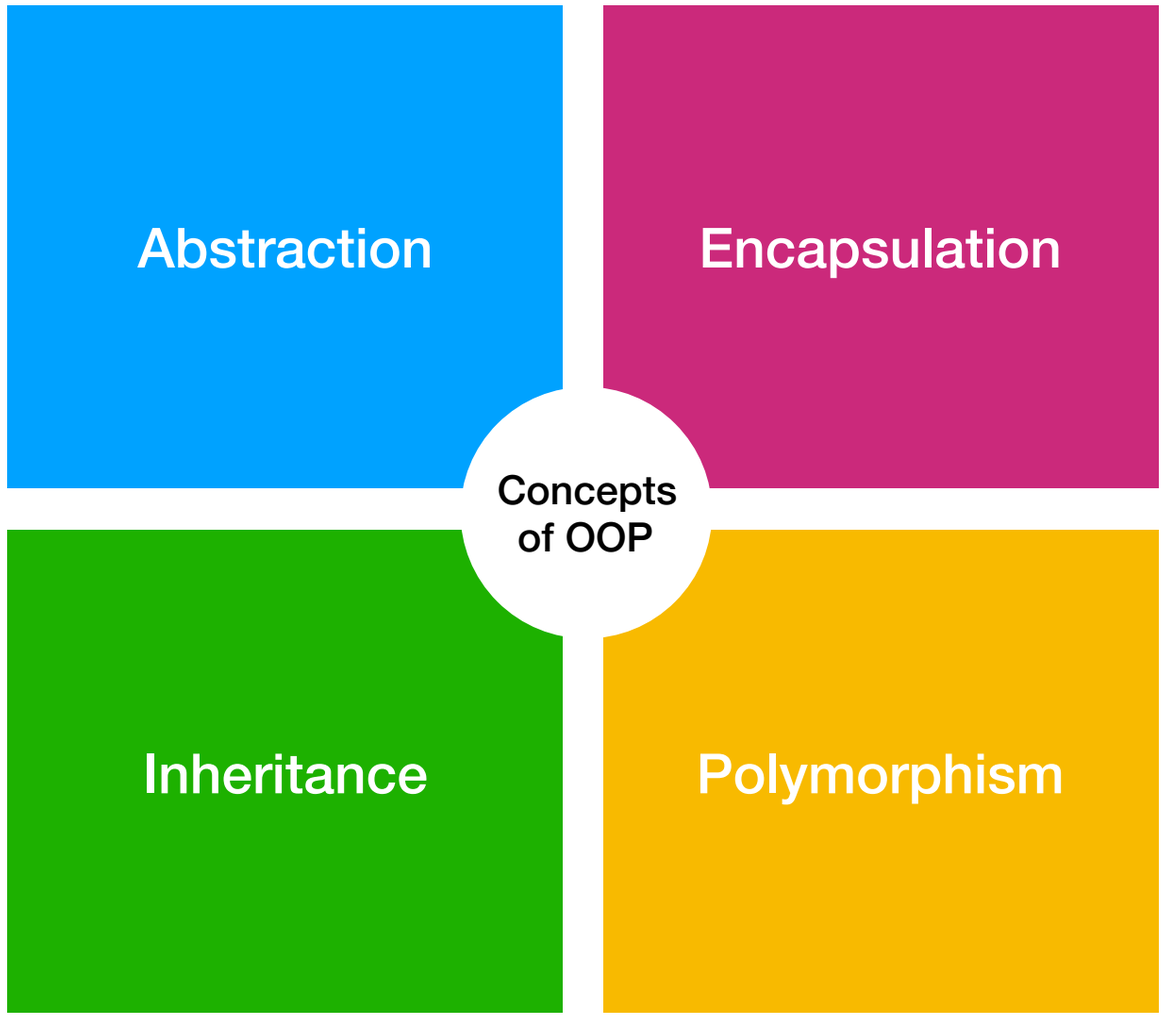
Part 2

Four Fundamental Concepts of OOP

Four Fundamental Concepts

Abstraction, Encapsulation, Inheritance, Polymorphism


Abstraction
Encapsulation
Concepts
of OOP
Inheritance
Polymorphism

Four Fundamental Concepts of OOP

OOP Concepts

- Encapsulation is the process of bundling data and operations on the data together in an entity.

- Abstraction is the process of exposing the essential details of an entity, while ignoring the irrelevant details, to reduce the complexity for the users.

- Inheritance is used to derive a new type from an existing type, thereby establishing a parent-child relationship.

