

# Objectives

- Review
- Polymorphism
- Interfaces

# Review Questions

1. What is inheritance?
1. What is the `super` keyword?
1. What does the `static` keyword do?
1. How is a class different from an object?

# Object Oriented Programming: Key Ideas

There are three underlying OOP principles:

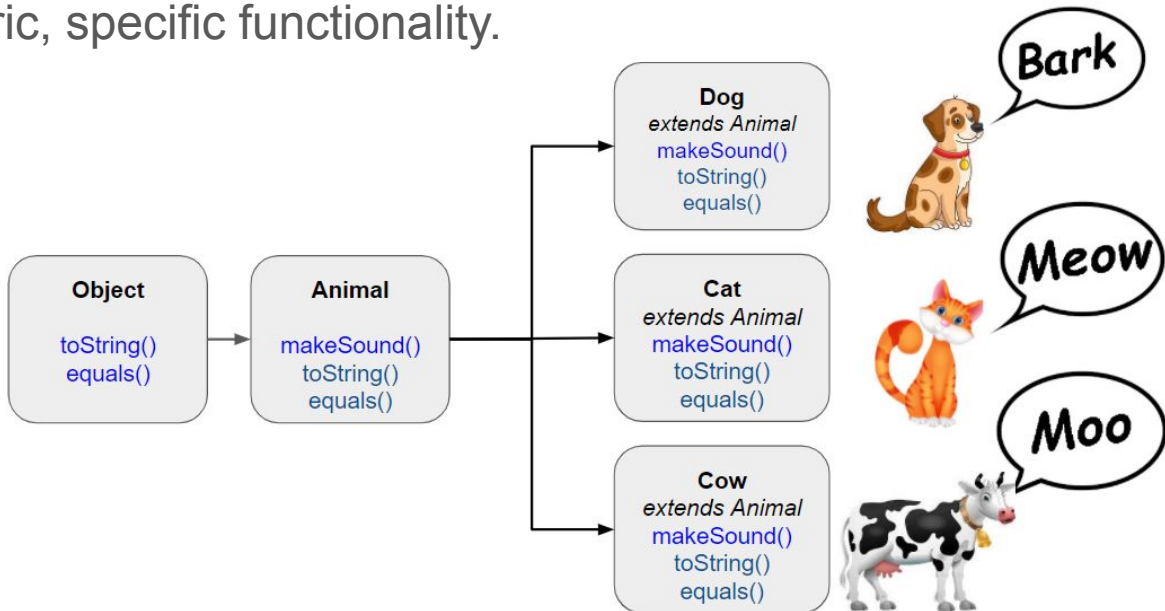
- — ~~Encapsulation~~
- — ~~Inheritance~~
- Polymorphism

By the end of today, you should be able to define **Polymorphism**.

# Polymorphism

Polymorphism is the ability for a subclass to be treated as its superclass, while still providing subclass specific responses. The subclass can be treated generically while still providing non-generic, specific functionality.

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have **many different forms** or stages.



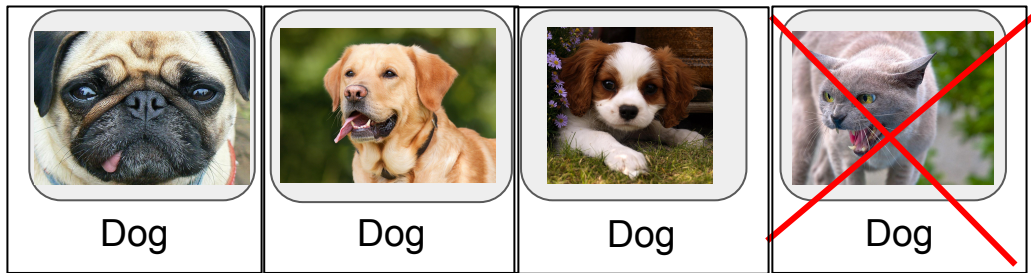
# Polymorphism through inheritance

Classes can be treated as any superclass in their hierarchy.

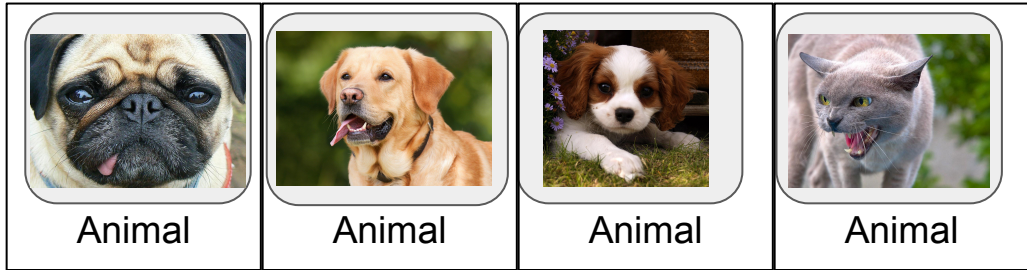
Since the Dog class extends the Animal class, we can treat Dog like an Animal.

