Week 4: Diving into OOP Pt. 2

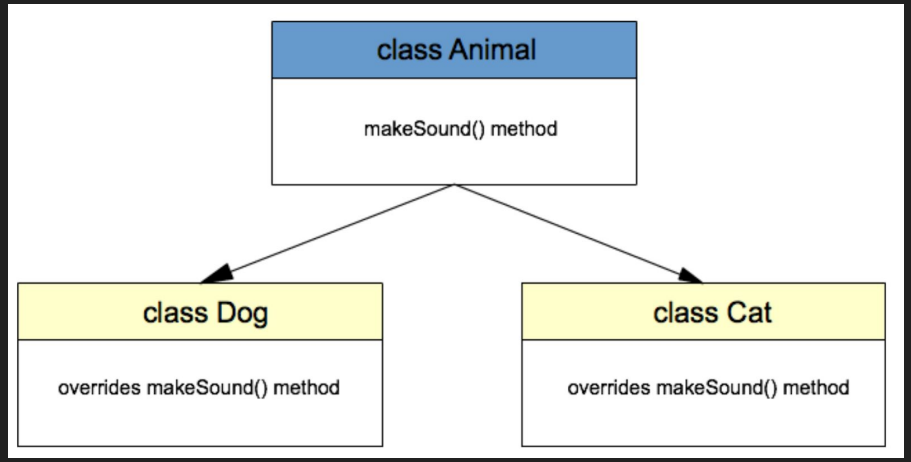
CS 151

Java - Compile Time vs Runtime Errors

- Compile Time Error
  - Errors detected before the code even executes, these prevent running the program
  - Syntax errors (missing semicolon, unmatched braces)
  - Type errors (assigning a String to integer variable)
  - using a class or package that isn't defined or improperly imported
  - in general, things that are fundamentally wrong with the code
- Runtime Error
  - Errors that come from theoretically sound code, but something happens when running
  - Nil pointer errors (attempting to reference `car.Color` when `car` is null)
  - Array Index out of bounds
  - Out of Memory errors (running out of memory due to excessive resource allocation, like a never ending while loop)

Polymorphism

- Inheritance:
  - defines a base class
  - other classes can be derived from it
- Polymorphism:
  - allows objects of derived classes to interpret methods from the base class more specifically
- Example:
  - Class Animal defines makeNoise()
  - various subclasses of Animal provide specific implementations of makeNoise()
  - Although the reference type is Animal, the actual method called depends on the object's actual class



Polymorphism

```java
class Animal {
    public void makeNoise() {
        System.out.println("Some generic animal noise");
    }
}

class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Bark");
    }
}

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

```java
class Bird extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Chirp");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        Animal myBird = new Bird();

        // Method run is based on the object's actual class
        myDog.makeNoise(); // Output: Bark
        myCat.makeNoise(); // Output: Meow
        myBird.makeNoise(); // Output: Chirp
    }
}
```

Polymorphism

- Why is this useful?
- It allows a single method to work on objects of different types
- This allows us to write more effective code with the Animal Class
  - Consider a list of Animals and we iterate through calling makeNoise()
  - As another dev working on Animal class, you don’t need to know how each Animal makes noise

```
// Create a list of Animal objects
List<Animal> animals = new ArrayList<>();

// Create references of type Animal for each object
Animal dog = new Dog();
Animal cat = new Cat();
Animal genericAnimal = new Animal();

// Add the Animal references to the list
animals.add(dog);
animals.add(cat);
animals.add(genericAnimal);

// Iterate through the list and call makeSound() on each Animal
for (Animal animal : animals) {
    animal.makeNoise(); // Polymorphism in action!
}
```

Polymorphism

- Think of it in terms of Parent class and Child class
- A Child learns from its Parents
  - has access to the fields/methods of the superclass
- But a child learns newer things
  - a child class has its own fields/methods that the parent cannot access

```
// Create a list of Animal objects
List<Animal> animals = new ArrayList<>();

// Create references of type Animal for each object
Animal dog = new Dog();
Animal cat = new Cat();
Animal genericAnimal = new Animal();

// Add the Animal references to the list
animals.add(dog);
animals.add(cat);
animals.add(genericAnimal);

// Iterate through the list and call makeSound() on each Animal
for (Animal animal : animals) {
    animal.makeNoise(); // Polymorphism in action!
}
```

Polymorphism