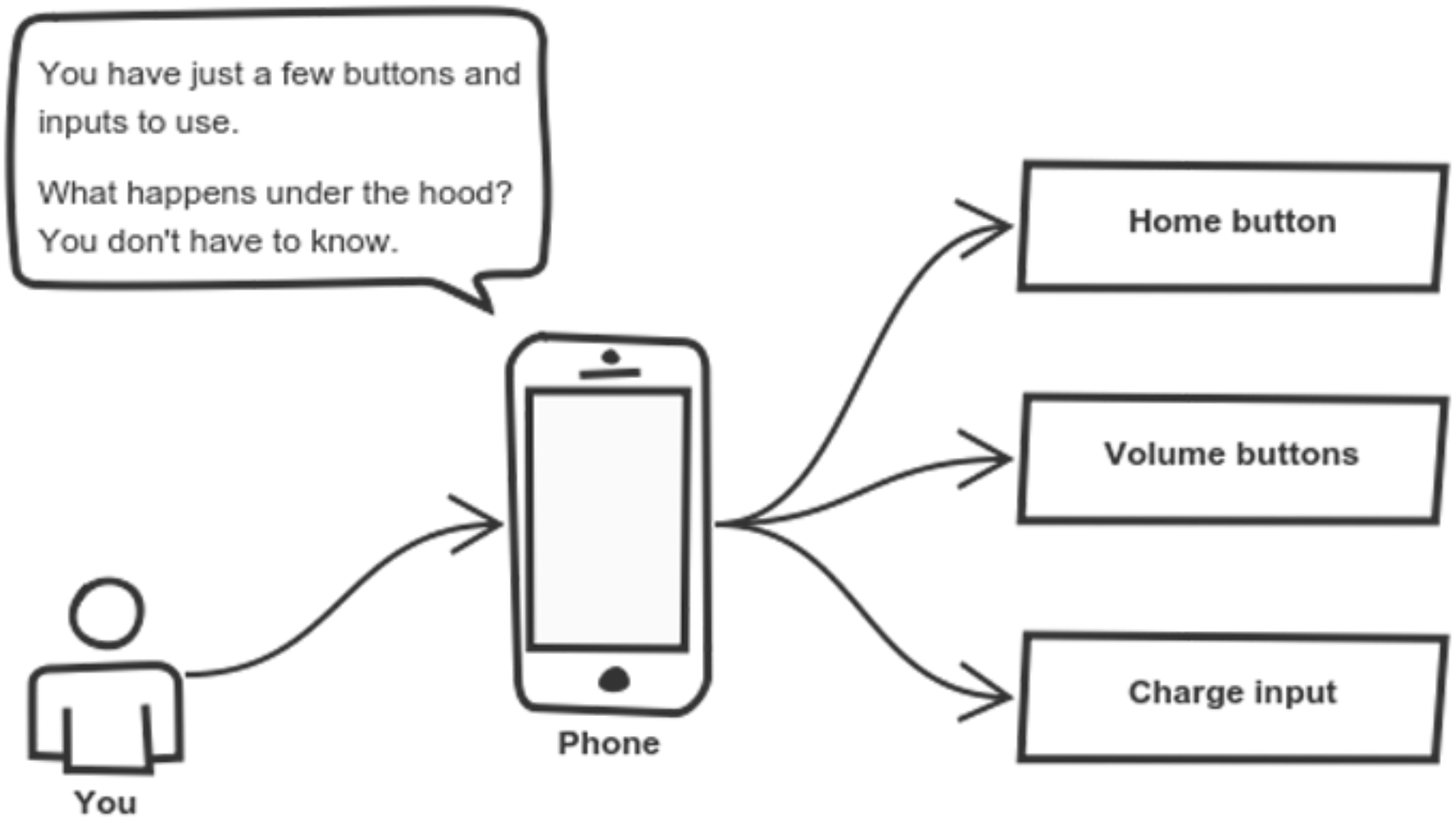
- Polymorphism lets an entity take on different meanings in different contexts.

