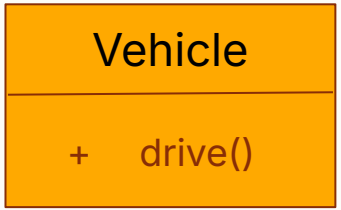
Objects and Polymorphism

Inheritance

- What does “inherit” actually mean?
- Does this always make sense?
- Locks you into *always* adopting the parents behavior

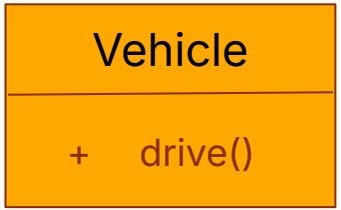
Inheritance

- Vehicle car = new Vehicle()
- Does every vehicle drive the same?



Inheritance

- Vehicle car = new Vehicle()
- Does every car drive the same?

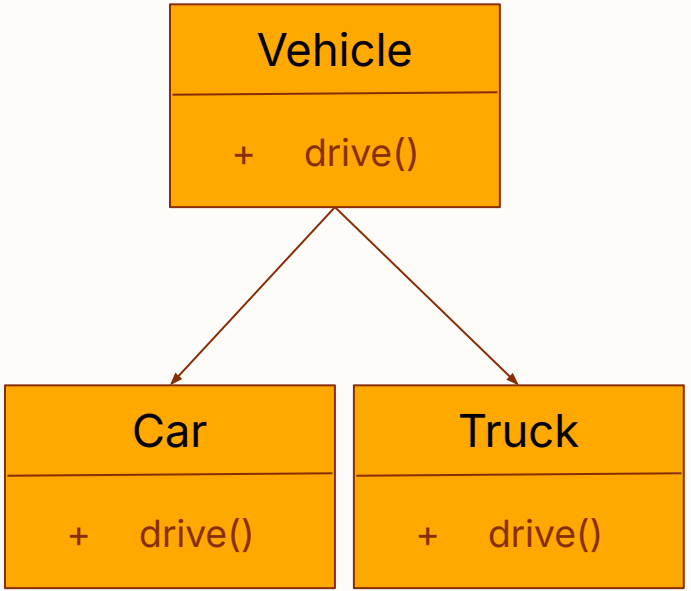


- Of course not!
- So what should we do?

Inheritance

- We could make child classes!

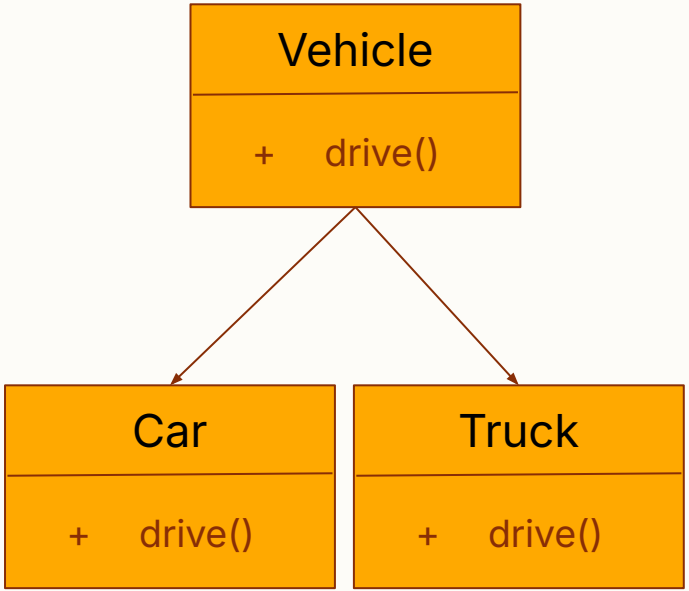
- Vehicle v1 = new Car()
- Vehicle v2 = new Truck()



