

# Polymorphism

# Learning objective

- Define **polymorphism**
- Understand the difference between **declared (compile-time) type** and **actual (run-time) type** for objects
- Learn the **purpose** and **benefit** of **polymorphism**
- Learn **polymorphism** implementation in **Java**



# Agenda

- Recall inheritance concepts/keywords
- Warm-up
- Mini lesson: polymorphism, declared and actual types, benefits of polymorphism
- Coding Time :)
- Exit Ticket

---

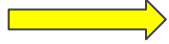
# Recall

**Subclass**



A class that is derived from another class.

**Superclass**



The class from which the subclass is derived

**Keyword super**



Allows to first execute the superclass method and then add on to it in the subclass.

**Method overriding**



Methods have the same name and parameters in the superclass and subclass.

**Method overloading**



Methods have the same name but different parameters in the same class.

# Warm-up

This is toddler toy that has pictures of animals with a handle that when pulled causes an arrow spins. When the arrow stops the toy plays the sound associated with that animal.

**How can we simulate this toy's functionalities in Java?**



# Possible implementation

If you were simulating this toy in software you could create an `Animal` superclass that had a `makeNoise()` method.

Each subclass of `Animal` would override the `makeNoise()` method to make the correct noise for that type.

**Would it work?**

This is **polymorphism**. We have a common parent class, but the behavior is specified in the child class.



# Polymorphism


Polymorphism is derived from Greek:

**poly == "many"**

**morph == "form"**

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

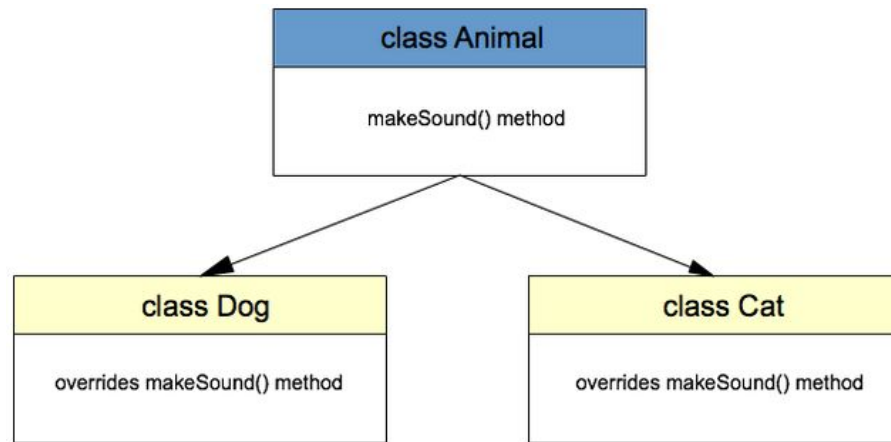
This principle can also be applied to object-oriented programming and languages like the Java. **Subclasses** of a class can **define their own unique behaviors** and yet **share some of the same functionality of the parent class**.



```
public class Animal {  
    public void makeSound() {  
        System.out.println("the animal  
makes sounds");  
    }  
}
```

```
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("woof woof");  
    }  
}
```

```
public class Cat extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("meow meow");  
    }  
}
```





## We know:

```
Animal myAnimal = new Animal();
```

```
Dog myDog = new Dog();
```

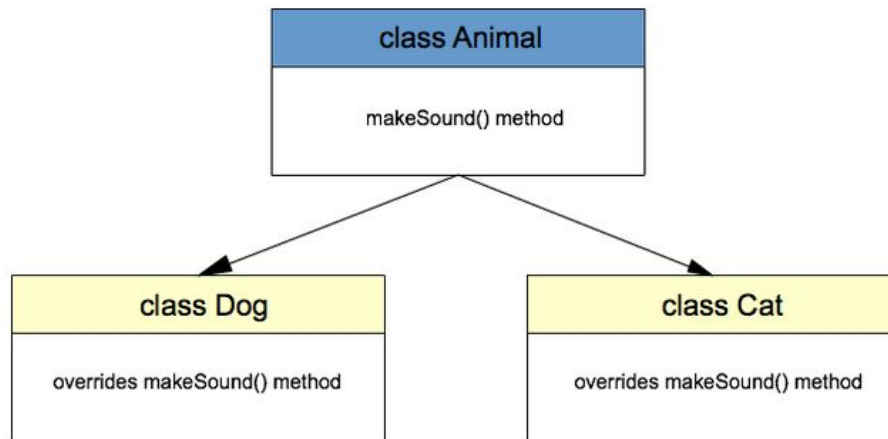
```
Cat myCat = new Cat();
```

## We can also do:

```
Animal myAnimal = new Animal();
```

```
Animal myDog = new Dog();
```

```
Animal myCat = new Cat();
```



# Variable Type

In Java an object variable has both a **declared (compile-time) type** and an **actual (run-time) type**. The declared (compile-time) type of a variable is the type that is used in the declaration. The actual (run-time) type is the class that actually creates the object using new.

Declared type of  
`myDog` is `Animal`



Run-time type of  
`myDog` is `Dog`



```
Animal myDog = new Dog();  
myDog.makeSound();
```

# At Compile Time

At compile time a program gets compiled, methods in or inherited by the declared type determine if all the call methods exist.

