessing Instance Variables](#)

31 / 83

---

---

---

---

---

---

---

---

## 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
}

```

32 / 83

---

---

---

---

---

---

---

---

## Overriding methods (2/4)

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

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

Andries van Den Eijnden © 2010 SQA/2018

33 / 83

---



---



---



---



---



---



---



---

## Overriding methods (3/4)

- We override methods by re-declaring and re-defining them
- Be careful – in declaration, the method signature (name of method and list of parameters) and return type must match that of the superclass's method exactly\*
  - or else Java will create a new, additional method instead of overriding
- `drive()` is the **method signature**, indicating that name of method is `drive` and it takes no parameters

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

\*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`

Andries van Den Eijnden © 2010 SQA/2018

34 / 83

---



---



---



---



---



---



---



---

## Overriding methods (4/4)

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

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

Andries van Den Eijnden © 2010 SQA/2018

35 / 83

---



---



---



---



---



---



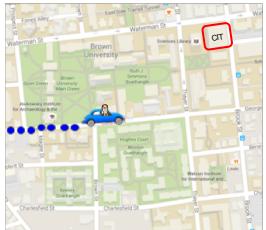
---



---

## 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



36 / 83

## 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
```

37 / 83

## Partially overriding methods (3/6)

- Just like previous example, use `@Override` to tell compiler we're about to override a method
  - Declare the `drive()` method, making sure that 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.addToMap();  
    }  
  
    public void addToMap() {  
        //code elided  
    }  
}
```

38 / 83

## 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
    }
}
```

Andries van Den Eijnden © 2010 SOA42018 39 / 83

## 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
    }
}
```

Andries van Den Eijnden © 2010 SOA42018 40 / 83

## 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()`?

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

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

Andries van Den Eijnden © 2010 SOA42018 41 / 83

## 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

42 / 83

---



---



---



---



---



---



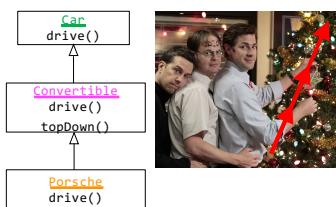
---



---

## 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!



43 / 83

---



---



---



---



---



---



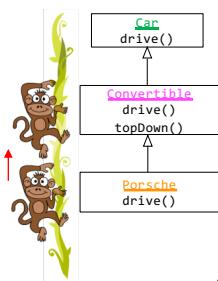
---



---

## 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`



44 / 83

---



---



---



---



---



---



---



---

## Inheritance Example

- Let's use the car inheritance relationship in an actual program
- Remember the race program from last lecture?
- Silly Premise
  - the department received a ~mysterious~ donation and can now afford to give all TAs cars! (we wish)
  - Lucy and Angel want to race from their dorms to the CIT in their brand new cars
    - whoever gets there first, wins!
    - you get to choose which car they get to use

45 / 83

---



---



---



---



---



---



---



---

## Last lecture's final design

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

46 / 83

---



---



---



---



---



---



---



---

## A refresher on polymorphism (1/2)

```
public class Race {
    private Racer _lucy;
    //other code elided

    public void startRace() {
        _lucy.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

47 / 83

---



---



---



---



---



---



---



---

## A refresher on polymorphism (2/2)

- A list of transporters can include cars, bikes, planes... but the only method we can call on each transporter is the move method defined by the `Transporter` interface
 

```
Transporter bike = new Bike();
```
- We can only call methods that `Transporter` declares
  - we sacrifice specificity for generality
- Why is this useful?
  - allows us to interact with more objects generally
  - i.e., a list of `Transporters`
    - can't have a list of `Cars` and `Bikes`

48 / 83

---



---



---



---



---



---



---



---

## Inheritance Example

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

49 / 83

---



---



---



---



---



---



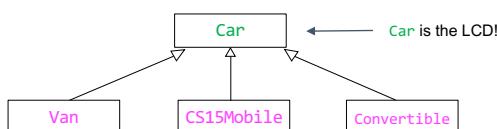
---



---

## Inheritance and Polymorphism (1/3)

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



50 / 83

---



---



---



---



---



---



---



---

## Inheritance and Polymorphism (2/3)

- Can we refer to `CS15Mobile` as its more generic parent, `Car`?
- Declaring `CS15Mobile` as a `Car` follows the same process as declaring a `Bike` as a `Transporter` object
- `Transporter` and `Car` are the declared types
- `Bike` and `CS15Mobile` are the actual types

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

Andries van Den Eijnden © 2010 SQA/2018

51 / 83

---



---



---



---



---



---



---



---

## 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
    }
}
```



Andries van Den Eijnden © 2010 SQA/2018

52 / 83

---



---



---



---



---



---



---



---

## Is this legal?

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

Car cs15Mobile = new CS15Mobile();
_lucy.useTransportation(cs15Mobile); ✓

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

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

Andries van Den Eijnden © 2010 SQA/2018

53 / 83

---



---



---



---



---



---



---



---

## 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();
    }
}
```

Andries van Den Eijnden © 2010 SQA/2018

54 / 83

---



---



---



---



---



---



---



---

## 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
- Polymorphism + Inheritance is good coding practice

Andries van Den Eijnden © 2010 SQA/2018

55 / 83

---



---



---



---



---



---



---



---

## TopHat Question 3

In the following code, the `Salesman` subclass extends the `Employee` superclass. `Employee` contains and defines a `work()` method, and `Salesman` overrides that method.