Inheritance

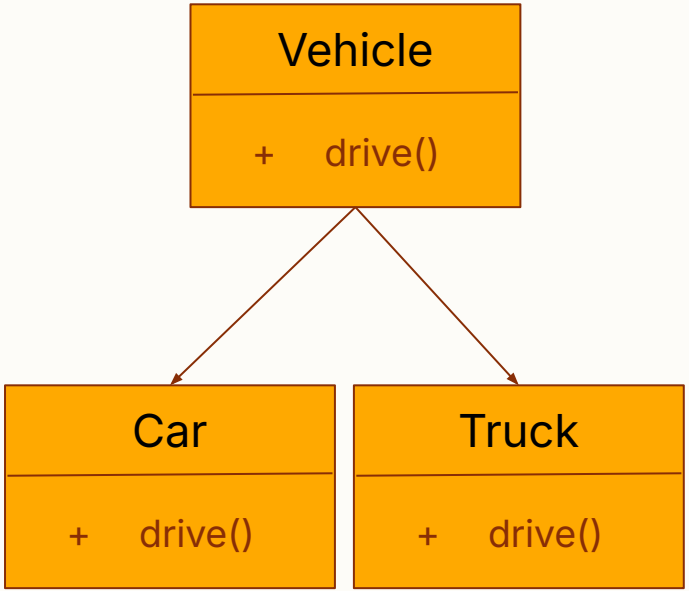
- Does our old code still make sense?

- Vehicle v = new Vehicle()
- How should it drive?



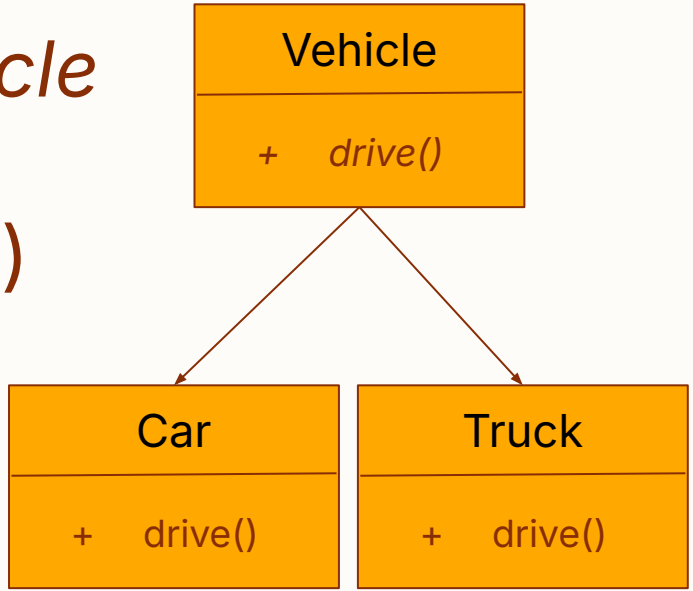
Abstract Classes

- Java “abstract” keyword
- Enforce that this class cannot be instantiated
- Mark methods+classes with “abstract” keyword



Abstract Classes

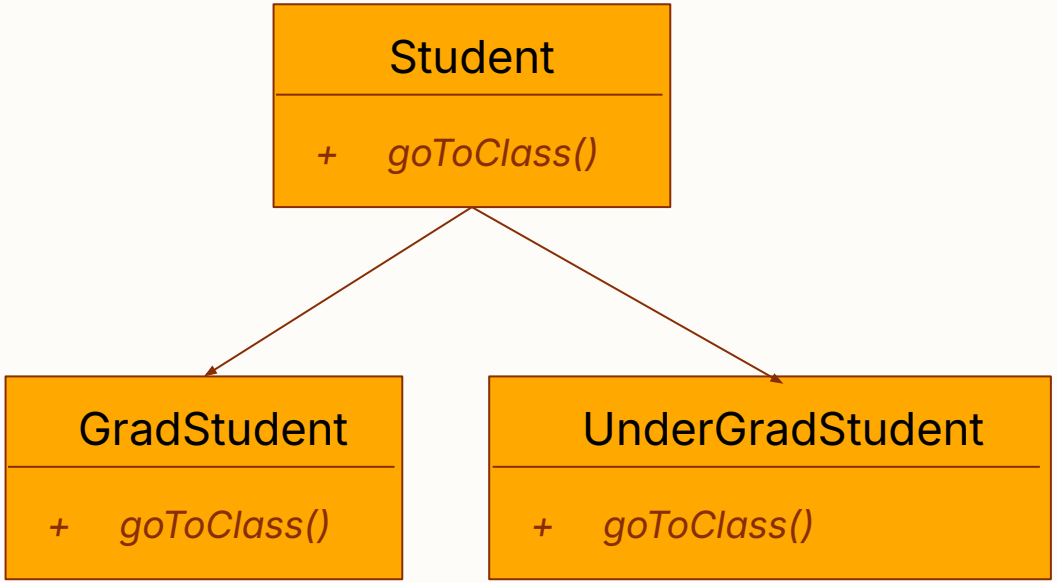
- *public abstract class Vehicle*
- public abstract void drive()
- Each class can implement drive() differently