```
Animal myDog = new Dog();
Animal myCat = new Cat();
Animal myBird = new Bird();
```

- Reference type (LHS)
  - type on the left determines methods and properties accessible at compile time
  - **you can only access methods defined in the Animal class**
  - it can be useful to treat a Dog object as an Animal object
  - many methods, such as `equals()`, or anything that involves a collection `[]Animal`, benefit from a more generic and flexible assignment
- Object type (RHS)
  - type of the actual class of the object created at runtime
  - `new Dog()` creates an object of type `Dog`
  - even though you can only call Animal methods, the actual method being run is determined by the object via poly
  - because the object itself is type Dog, Java uses polymorphism to execute the Dog's overridden version of the method

Polymorphism

```
Animal myDog = new Dog();
Animal myCat = new Cat();
Animal myBird = new Bird();
```

- Compile-time (LHS)
  - Checks that we are calling methods that are defined in Animal class
  - dog.makeSound() would be valid because the reference type declares it
- Run-time (RHS)
  - JVM determines the actual object type (Dog), finds the `makeSound()` method in the Dog class, and invokes the overridden version
  - Intuitively: A Dog object in memory should always run the Dog version of `makeSound()`

Casting

- Upcasting
  - converting a subclass reference to a superclass reference
  - implicit casting and is always safe
  - allows treating an object more generally - good for polymorphism, such as storing objects in a collection of the superclass type
- Downcasting
  - converting a superclass reference into a more specific subclass reference
  - is not inherently safe (not all Employees are PartTimeEmployees) so requires explicit casting
  - If you downcast to something that isn't a subclass, you will hit `ClassCastException` at runtime
    - you can check against this by using `instanceof` to validate

```java
class Employee {
    public void work() {
        System.out.println("Employee is working");
    }
}
class HourlyEmployee extends Employee {
    public void calculatePay() {
        System.out.println("Calculating hourly pay");
    }
}
public class Main {
    public static void main(String[] args) {
        // Upcasting to Employee
        Employee emp = new HourlyEmployee(); // Implicit upcasting
        emp.work(); // Output: Employee is working - but what if HourlyEmployee had overriden work()?

        // Downcasting back to HourlyEmployee
        if (emp instanceof HourlyEmployee) {
                HourlyEmployee hourlyEmp = (HourlyEmployee) emp;
                hourlyEmp.calculatePay();
                // Output: Calculating hourly pay
        }
        // genericEmp cannot be casted to HourlyEmployee unless RHS is HourlyEmployee or one of its subclasses
        // Employee genericEmp = new Employee();
        // ((HourlyEmployee) genericEmp).calculatePay(); // This would throw ClassCastException
    }
}
```

Abstract Classes

- Can this method be improved?
  - `.getPay()` might be different for an Hourly vs Salaried Employee
  - Why doesn't polymorphism solve this problem?
    - Employee class cannot provide a default implementation that would make sense for all subclasses
- Ideally: We postpone the definition of `getPay()` until the type of employee is known
- Abstract classes have headings but no method body, just a placeholder

```
public boolean samePay(Employee other) {
    return(this.getPay() == other.getPay());
}
```

```
abstract class Employee {
    // Abstract method, subclasses must implement
    public abstract double getPay();

    // Method to compare pay between employees
    public boolean samePay(Employee other) {
        return this.getPay() == other.getPay();
    }
}
```

Abstract Classes

- Abstract Classes can define methods for subclasses without providing concrete implementations
  - by declaring `getPay` an *abstract* method, all subclasses MUST provide an implementation
- As the name suggests, an abstract Class cannot be instantiated. In this case, an Employee object cannot be created.
- This helps ensure proper polymorphic behavior

```
public boolean samePay(Employee other) {
    return(this.getPay() == other.getPay());
}
```

```
abstract class Employee {
    // Abstract method, subclasses must implement
    public abstract double getPay();

    // Method to compare pay between employees
    public boolean samePay(Employee other) {
        return this.getPay() == other.getPay();
    }
}
```

Abstract Classes

- Abstract classes cannot be private
  - They are designed to serve as base classes and be inherited, and `private` would prevent other classes from accessing it
- Abstract methods have no method body and just end with a semicolon
- If a derived class of an abstract class does not define all the abstract methods, then it too is an abstract class and needs to add the `abstract` modifier