Encapsulation

OOP Concepts

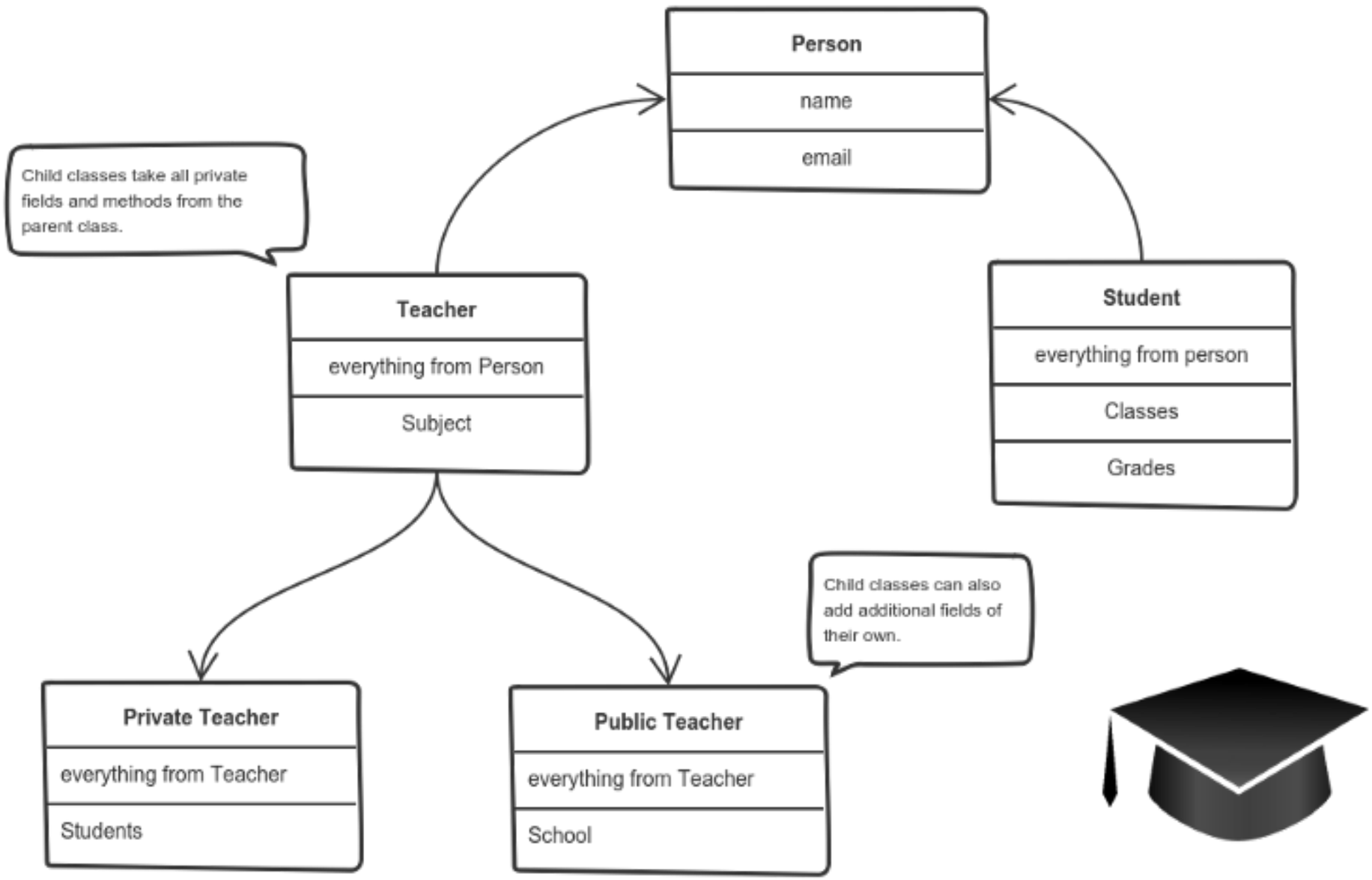
Abstraction

OOP Concepts



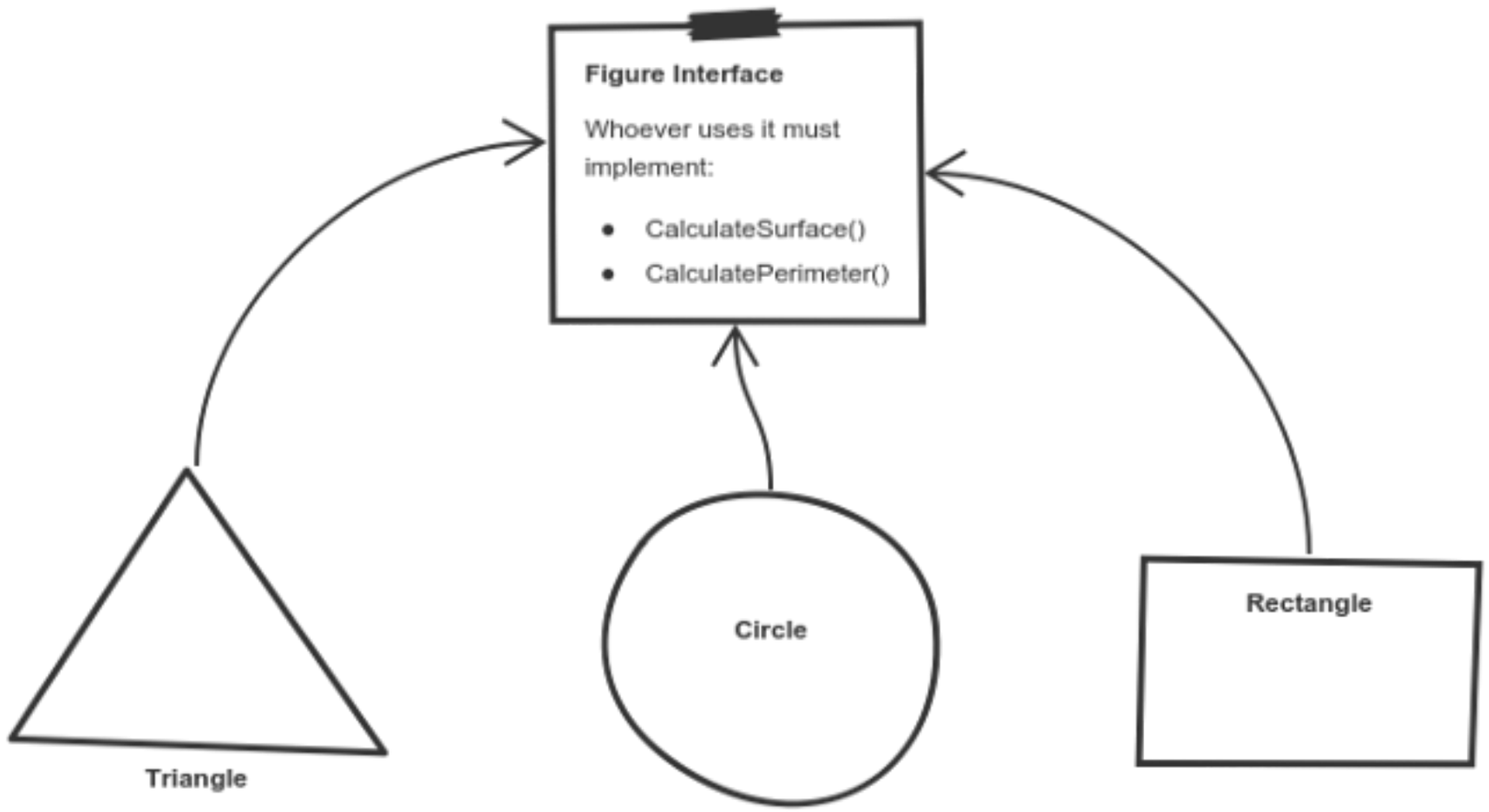
Inheritance

OOP Concepts



Polymorphism

OOP Concepts



Triangle, **Circle** and **Rectangle** inherit the Figure interface or abstract class.

They implement their own versions of **CalculateSurface()** and **CalculatePerimeter()**.

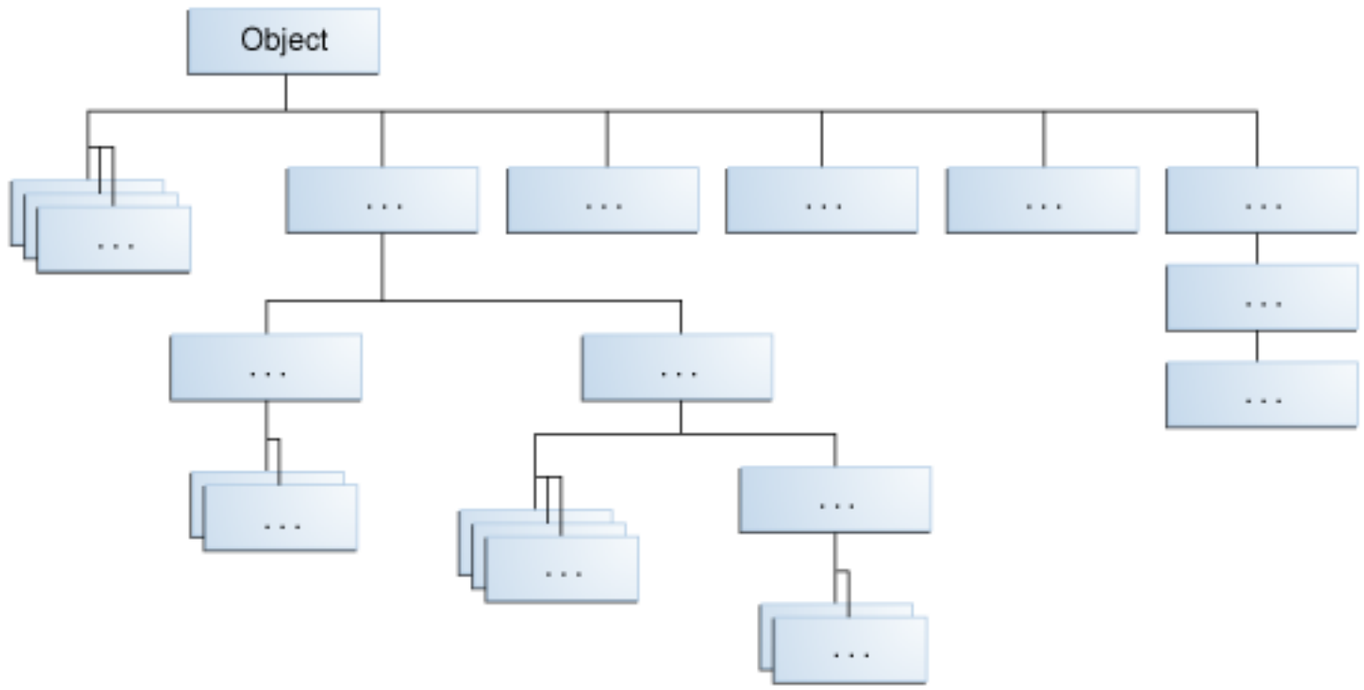
They can be used in a mixed collection of Figures.