Abstract Classes

- Let’s do it!

- Student class
- Multiple types



What did that tell us?

- Student Class
  - Described General Behavior
- Child Classes
    - Described Implementation
- Can we mix these?

Abstract Classes Continued

- A class is abstract if
  - It has at least one abstract method
- Not all methods need to be abstract

- Lets add a getGPA() method

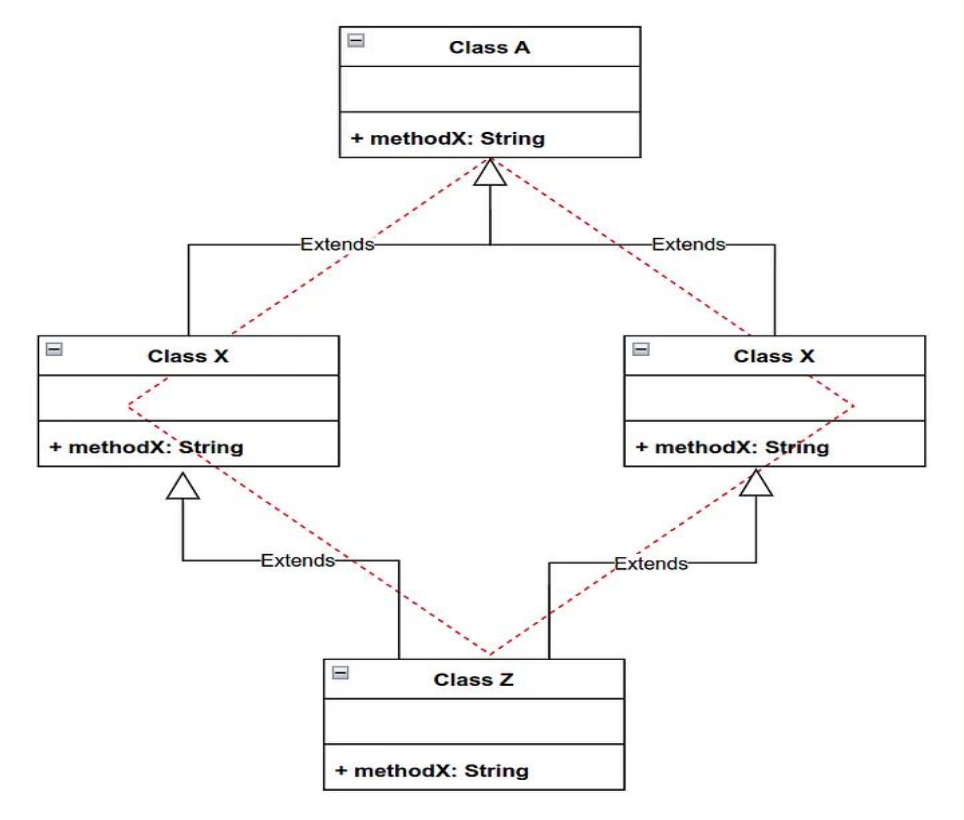
What if we want more?