```
public boolean samePay(Employee other) {
    return(this.getPay() == other.getPay());
}
```

```
abstract class Employee {
    // Abstract method, subclasses must implement
    public abstract double getPay();

    // Method to compare pay between employees
    public boolean samePay(Employee other) {
        return this.getPay() == other.getPay();
    }
}
```

Abstract Classes

- Any class with an abstract method is called an abstract class and must use the keyword `abstract`
- A class that has no abstract methods is called a *concrete class*
- Although you cannot create an Object of an abstract class, you can use it as a parameter in various methods

```
abstract class Employee {
    // Abstract method that subclasses must implement
    public abstract double getPay();

    // Method to compare pay between employees
    public boolean samePay(Employee other) {
        return this.getPay() == other.getPay();
    }
}
```

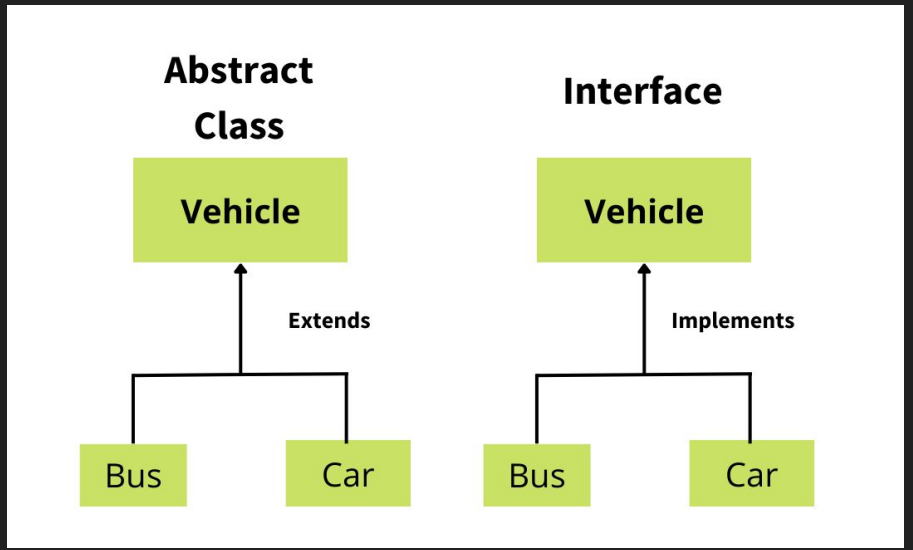
```
class HourlyEmployee extends Employee {
    private double hourlyRate;
    private double hoursWorked;

    public HourlyEmployee(double hourlyRate, double hoursWorked) {
        this.hourlyRate = hourlyRate;
        this.hoursWorked = hoursWorked;
    }

    @Override
    public double getPay() {
        return hourlyRate * hoursWorked;
    }
}

class SalariedEmployee extends Employee {
    private double annualSalary;

    public SalariedEmployee(double annualSalary) {
        this.annualSalary = annualSalary;
    }