Every reference type in Java can be treated as Object, since it is the root superclass of all hierarchies. So we can also treat Dog as an Object.

```
Dog[] petArray = new Dog[4];
```



```
Animal[] petArray = new Animal[4];
```



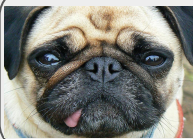


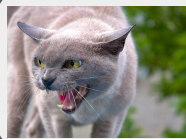
# Polymorphism with Inheritance

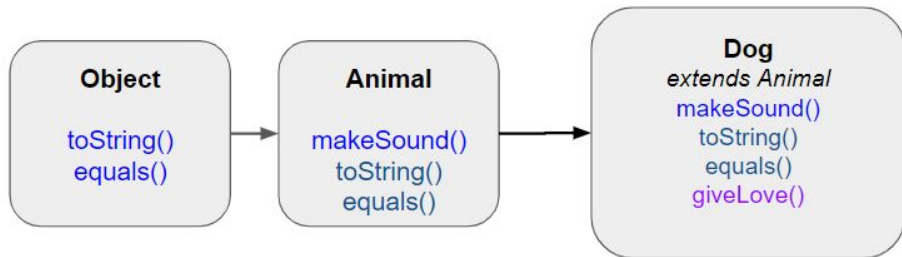
When a *subclass* is **upcast** to its *superclass* the subclass specific *overrides* will still be invoked.

This allows for a subclass to be treated as one of their more generic superclasses and still give responses specific to that subclass.

Translation: When a Dog class is upcast to its parent, Animal class, we can refer to it as its more generic parent (Animal) and we will still get Dog responses.

```
Animal[] petArray = new Animal[4];
```

			
Animal	Animal	Animal	Animal



petArray[0].makeSound() - says  
"snort"

# Objectives

- Interfaces

# Java Interfaces

An interface is best thought of as a contract between the interface itself and a particular class.

Consider the following real world examples:

- A fast food restaurant franchise might stipulate that the franchisee must place a giant logo in the front of the building.
  - *The franchisee is free to choose whatever contractors or workers it needs to actually mount the logo.*
- A fast food restaurant must have the exact menu stipulated by the franchise.
  - *The franchise will not send its own cooks to the restaurant, it is the franchisees responsibility to hire local cooks and make sure that the food is cooked to specification.*



# Interfaces

1. Defines what something can do or how it can be used, but **not how it does it**.
2. A contract that defines what methods an object must have to be of that type, but does not define how those methods will work.
3. **Provides method signatures, but not implementation.**
4. Transitive like inherited methods
  - a. If Class A implements interface B, then A “is-a” B (and so are A’s children)
5. **A single class may have multiple interfaces**, allowing to be part of multiple groupings.
6. Can represent either an HAS-A or an IS-A relationship

```
public interface Drivable {  
    void turn(String direction);  
    boolean accelerate();  
    boolean decelerate();  
}
```

# Implementing Interfaces

A class implements an interface using the `implements` keyword.

```
public class Bicycle implements Drivable
```

When a class implements an interface it must provide Overrides for **ALL** method signatures defined in the interface.

A class can implement an interface and `extend` a subclass.

```
public class Bicycle extends Vehicle implements Drivable
```

A class can only `extend` a single subclass, but can implement multiple interfaces.

```
public class Bicycle extends Vehicle implements Drivable, Explodable
```

# Java Interfaces

This is all of the code we need to write for an interface.

```
public interface Drivable {  
  
    void turn(String direction);  
    boolean accelerate();  
    boolean decelerate();  
  
}
```

1. Interfaces contain only method signatures.
2. The accessor on methods is optional since methods in an interface **can only be public**
3. Interfaces cannot be instantiated, but are a data type
  - a. `Drivable car = new Drivable();` ❌
  - b. `Drivable car = new Car();` ✅
4. Interfaces can have constants (public static final), but not instance variables (they do not have a state, just a outline of what something is able to do)

If the Car class has implemented the Drivable interface, this works.

# Java Interfaces: Abstract Method Rules

When implementing abstract methods on a concrete class, the following rules are observed:

- To fulfill the Interface's contract, the concrete class must implement the method with the exact same return type, exact same name, and exact same number of arguments (with correct data types).
- The access modifier on the implementation cannot be more restrictive than that of that parent Interface.
  - For example - the concrete class cannot implement the method as private if the abstract class has marked it as public.
- All abstract methods are assumed to be public.

# Java Interfaces: Data Members

It is possible for interfaces to have data members, if they do, **they are assumed to public, static, and final.**

# Java Interfaces: Polymorphism References

Interfaces allow us to create references based on the interface, but instantiate an instance of the concrete class instead.

```
Vehicle vehicle = new Car();
```

Have we seen this before? Think about Lists, Maps, and Sets.

To the left of the equal sign is the reference, note it is of the interface type.

To the right of the equal sign is the instantiation of the object, note it is of the concrete class.

# Polymorphism using Interfaces

The interfaces create a data type to which the object can be cast. This allows objects with the same interfaces to be grouped generically, while still providing their specific response when a method is invoked.

```
List<Drivable> drivables = new ArrayList<Drivable>();  
drivables.add( new Car() );  
drivables.add( new Bicycle() );  
drivables.add( new HotAirBalloon() );  
  
for (Drivable d : drivables) {  
  
    d.turn( "Left" );  
    d.accelerate();  
  
}
```

# Summary

- An interface creates a contract (to define its abstract methods) with a class that implements it
  - A class implements an interface using the **implements** keyword
  - `public class Truck implements Driveable{ }`
- Interfaces contain abstract methods and can not be instantiated like a class
  - `Driveable truck = new Driveable();` // ERROR!
  - `Driveable truck = new Truck();` // No Error
- Unlike inheritance, a class can implement many interfaces
  - `public class Truck implements Driveable, Sellable { }`
- The implementing class must provide method definitions for every abstract method in the interface.
  - `@Override`
  - `boolean accelerate(){ // Code }`



# Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces
- Should be able to demonstrate an understanding of where inheritance can assist in writing polymorphic code
- Should be able to state the purpose of interfaces and how they are used
- Should be able to implement polymorphism through inheritance
- Should be able to implement polymorphism through interfaces