- What if our GradStudent was also a Commuter?
- public class Commuter

- Now we are forced to pick between extending Commuter or Student

Diamond Problem

- Java does not support multi-inheritance
- Diamond Problem



Interfaces

- Java solves this with interfaces
- Similar to abstract classes
  - Describe method behavior without implementation
- Can implement as many interfaces as the class developer wants

Interfaces

```
public interface Commuter {

    public void driveToCampus()

}
```

Interfaces

- Methods *cannot* specify body
- Use “implements” instead of “extends”
- Interfaces are like contracts

- Lets see that contract in action!
- “must implement the inherited abstract method”

Interfaces

- Implement multiple interfaces with commas
- Java has a number of interfaces built in
- “Comparable” will be very useful to you

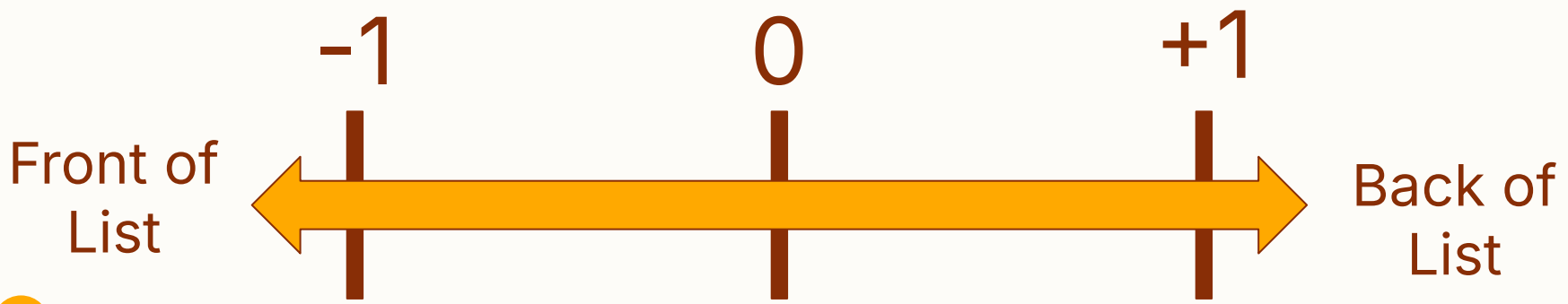
Comparable

- Interface with single method
- “compareTo”
- Used to specify how Collections.sort() works under the hood

- Let’s do it! (with GPA)

Comparable

- How sort two elements?
- “this”
- “other”



Summary

Abstract Classes	Interfaces
No-multi inheritance	Can implement infinite interfaces
Can have default behavior	Cannot* have default behavior
“extends” keyword	“implements” keyword
Not many built in	Many built in interfaces

How do I pick?

- What is the core difference?

How do I pick?

- What is the core difference?

- Default behavior!
- But interfaces are harder to think about
- So why use them?
    - Can’t I just program around hierarchies?

Dynamic Dispatch

- Polymorphism in action
- Interfaces work just like classes
    - Including usage in declarations

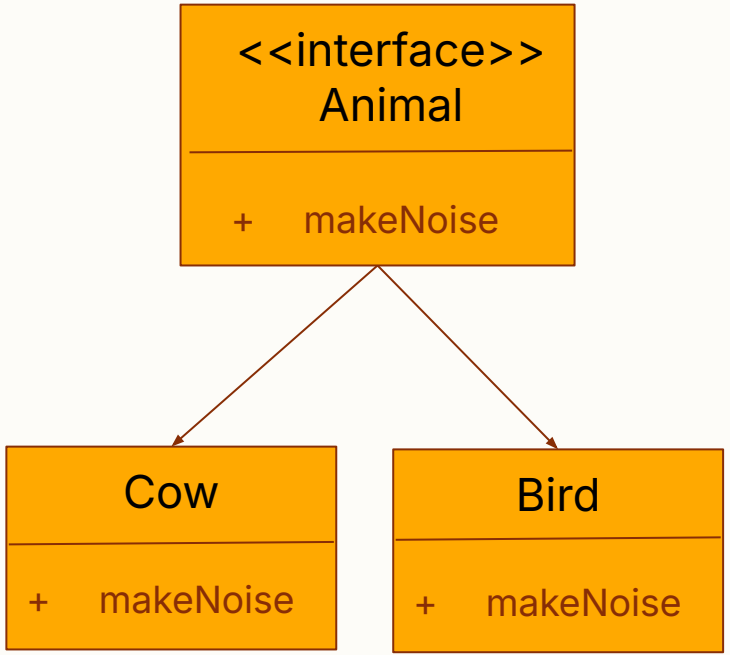
- Interfaces can be used on the left on the equals sign
- Interface i = new ConcreteClass()