    @Override
    public double getPay() {
        return annualSalary / 52;    }
}
```

Abstract Classes

- When do we use abstract classes?
  - You don’t want the base class to be instantiated on its own
  - You want to enforce that all subclasses implement makeNoise() or other methods without providing default implementation
- When to avoid abstract classes?
  - You can provide a reasonable implementation for all methods in the base class
  - You want to allow the base class to be instantiated directly, and makeNoise() could have a generic but meaningless implementation (“some generic animal sound”)
  - When an interface will suffice

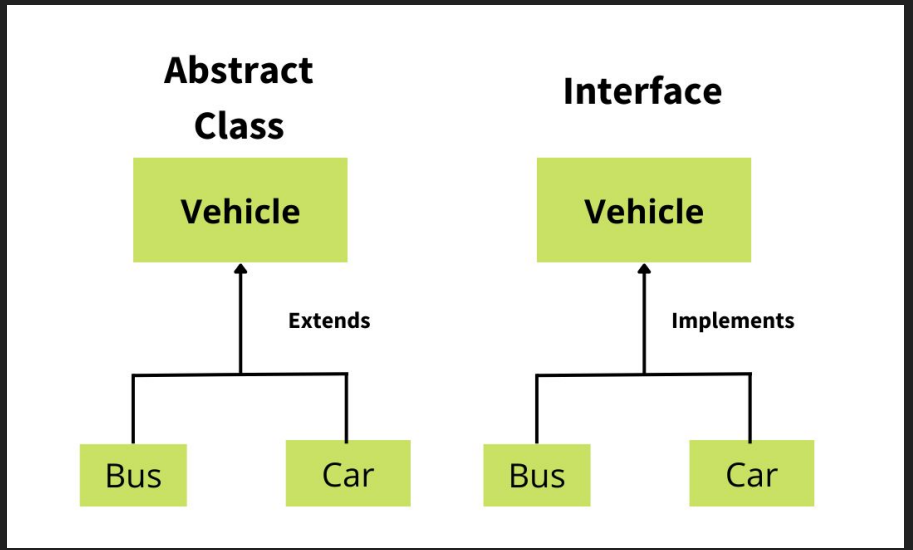
Interfaces

- An interface is similar to a base class in inheritance, but it is not a class
  - Some languages allow one class to be derived from two or more base classes, although this is not allowed in Java
  - Instead, Java allows a class to *implement multiple interfaces*



Interfaces

- An interface is something like an extreme abstract class
  - but it is *not a class*
  - An interface is a *type* that can be satisfied by any class that implements it
  - Think of an interface as an agreement that each class implementing it must satisfy
- It specifies the set of methods that any implementing class must have
  - Contains method headings and constant definitions only
  - No instance variable or any complete method definitions



Interfaces

- A class implementing an interface must implement ALL methods defined in the interface
- An interface and all its method headings should be public
  - and any class that implements it must keep these methods public
- Because an interface is a type, a method can be written with a parameter of an interface type

```
interface Chargeable {
    int STANDARD_VOLTAGE = 220;
    void charge();
}

class Smartphone implements Chargeable {
    private String model;
    public Smartphone(String model) {
        this.model = model;
    }

    @Override
    public void charge() {
        System.out.println("Charging " + model + "
at " + STANDARD_VOLTAGE + " volts.");
    }
}

class ChargingStation {
    public void plugInAndCharge(Chargeable device) {
        System.out.println("Device plugged in.");
        device.charge();
    }
}
```

Interfaces

- Interfaces can also declare defined constants
  - all variables in an interface must be public, static, and final
  - Only because this is understood, Java allows these modifiers to be omitted
- A class can only *inherit* from one base class but can *implement* any number of interfaces

```
interface Chargeable {
    int STANDARD_VOLTAGE = 220;

    void charge();
}

class Smartphone implements Chargeable {
    private String model;

    public Smartphone(String model) {
        this.model = model;
    }

    @Override
    public void charge() {
        System.out.println("Charging " + model + " at
" + STANDARD_VOLTAGE + " volts.");
    }
}
```

Interfaces

```java
// Define an interface
interface Drivable {
    void drive();
}

// Implement the interface in a class
class Car implements Drivable {
    @Override
    public void drive() {
        System.out.println("The car is driving.");
    }
}
```

```java
// Another class with a method that accepts a Drivable interface as a parameter - only things marked Driveable can startDriving
class Driver {
    public void startDriving(Drivable vehicle) {
        vehicle.drive();
    }
}

public class Main {
    public static void main(String[] args) {
        Drivable myCar = new Car();
        Driver driver = new Driver();

        // Pass the interface type as a parameter
        driver.startDriving(myCar);
        // Output: The car is driving.
    }
}
```

Interfaces

```
// Define the first interface
interface Drivable {
    void drive();
}

// Define the second interface
interface Chargeable {
    void charge();
}

// Concrete class that implements both interfaces
class ElectricScooter implements Drivable,
Chargeable {
    private String model;

    public ElectricScooter(String model) {
        this.model = model;
    }

    // Implementing the drive method from Drivable
    @Override
    public void drive() {
        System.out.println("The electric scooter is zooming
along!");
    }