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

Whose `work()` method is being called?

- `Employee`
- `jim`
- `dwight`
- `Salesman`



Andries van Den Eijnden © 2010 SQA/2018

56 / 83

---



---



---



---



---



---



---



---

## Let's examine inheritance further

1. [Model inheritance relationship](#)
2. [Adding new methods](#)
3. [Overriding methods](#)
4. [Accessing Instance Variables](#)

Andries van Den Eijnd 2010 SQ4/2010

57 / 83

---



---



---



---



---



---



---



---

### 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 but not **how** they're implemented

Andries van Den Eijnd 2010 SQ4/2010

```
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
}
```

58 / 83

---



---



---



---



---



---



---



---

### 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();
    }
}
```



```
public class Car {
    private Engine _engine;
    //other instance variables elided
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
    }
}

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

59 / 83

---



---



---



---



---



---



---



---

## 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

60 / 83

Andries van Den Eijnden 2010 SOA2010

---



---



---



---



---



---



---



---

## Defining Accessors and Mutators in Superclass

- Assume **Car** also has **\_myRadio; Radio** class defines **setFavorite()** method
- Car** can provide access to **\_myRadio** via **getRadio()** and **setRadio(..)** methods
- Important to consider this design decision in your own programs – which properties will need to be directly accessible to other classes?
  - don't always need both **set** and **get**
  - they should be provided very sparingly
  - setter** should **error-check** received parameter(s) so it retains some control, e.g., don't allow negative values

```
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;
    }
}
```

61 / 83

Andries van Den Eijnden 2010 SOA2010

---



---



---



---



---



---



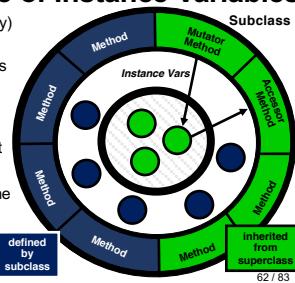
---



---

## 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



Andries van Den Eijnden 2010 SOA2010

---



---



---



---



---



---



---



---

## 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

- o first, `getRadio` is called, and returns the `radio`
  - o 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);  
    }  
}
```

63 / 83

## 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()`

64 / 83

## 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

```
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;  
    }  
}
```

65 / 83

## 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`

66 / 83

---



---



---



---



---



---



---



---

## `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);
    }
}
```

67 / 83

---



---



---



---



---



---



---



---

## `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) 68 / 83

---



---



---



---



---



---



---



---

### 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;
    }
}
```

69 / 83

Andries van Den Eijnden © 2010 SQA/2018

---



---



---



---



---



---



---



---

### 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 driver) {
        super(driver);
        _top = new ConvertibleTop();
    }
    public void dragRace(){
        this.getRacer().stepOnIt();
    }
}
```

70 / 83

Andries van Den Eijnden © 2010 SQA/2018

---



---



---



---



---



---



---



---

### 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 driver) {
        //oops, forgot to call super()
        _top = new ConvertibleTop();
    }
    public void dragRace(){
        this.getRacer().stepOnIt();
    }
}
```

71 / 83

Andries van Den Eijnden © 2010 SQA/2018

---



---



---



---



---



---



---



---

## 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 `Passengers`

72 / 83

---



---



---



---



---



---



---



---

## 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 driver, 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
}
```

73 / 83

---



---



---



---



---



---



---



---

## 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



74 / 83

---



---



---



---



---



---



---



---

## 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

75 / 83

---



---



---



---



---



---



---



---



---

## abstract Methods and Classes (3/6)

- Here, we've modified **Car** to make it an **abstract** class: a class with at least one **abstract** method
- We declare both **Car** and its **loadPassengers** method **abstract**: if one of a class's methods is **abstract**, the class itself must also be declared **abstract**
- An **abstract** method is only **declared** by the **superclass**, not **implemented** – thus use semicolon after declaration instead of curly braces

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

76 / 83

---



---



---



---



---



---



---



---



---

## 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

<pre>public class Passenger {     public Passenger() { //code elided }     public void sit(Seat st) { //code elided } }</pre>	<pre>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 } }</pre>
---	---

77 / 83

---



---



---



---



---



---



---



---



---

## 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 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
    }
}
```

Andries van Den Eijnden © 2018 SQA/CS15

```
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());
    }
}
```

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

78 / 83

---



---



---



---



---



---



---



---

## 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

79 / 83

---



---



---



---



---



---



---



---

## So.. What's the difference?

- You might be wondering: what's the difference between **abstract** classes and interfaces?
- **abstract** classes:
  - can define instance variables
  - can define a mix of concrete and **abstract** methods
  - you can only inherit from one class
- Interfaces:
  - cannot define any instance variables/concrete methods
  - think of an interface as a class with only undefined methods (no instance variables)
  - but you can implement multiple interfaces

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

80 / 83

---



---



---



---



---



---



---



---

## Quick Comparison: Inheritance and Interfaces

### Inheritance

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

Andries van Den Eijnden © 2010 SQA/2010

### Interface

- You 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 defined in the interface

81 / 83

---



---



---



---



---



---



---



---

## 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

Andries van Den Eijnden © 2010 SQA/2010



82 / 83

---



---



---



---



---



---



---



---

## Announcements

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

Andries van Den Eijnden © 2010 SQA/2010

83 / 83

---



---



---



---



---



---



---



---