Dynamic Dispatch

- Declare variables as interfaces

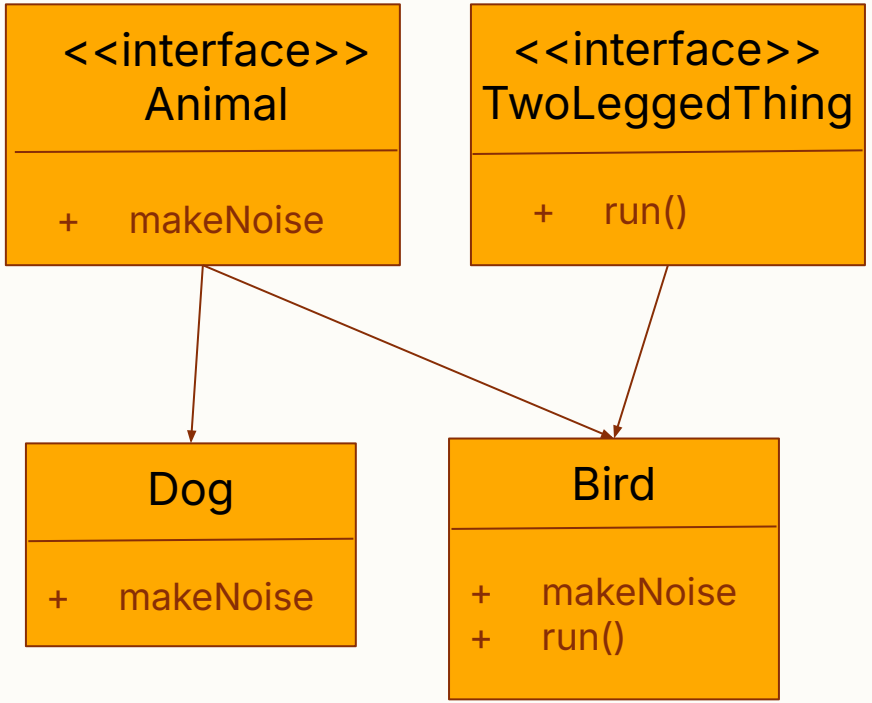
- Dispatch abstract class/interface
- Dynamically determine behavior

- Let’s do it!



Dynamic Dispatch

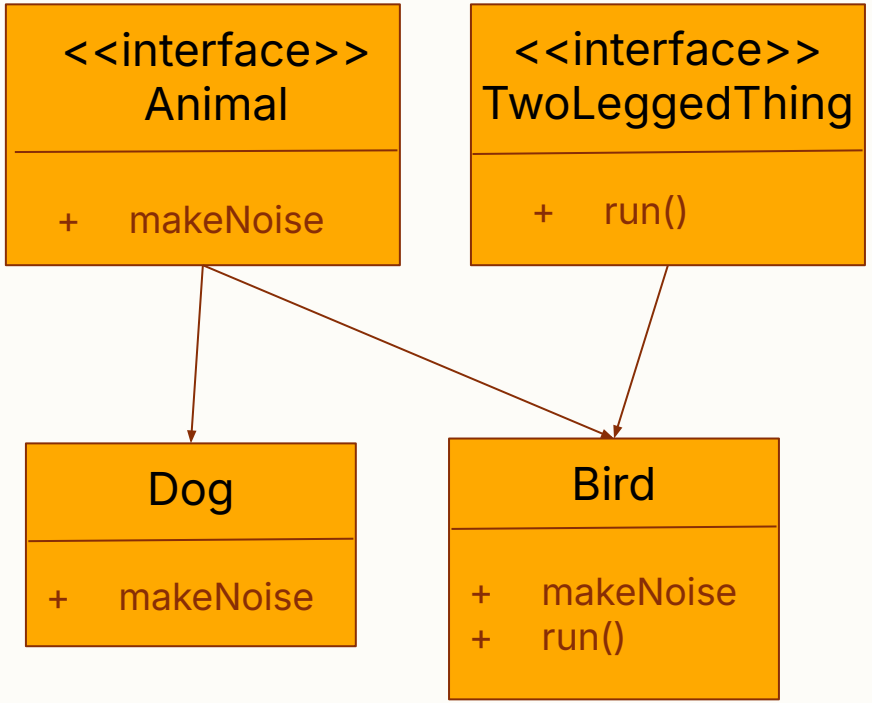
- This works both way
- Dog is not a TwoLeggedThing