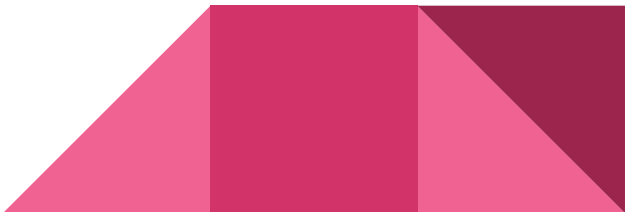
```
Animal myDog = new Dog();
```



myDog has been declared a Animal reference type, but instantiated as a Dog object

```
myDog.makeSound();
```

For this program to compile, makeSound() needs to be a method defined in the Animal class. If it is only defined in the Dog class, it will not compile regardless of whether or not it exists in the Dog class.




# At Run Time

At the time a program runs, the methods in the actual object type gets executed. If the method doesn't exist there, Java looks to the superclass for the method.

```
Animal myDog = new Dog();  
  
myDog.makeSound();
```

When this program runs, Java will look for the `makeSound()` method in the `Dog` class first. If it doesn't exist, it will look in the parent class.



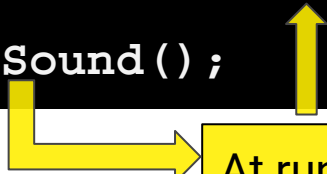
# Run-time progression

At run-time, the execution environment will first look for the `makeSound()` method in the `Dog` class since that is the actual or run-time type.

If it doesn't find it there it will look in the parent class and keep looking up the inheritance tree until it finds the method.

It may go up all the way to the `Object` class. **The method will be found, since otherwise the code would not have compiled.**

```
Animal myDog = new Dog();  
myDog.makeSound();
```



At run-time, Java looks for the `makeSound()` method in the `Dog` class first.

# What about this declaration?

```
Dog myDog = new Animal();  
myDog.makeSound();
```

**NO!!!!**

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Type mismatch: cannot convert from Animal to Dog

myDog cannot be resolved to a variable

# Results

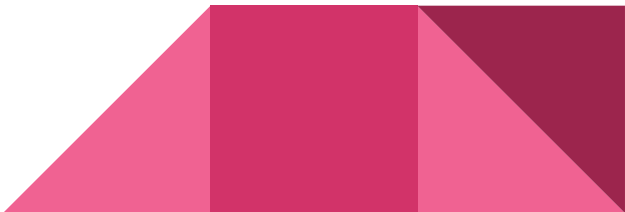
```
public class Animal {  
    private String name;  
  
    public Animal (String inputName) {  
        name = inputName;  
    }  
  
    public void makeSound () {  
        System.out.println("Animals make sounds");  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog (String inputName, String inputBreed) {  
        super(inputName);  
        breed = inputBreed;  
    }  
  
    public void makeSound () {  
        System.out.println("woof woof....");  
    }  
}
```

```
public class Driver {  
    public static void main(String[] args) {  
        Animal myAnimal = new Animal("Cooper");  
        Animal myDog = new Dog("Pipo", "poodle");  
  
        myAnimal.makeSound();  
        myDog.makeSound();  
    }  
}
```

Output:

Animals make sounds  
woof woof....



# Results

```
public class Animal {  
    private String name;  
  
    public Animal (String inputName) {  
        name = inputName;  
    }  
  
    public void makeSound () {  
        System.out.println("Animals make sounds");  
    }  
}
```

```
public class Dog extends Animal {  
    private String breed;  
  
    public Dog (String inputName, String inputBreed) {  
        super(inputName);  
        breed = inputBreed;  
    }  
  
    public void makeSound () {  
        System.out.println("woof woof....");  
    }  
  
    public void getBreed () {  
        System.out.println("I am a cute " + breed) ;  
    }  
}
```

```
public class Driver {  
    public static void main(String[] args) {  
        Animal myAnimal = new Animal("Cooper");  
        Animal myDog = new Dog("Pipo", "poodle");  
  
        myAnimal.makeSound();  
        myDog.makeSound();  
        myDog.getBreed();  
    }  
}
```

## Output:

```
Driver.java:10: error: cannot find symbol  
        myDog.getBreed();  
              ^  
symbol:   method getBreed()  
location: variable myDog of type Animal  
1 error
```



# Why make myDog an Animal?



If we lose the access to Dog methods, why would I declare myDog as an Animal?  
Why not a Dog?

**Animal myDog = new Dog("Pipo", "poodle");**

**Vs.**

**Dog myDog = new Dog("Pipo", "poodle");**

# Why make myDog an Animal?

- You may want to create an Array or ArrayList that stores different types of Animal and declaring myDog as an Animal allows you to combine the object with other Animal objects.

```
ArrayList<Animal> animalList = new ArrayList<Animal>(  
    Arrays.asList(myAnimal, myDog, myCat));  
for (Animal animal: animalList)  
    animal.makeSound();
```

- Sometimes you need to declare the variable before you know what type of object you will have

```
Animal otherAnimal;
```



# Polymorphism - Rules

1. An object can be assigned to a variable of a type that is a superclass.
2. At compile-time, Java only knows that the objects are of the variable type.
3. At run-time, Java knows what class the objects actually are. So if you call a method that gets overridden in a subclass, the overridden version will get used.



# Advantages of Polymorphism

1. It provides **reusability** to the code. The classes that are written, tested and implemented can be reused multiple times.
2. A **single variable can be used to store multiple data values**. The value of a variable you inherit from the superclass into the subclass can be changed without changing that variable's value in the superclass; or any other subclasses.
3. With lesser lines of code, it becomes **easier for the programmer to debug** the code.



# Recap: Polymorphism

- At compile time, methods in or inherited by the declared type determine the correctness of the object.
- At run-time, the method in the actual object type is executed.



Coding Time!!!!!!!