Inheritance

Inheritance

OOP Concepts

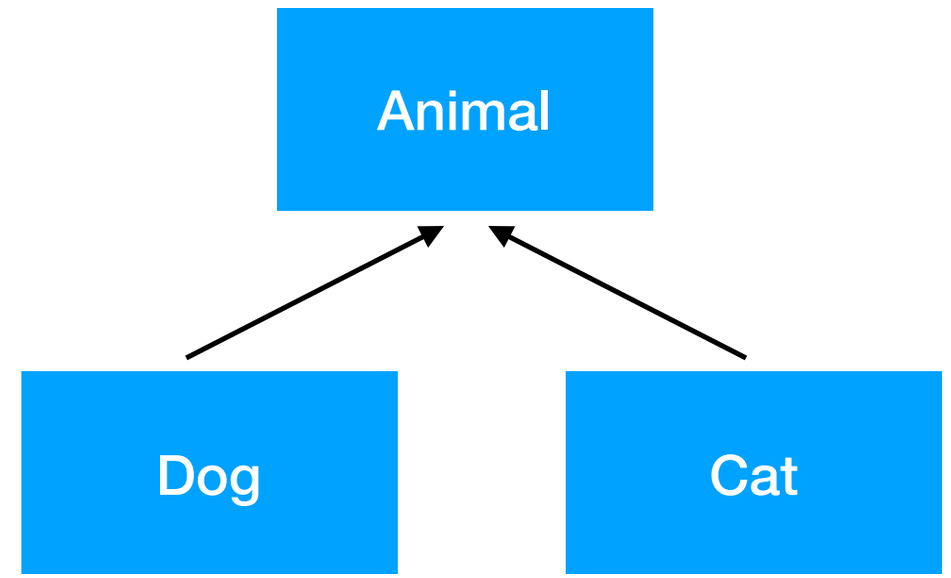
- OOP Allows you to
  - Define new classes from existing classes
  - Reuse fields and methods
  - Avoid redundancy



Inheritance

OOP Concepts

- is-a relationship
- superclass and subclass
- extends keyword
- super
- overriding methods



is-a relationship

Inheritance

Inheritance is an **is-a** relationship

Inheritance is used only if an **is-a** relationship is present between two classes

- A *car* is a *vehicle*
- An *orange* is a *fruit*
- A *surgeon* is a *doctor*
- A *dog* is an *animal*

is-a relationship

Inheritance

A Dog **is-a**n Animal.

So how do we make the relationship?

Animal

```
class Animal() {

}
```

Dog

```
class Dog() {

}
```

is-a relationship

Inheritance

Animal is a **super class** (parent class or base class)

Dog is a **sub class** (child class or derived class)