Dynamic Dispatch

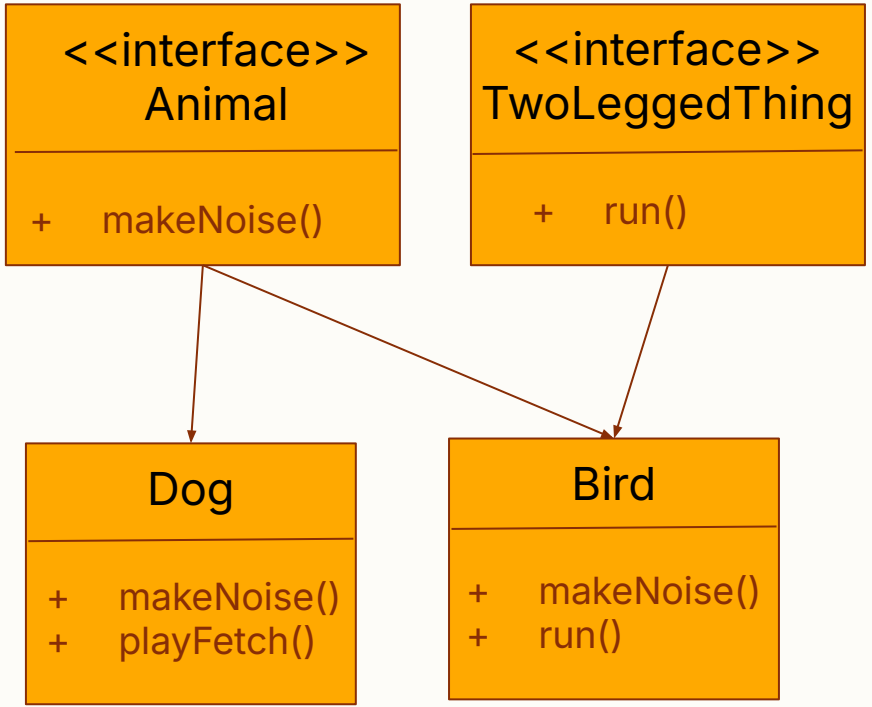
- We can also use this in declarations

public static void wakeBeast(Animal a)