    // Implementing the charge method from Chargeable
    @Override
    public void charge() {
        System.out.println("Charging the electric scooter...");
    }
}

public class Main {
    public static void main(String[] args) {
        ElectricScooter myScooter = new ElectricScooter("Xiaomi
M365");

        myScooter.drive();  // Output: The electric scooter is
zooming along!
        myScooter.charge(); // Output: Charging the electric
scooter...
    }
}
```

Interfaces - instanceOf

- You can also check interface implementation with instanceof
  - if myPhone instanceOf `Smartphone`
  - if myPhone instanceOf `Chargeable`
- Compare this to:

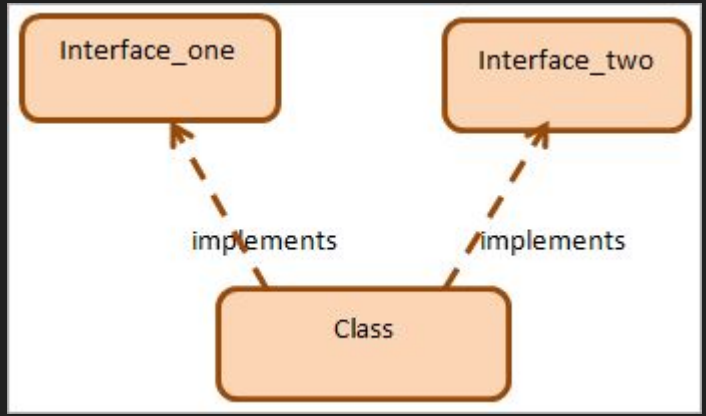
```
    public void plugInAndCharge(Chargeable
device) {
        System.out.println("Device plugged
in.");
        device.charge();
    }
```

- The reference (left hand side) determines the methods available to call

```
public static void main(String[] args) {
    // Create a list of objects, some of which are Drivable
    Object[] objects = new Object[] {
        new ElectricScooter("Model X1"),
        new Car("Tesla"),
        "Just a String", // Not Drivable
        new ElectricScooter("Model Y2")
    };

    for (Object obj : objects) {
        // Use instanceof to check if the object is
Drivable
        if (obj instanceof Drivable) {
            // Cast the object to Drivable and call drive()
            ((Drivable) obj).drive();
        } else {
            System.out.println("This object is not
drivable: " + obj);
        }
    }
}
```

Interfaces - Inconsistency

- When a class implements 2 interfaces
  - Inconsistency will occur if the interfaces have constants with the same name but different values
  - Or when they contain methods with the same signature or different return types
  - If a Class definition implements two inconsistent interfaces, then that Class definition is illegal and compile time exception will be thrown



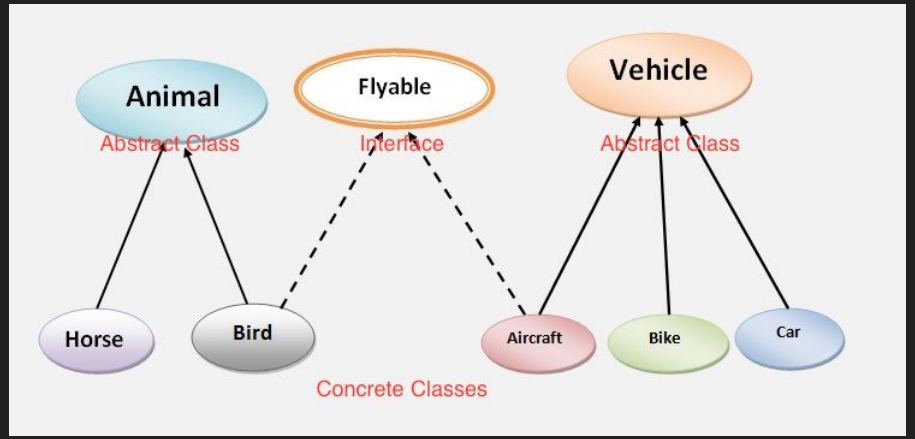
Interfaces vs Polymorphism

- Could this polymorphism example have been interfaces?
  - class Bird implements NoiseMaker which stipulates `void makeNoise`
- Interfaces are contracts - they specify what methods a Class must implement, but doesn't provide any behavior
- Because all Animals have shared attributes, it makes sense for them to have the same superclass

```
class Bird extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Chirp");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        Animal myBird = new Bird();