Animal

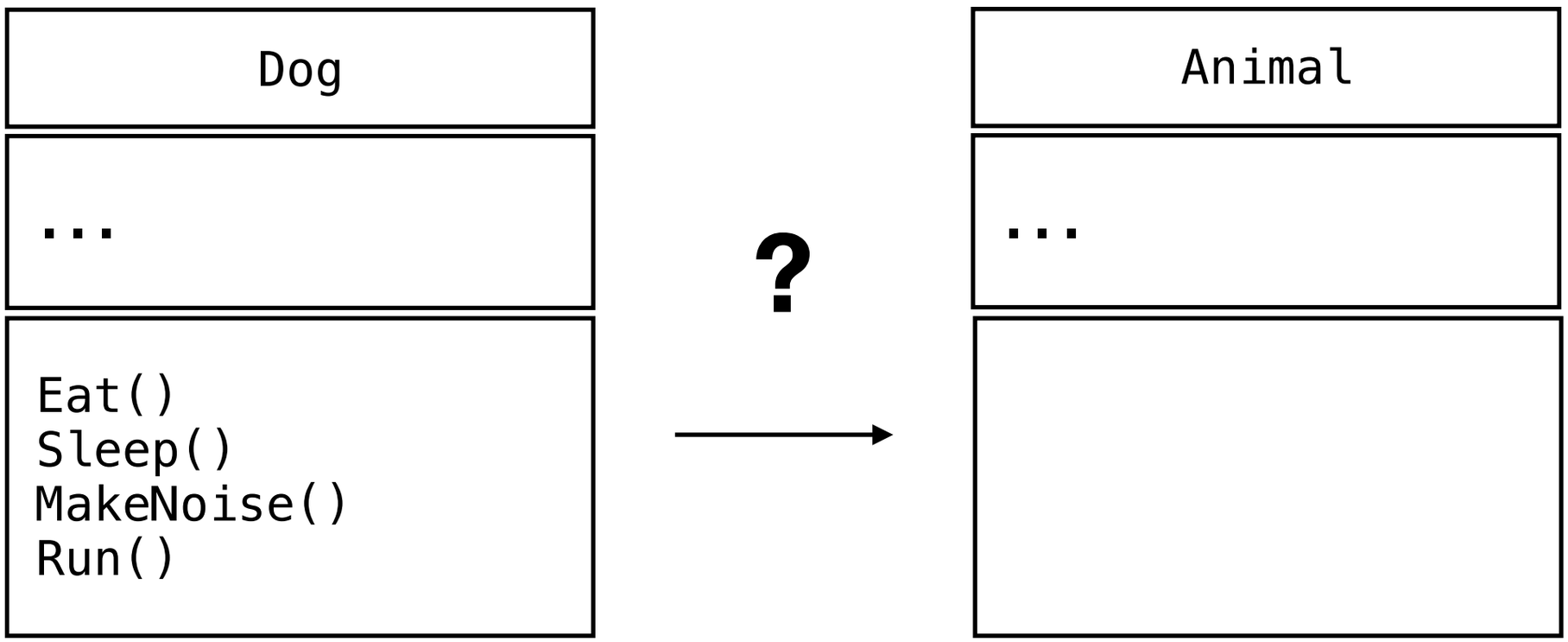
```
class Animal() {

}
```

Dog

```
class Dog() extends Animal {

}
```

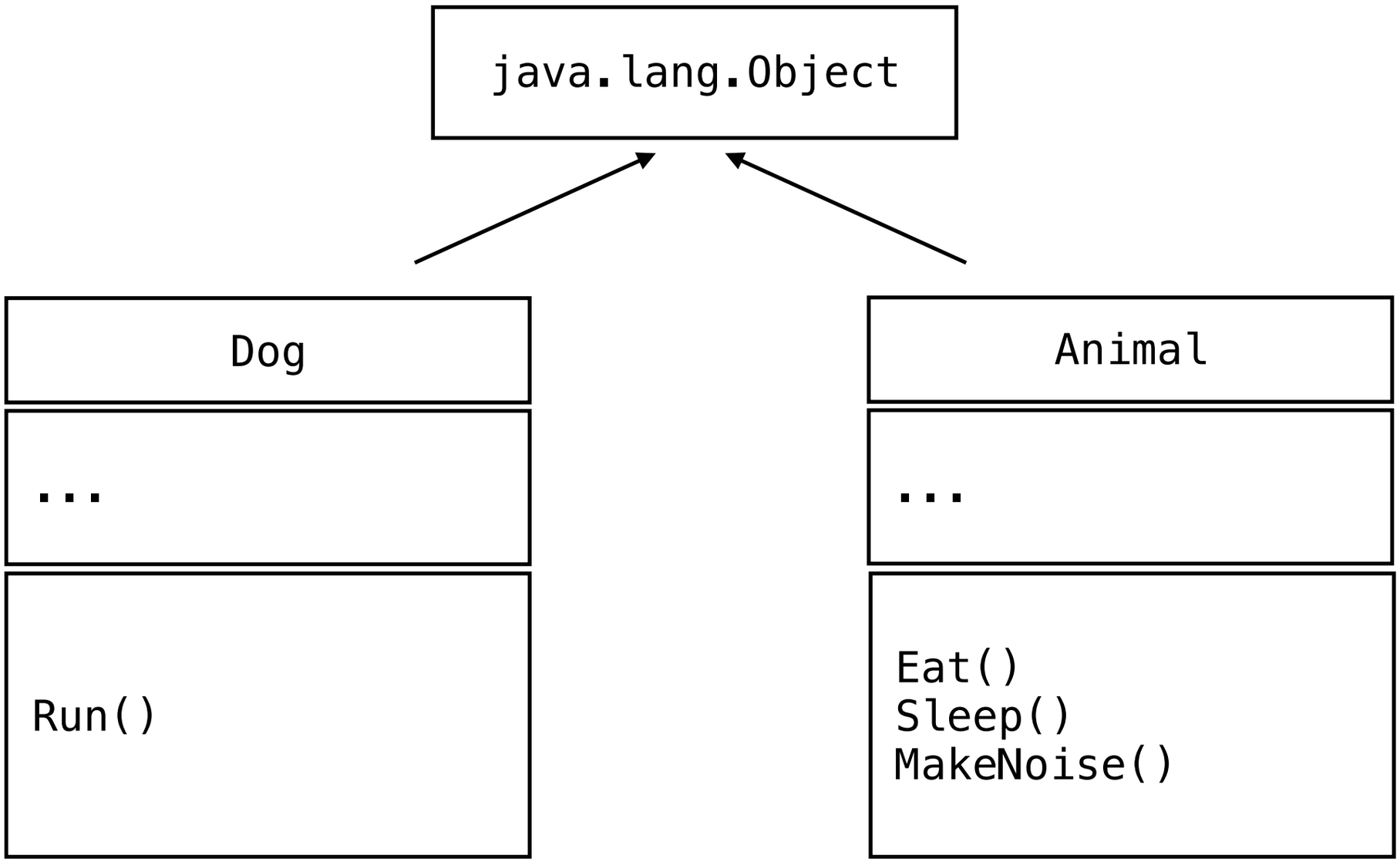
Common Behaviors

Inheritance



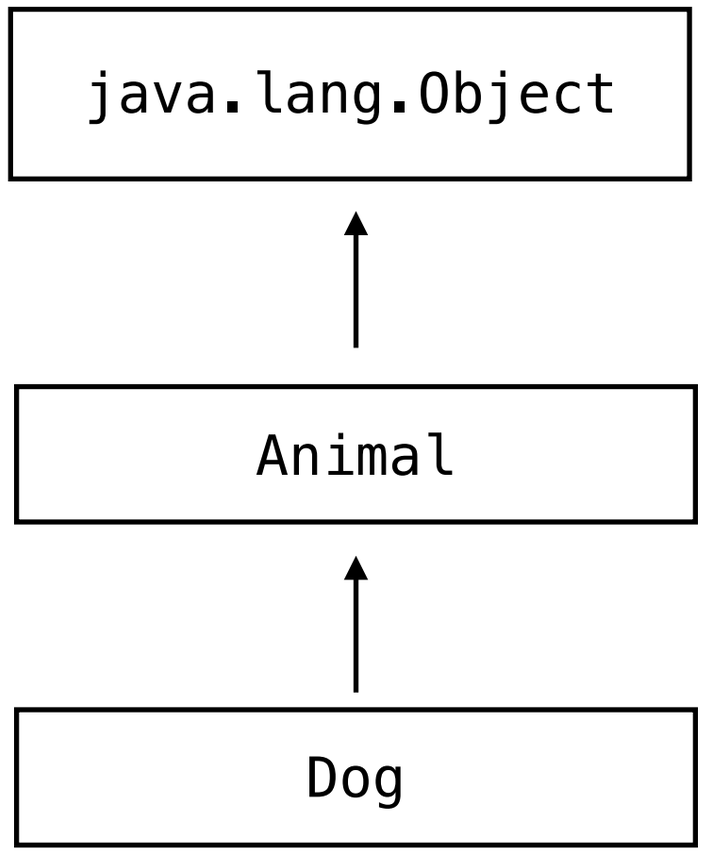
Inherit methods

Inheritance



Dog Inheritance Tree

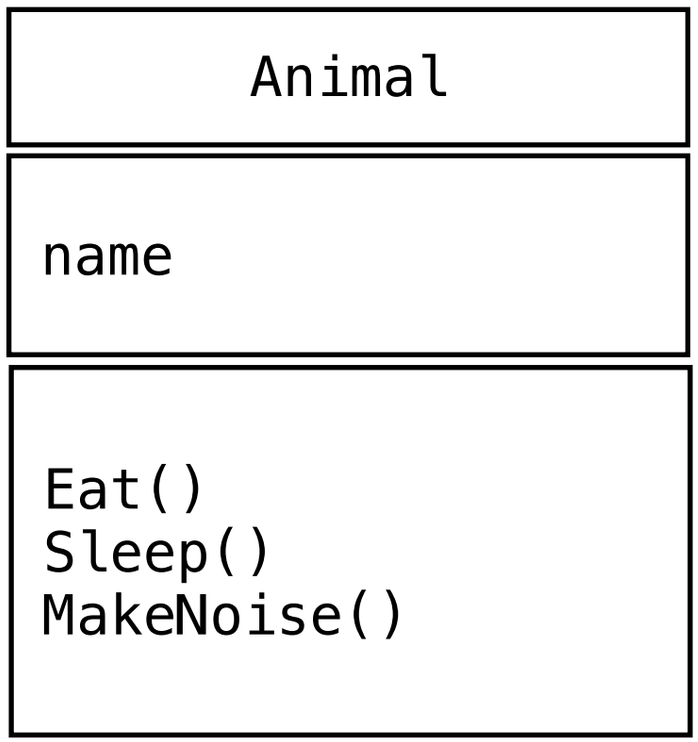
Inheritance



Inherit Member Variables

Inheritance

- Add a member variable to the Animal class
- name
- Dog inherits name



Coding Exercise

Inheritance

1. Move the methods from the Dog class to the Animal class so that Dog can now inherit them

2. Inside each of the four methods add a print statement to display:

    1. Animals eat…

    2. Animals sleep…

    3. Animals make noise…

    4. Dogs run…