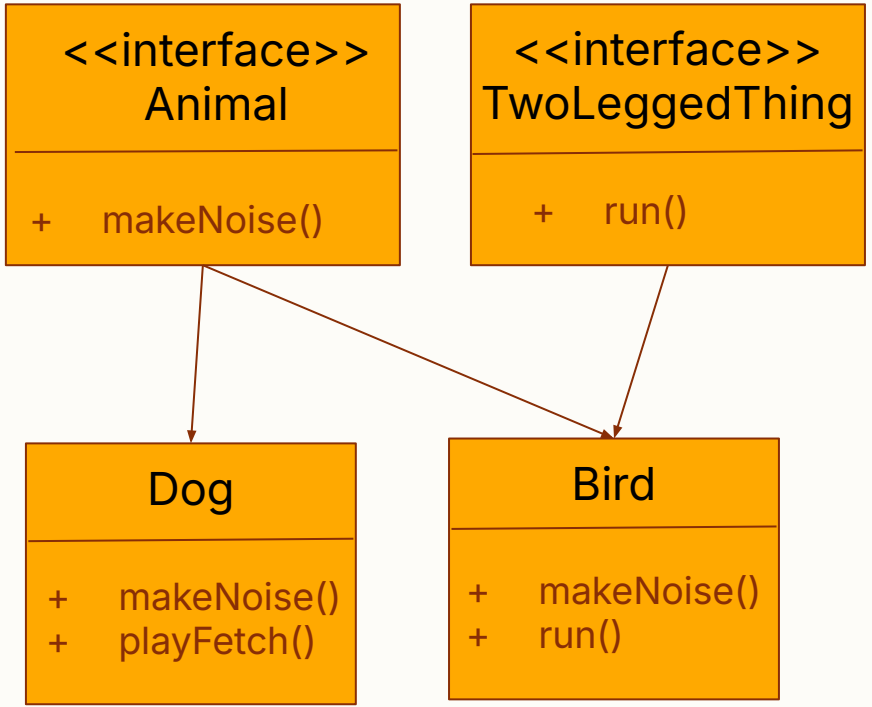
Dynamic Dispatch

- wakeBeast *cannot* use methods not defined in animal
- Even if Dog is passed



Dynamic Dispatch

- Extremely powerful construct
- Buzzword for this is
  - Strategy Pattern
- Interchangeable behavior by design



Compile vs Runtime Assignment

- Left side = Checked at compile time
- Right side = Checked at runtime

- Can we do this?

```
TwoLeggedThing bird = new Bird()
bird.makeNoise()
```

A secret, third option?

- We want a class the keeps Track of logs
- These logs are ordered
- Each log is a string
- Want method to print every other log

A secret, third option?

```
LogList ll = new LogList();
ll.addLog("Event 1 occurred");
ll.addLog("Event 2 occurred");
ll.addLog("Event 3 occurred");
ll.addLog("Event 4 occurred");
ll.addLog("Event 5 occurred");
ll.printEveryOther()
```

A secret, third option?

- What if I wanted my class to have List-like behavior?

A secret, third option?

- What if I wanted my class to have List-like behavior?

- List is an interface!
- ArrayList is a class!
- Lets go check some documentation

A secret, third option?

- How you would have handled this before 1181?
- Is polymorphism really the best option?

A secret, third option?

- How you would have handled this before 1181?
- Is polymorphism really the best option?

- Do we really want our list to have a remove()?

Type Composition

- Let’s just wrap the type instead!

```
public class LogList{
    private ArrayList<String>list = new ArrayList()

    ... Provide only methods we want to

}
```

When do I use this??

- Depends who you work for!
- “Favor composition over inheritance”
- As you code scales, they will both get messy

Brief Aside: Javadoc

- Why do we make you do this?
- Nobody handmade that documentation from earlier
- Generated from source code
- Let’s see it!

Review

- Behavior Modularity
  - Abstract Classes
  - Interfaces
- Separation of Implementation
  - Definitions
  - Implementations

Review

- How can we use this to solve actual problems?
- Data Modeling
- Let’s do an example!

Interface vs. Abstract Class

- Suppose you are creating a media app that allows users to listen to music but also view artwork
- I want to create a class called Media
- Should this be an interface, abstract class, or concrete class?

Media Example

- Considering some of the media items cannot be listened to, what interfaces might make sense to create?

Interface vs. Abstract Class

- Suppose I am creating a system to manage both autonomous and driveable vehicles

Vehicle Tracking System

- Should the following be implemented via an interface, abstract, or concrete class?
  - Vehicle
  - Car
  - UAV
  - Driveable

Practice Problem

- Local Library
- Inventory System
- Managing a large amount of books

Practice Problem

- All books have
  - A Dewey Decimal Number
  - A title
  - A number of days left on loan

Practice Problem

- All Books cost money to borrow
  - Except fiction books are free if you are under the age of 12
- Non-fiction books can have their loans renewed

Practice Problem

- Book Types (DD number, title)
  - Fiction (Cost money)
  - Non-Fiction (Cost Money, can be renewed)

Data Modeling

- Good start to solving *any* problem

- Model how you want your data first
- Implement later
- Adjust model
- Repeat

Data Modeling

- Using the tools we have so far

- How should we model this problem?
- Consider what has “default behavior”

Problem Overview

- All books have:
  - A Dewey Decimal Number
  - A title
  - A number of days left on loan
- Fiction books are free under 12
- Non-fiction books can be renewed

Modeling with Interfaces

- “able” interfaces
- Renewable Interface
- Chargeable Interface
- Abstract Book Class

- Let’s do it!