        // Method run is based on the object's actual class
        myDog.makeNoise(); // Output: Bark
        myCat.makeNoise(); // Output: Meow
        myBird.makeNoise(); // Output: Chirp
    }
}
```

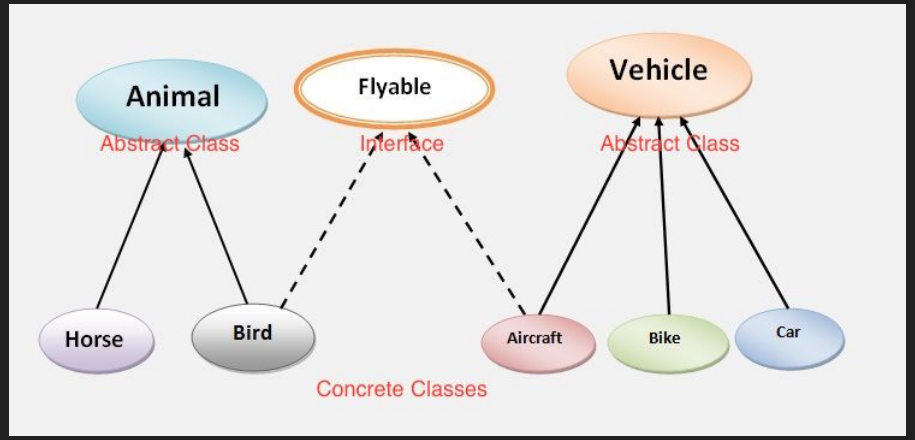
Interfaces vs Abstract Classes

- Interface:
  - Specifies methods that must be implemented
  - Classes can implement multiple interfaces, allowing for a form of multiple inheritance
  - Cannot contain instance variables, only constants (static final variables)
- Abstract Classes:
  - Common base for a group of related classes, some methods can be defined for shared code and some left abstract
  - A class can only extend one abstract class, making it more restrictive
  - Like other classes, it can have instance variables and fully implemented methods to provide common functionality



Interfaces vs Abstract Classes

- Use an interface if:
  - You need somewhat unrelated classes to implement the same methods
  - You want to specify method signatures without implementation details
  - Your classes don’t share an ancestor
- Use an abstract class if:
  - You have common behavior that multiple related classes should inherit
  - You want to provide some default method implementations but leave others abstract
  - You have fields/constructors in the base class that apply to all subclasses



Midterm - October 3rd

Topics covered on the midterm will include, but are not limited to:

- Any material from slides weeks 1 - 6
- Data types, variable assignment, functions/methods in Java
- Classes, Objects, Constructors
- Inheritance, Interfaces, Abstract Classes, Overloading, Overriding, Casting
- UML Class Diagrams, UML Sequence Diagrams
- SDLC, Exception Handling
- Multiple Choice, 50 questions, during normal class time

Please bring a pencil/eraser and your student ID

HW:

- Create abstract class `MediaContent`
  - methods: play(), getDuration()
  - attributes: title, releaseYear, duration
- Create interface: `Downloadable`
  - methods: download()
- Class `Movie` (extends MediaContent, implements Downloadable)
  - attributes: Genre, Director
  - bonus: can you limit the number of acceptable Genres to Comedy and Action?
- class `TVSeries` (extends MediaContent, implements Downloadable)
  - attributes: seasons, episodesPerSeason
- class `Documentary` (extends MediaContent)
  - attributes: category, narrator
- Main class:
  - Create an array of `MediaContent` with items from all 3 classes
  - Loop through and call play() and getDuration() for each object
  - Use instanceof to check for `Downloadable` objects and if available, download it

```
if (media instanceof Downloadable) {
    ((Downloadable) media).download();
}
```

HW 2:

- Create a Zoo class with:
  - Abstract Class Animal
    - methods `makeSound`, `getDiet`
  - Interface Trainable
    - method `performTrick`
  - Class Mammal (extends Animal)
    - Attribute: furType
    - Implements methods from Animal with “generic mammal sound” and “omnivore”
  - Class Bird (extends Animal, implements Trainable)
    - attribute: wingspan (double)
    - getDiet, performTrick return “Insectivore” and “flying in circles!”
- Class Lion (extends Mammal)
  - makeSound, getDiet returns “Roar” and “Carnivore”
- Class Parrot (extends Bird)
  - makeSound, getDiet return “Squawk” and “Herbivore”
  - performTrick returns “the Parrot is mimicking sounds!”
- Main:
  - Create an Array of Lion, Parrot, Mammal, Bird
  - Loop through and print the results of calling `makeSound`, `getDiet`
  - Use `instanceOf` to call `performTrick` on Trainable objects