3. Add the “name” member variable to the Animal class

4. Create com.aim.animals.AppInheritance1.java

    1. Create an instance of Dog

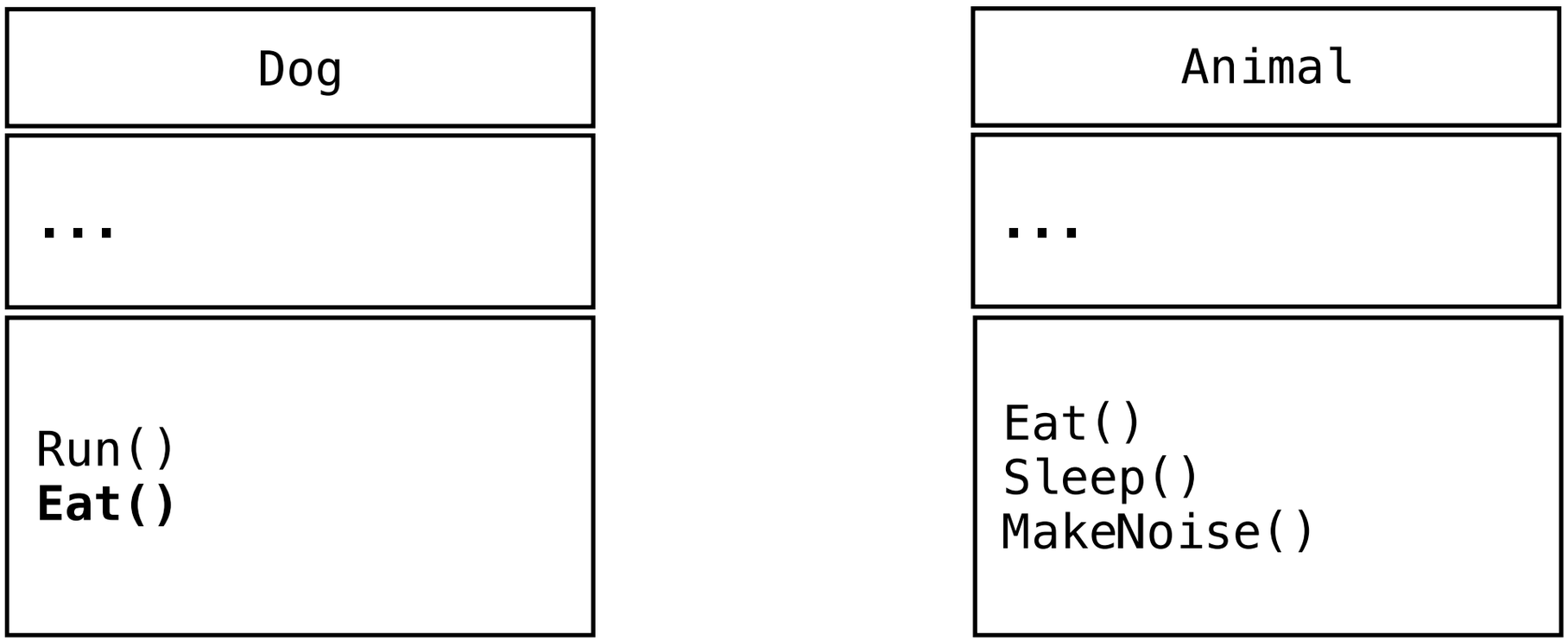
    2. Set the name of the Dog

    3. Call each of the four behavior methods available to the dog.

Override Methods

Override Methods

Inheritance



Override Methods Rules

Inheritance

- Both the superclass and the subclass **must** have the
  - same method name
  - same return type
  - same parameter list.
- We cannot override the method declared as **final** and **static**

super keyword

Inheritance

```
class Dog extends Animal {
   ...
   public void makeNoise() {
      super.makeNoise();
      System.out.println(“Bow wow wow!”);
   }
   ...
}
```