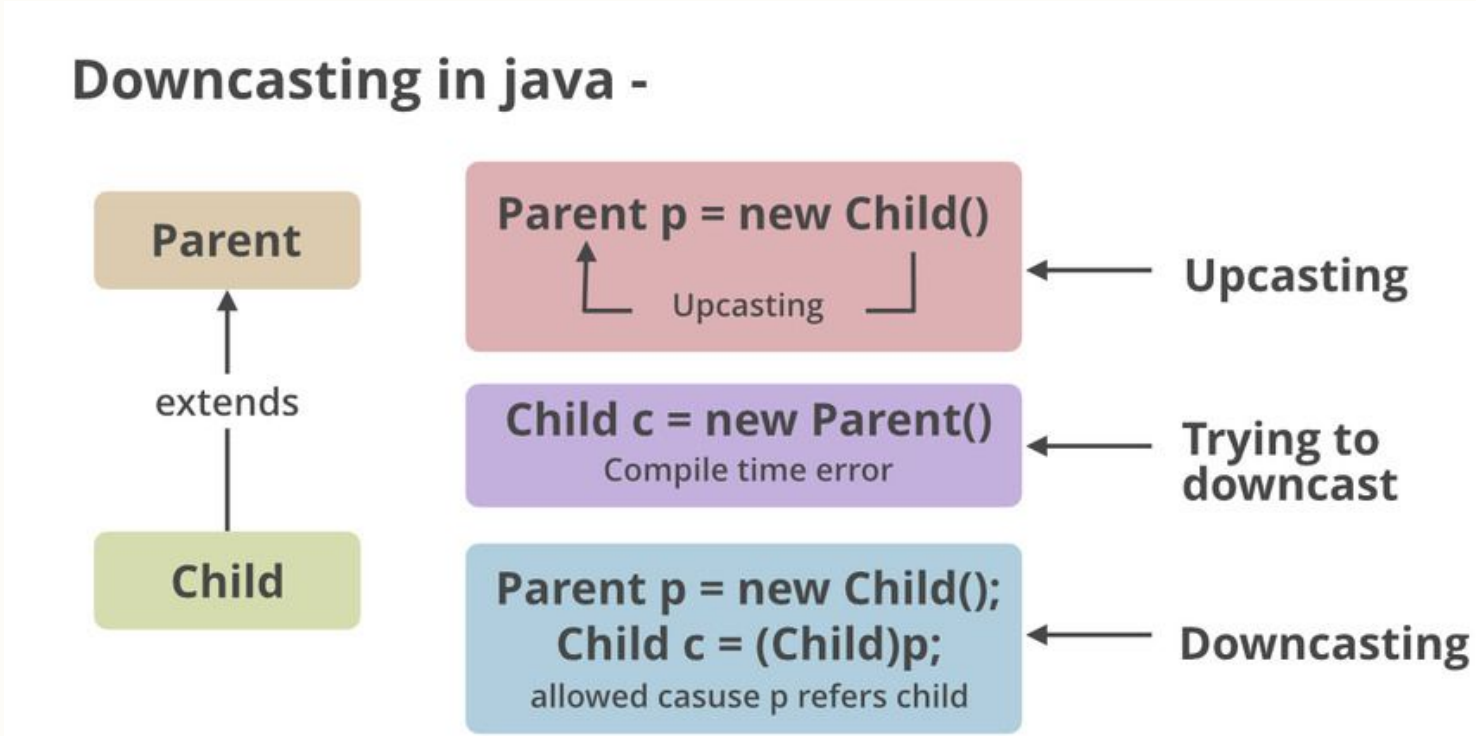
Casting

- Java will let you convert between types
- Cast to interfaces

- checkOut((Borrowable) b3);

- Upcasting vs. downcasting

Casting


Downcasting in java -
Parent
extends
Child
Parent p = new Child()
Upcasting
Upcasting
Child c = new Parent()
Compile time error
Trying to downcast
Parent p = new Child();
Child c = (Child)p;
allowed casuse p refers child
Downcasting