Coding Exercise

Inheritance

1. Implement the eat() method inside of the Dog class and print out “Dogs eat dog food”.

2. Add the final keyword to the sleep() method in Animal class

3. Attempt to implement the sleep() method in the Dog class

4. Implement the makeNoise() method in the Dog class.

    1. Add the super method to call the Animal makeNoise() method

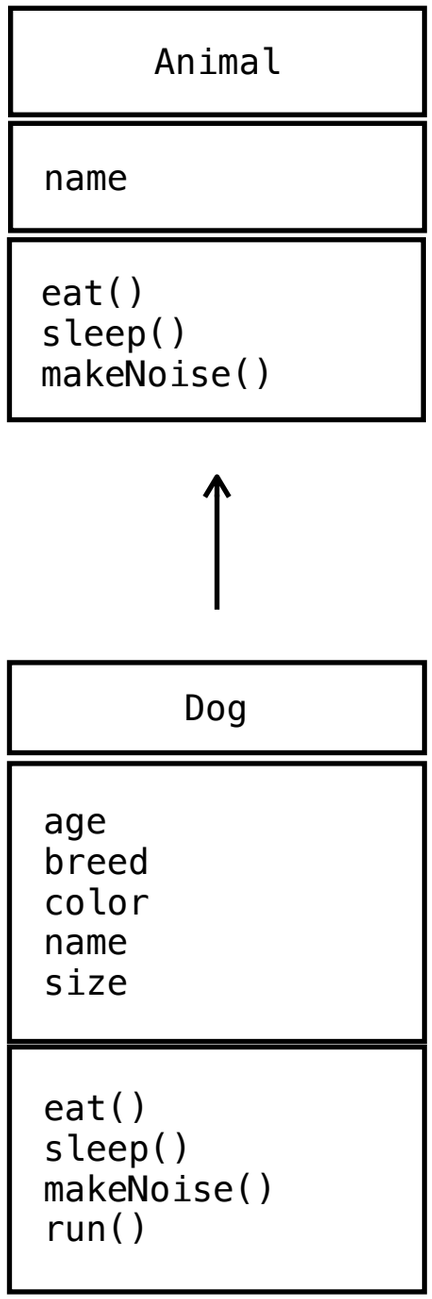
    2. Add print statement “Bow wow wow!”

5. Run AppInheritance1.java

6. What is your output?

UML Class Diagram

Blueprint for an object



Constructors

Why use inheritance?

Abstraction

Abstraction

OOP Concepts

- Abstraction is a general concept formed by extracting common features from specific examples or the act of withdrawing or removing something **unnecessary**.

- You can use abstraction using **Interfaces** and **Abstract** classes

- Abstraction solves the problem at **Design** time

- For simplicity, abstraction means hiding implementation using Abstract classes and Interfaces

- Partial abstraction (0-100%) can be achieved with abstract classes

- Total abstraction (100%) can be achieved with interfaces

Abstract Class

Defining an Abstract Class

Abstract Class

- Similar to interfaces
- May or may not contain abstract methods
- Cannot be instantiated
- Inheriting class must implement abstract methods

```
public abstract class Animal {

    public abstract String makeNoise();

}


public class Dog extends Animal {

    public String makeNoise() {
        return "Woof woof”;
    }
}
```

Example

Abstraction

```
package com.aim.animals;

public abstract class Animal {

    public abstract void makeNoise();

    public abstract String toString();

    public void eat() {
        System.out.println("Animals eat...");
    }

    public final void sleep() {
        System.out.println("Animals sleep...");
    }

}
```

Interface

Defining an Interface

Interface

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations (implicitly public static final)
    String SOME_CONSTANT = “I am a constant”;

    // method signatures
    void doSomething(int i, double d);

    int doSomethingElse(String s);
}


public interface Pet {
    void play();
}

public interface Edible {
    void howToEat();
}
```

Abstract vs Interface

When to use an abstract class

Abstraction

An abstract class is a good choice if we are using the inheritance concept since it provides a common base class implementation to derived classes.

An abstract class is also good if we want to declare non-public members. In an interface, all methods must be public.

If we want to add new methods in the future, then an abstract class is a better choice. Because if we add new methods to an interface, then all of the classes that already implemented that interface will have to be changed to implement the new methods.

If we want to create multiple versions of our component, create an abstract class. Abstract classes provide a simple and easy way to version our components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, we must create a whole new interface.

Abstract classes have the advantage of allowing better forward compatibility. Once clients use an interface, we cannot change it; if they use an abstract class, we can still add behavior without breaking the existing code.

If we want to provide common, implemented functionality among all implementations of our component, use an abstract class. Abstract classes allow us to partially implement our class, whereas interfaces contain no implementation for any members.

When to use an interface

Abstraction

If the functionality we are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing a common functionality to unrelated classes.

Interfaces are a good choice when we think that the API will not change for a while.

Interfaces are also good when we want to have something similar to multiple inheritances since we can implement multiple interfaces.

If we are designing small, concise bits of functionality, use interfaces. If we are designing large functional units, use an abstract class.

Abstract vs Interfaces Examples

Abstraction

- “strong” is-a relationship (Abstract Classes)
  - parent-child relationship
  - The Gregorian calendar is a calendar, so the relationship between the java.util.GregorianCalendar and the abstract class java.util.Calendar is modeled using class inheritance.
- “weak” is-a relationship (Interface)
  - is-kind-of relationship
  - object possesses a certain property
  - All Strings are comparable, so the String class implements the Comparable interface

Encapsulation

Encapsulation

OOP Concepts

- Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
- You can implement encapsulation using **access modifiers** (public, protected & private)
- Encapsulation solves the problem at **Implementation** time
- For simplicity, encapsulation means hiding data using getters and setters

Encapsulation

OOP Concepts

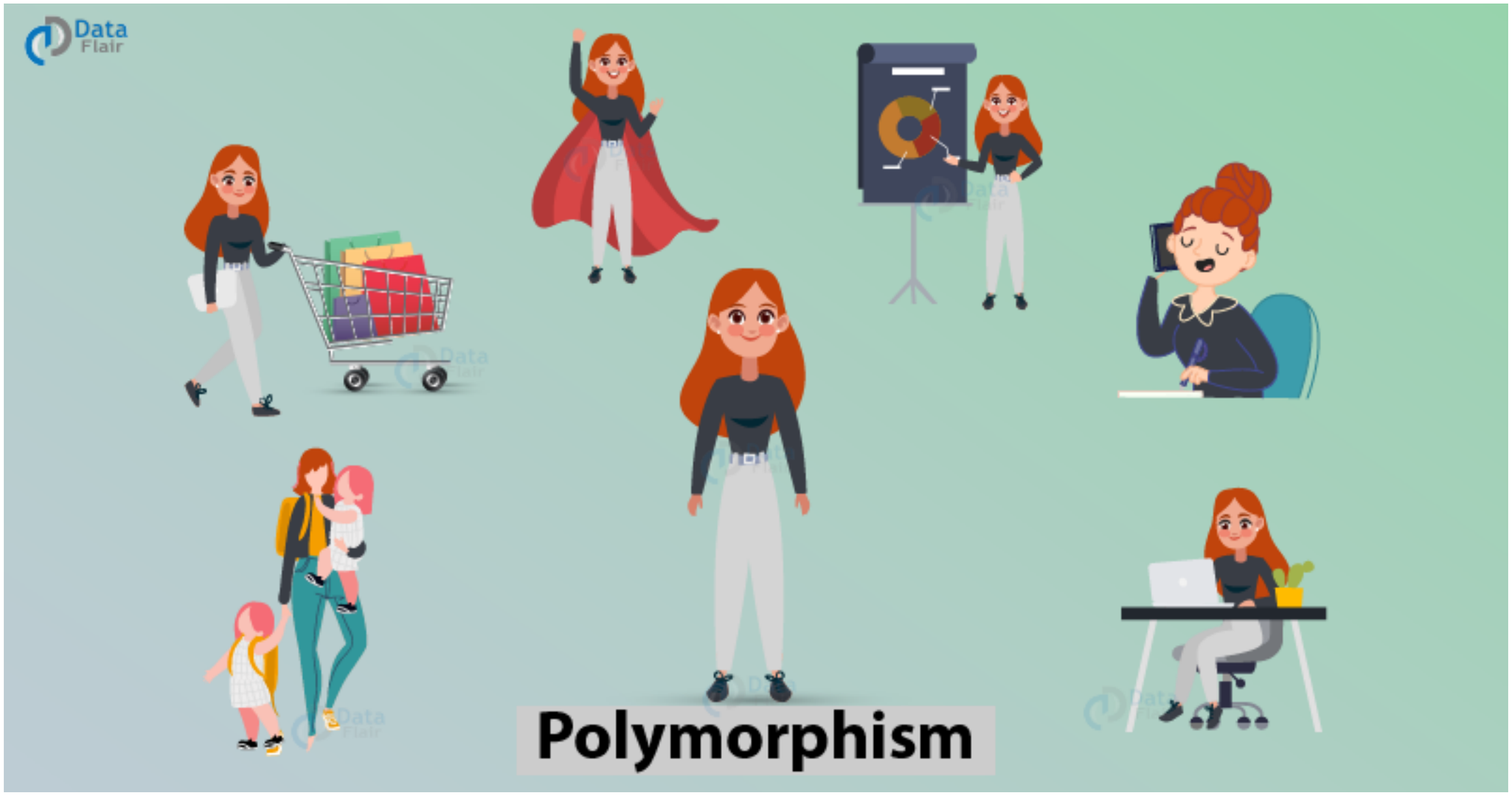
Encapsulation is one of the four fundamental OOP concepts.

```
public class Dog {

    private String breed;
    private String size;
    private short age;
    private String color;

    public String getBreed() {
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }

}
```

Comparison

Abstraction vs Encapsulation

Abstraction	Encapsulation
Abstraction is a general concept formed by extracting common features from specific examples or the act of withdrawing or removing something **unnecessary**.	Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interfaces** and **Abstract** classes	You can implement encapsulation using **Access modifiers** (public, protected & private)
Abstraction solves the problem at **Design** time	Encapsulation solves the problem at **Implementation** time
For simplicity, abstraction means hiding implementation using Abstract classes and Interfaces	For simplicity, encapsulation means hiding data using getters and setters


Data Flair
Polymorphism

Polymorphism

OOP Concept

- Ability to take on many different forms
- provides a way to use a class like its parent

```
// + operator used to perform addition
int a = 5;
int b = 6;
int sum = a + b;                //  Output = 11

// + operator used to perform string concatenation
String firstName = "abc ";
String lastName = "xyz";
name = firstName + lastName;  //  Output = abc xyz
```

Types of Polymorphism

Polymorphism

- Static Polymorphism
  - Compile time
    - Method Overloading - same method name, different parms
- Dynamic Polymorphism
  - Run time
  - Method Overriding - same method, different classes

Method Overloading (Static Polymorphism)

Polymorphism

- Static (compile time) polymorphism
- Two or more methods have the same name but different parameters

```
class Adder {

   static int add(int a, int b) {
      return a + b;
   }

   static double add( double a, double b) {
      return a + b;
   }

   public static void main(String args[]) {
      System.out.println(Adder.add(11,11));
      System.out.println(Adder.add(12.3,12.6));
   }
}
```

Method Overriding (Dynamic Polymorphism)

Polymorphism

- Dynamic (runtime) polymorphism
- The same method is implemented in related classes
- Overrides method of a superclass at runtime

```
public Class Animal {

   public void makeNoise() {
      System.out.println(“Animals make noise...”);
   }
}

public Class Dog extends Animal {
   void makeNoise() {
      System.out.println(“Woof woof”);
   }

   public static void main(String[] args) {
      Animal animal = new Dog();
      animal.makeNoise(); // Output “Woof woof”
   }
}
```

Update Dog

Add Inheritance to Dog

Add Some Polymorphism

Coding Exercise

OOP Concepts

1. Change the Dog class to use Encapsulation on the member variables

2. Change Animal to an abstract class

3. Write a program to create an Animal object with a reference to a Dog
    1. Animal animal = new Dog();
    2. Call makeNoise() on the animal object
    3. Run the program. What is your output?

4. Change the makeNoise() method to an abstract method
    1. Run the program. What is your output?

Coding Exercise 2

OOP Concepts

1. Create more animals! And implement some member variables and methods
    1. Duck, Chicken, Cow, Bird
2. Create an interface called Edible
    1. add an abstract method howToEat();
    2. implement this interface on appropriate animals
3. Create an interface called Pet
    1. add an abstract method play();
    2. implement this interface on appropriate animals
4. Create a program