Object-Oriented Programming

SPRIHA JOSHI

Objectives for Today

Understanding Objects and Classes

Understanding what OOP programming is and why it is useful

Understanding the four pillars of OOP and the differences among them:

- Polymorphism
- Inheritance
- Abstraction
- Encapsulation

Understanding the difference between an Abstract Class and Interface

Understanding Inheritance and Interfaces

Understanding the different Access Modifiers and Types of Methods

Object-Oriented Programming?

What is OOP?: It's a way of programming that builds around "objects" — these are bundles of data and related behaviors, much like real-world objects.

Why Use OOP?: It helps to manage complex software by organizing it into digestible chunks, making it easier to use, fix, and improve over time.

1. Code re-use
2. Modularity
3. Flexibility
4. Readability
5. Security

Starting Problem

We have to write a program for a shelter. The program has to allow the owner/user to keep a track of all pets that are currently in in their shelter, to add new pets, to remove pets from the inventory when adopted, etc.

```
public class Dog
{
     private String Myname;
     private String Mybreed;

     public void bark()
{
          System.out.println(“Woof Woof”);
     }
     public void setname(String name)
     {
          Myname = name;
     }

}
```

Classes and Objects

Intuition

Let’s say someone told you today that they bought a book. Before asking any questions, or them telling you more about the book … you already have an idea of the “book”

For example you know:

It could be a novel or a cookbook, or atlas or even a course textbook?

It has a title, an author, a publisher, price etc.

Could be a physical copy, paperback or hardback, could be an e-book?

If we now translate this back to the OOP paradigm:

This idea of the **“book”** is a class while the **specific book** that was purchased will be an **object.**

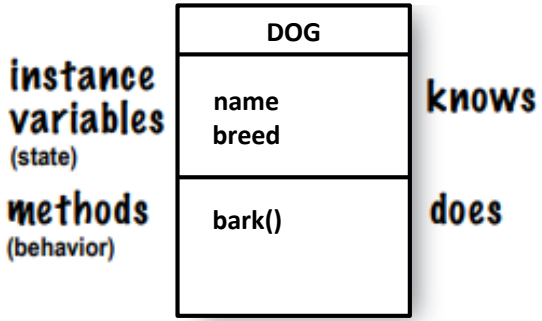
Classes and Objects

A class is a **blueprint** for an object. It tells the compiler **what** every object of that particular type must know and what it can do.

An object consists of data encapsulated with a set of methods which operate only on this data

The **data** of an object determine the **state** of the object => what the object **knows**

The **methods** associated with an object can change the state of the object, or provide info on the state of the object, or determine relationships of the object with other objects => what the can object can **do**



Classes and Objects

```
public class Dog
{

	private String Myname;

	private String Mybreed;


	public void bark()
{
	System.out.println(“Woof Woof”);
 }
 public void setname(String name)
 {
	Myname = name;
 }


}
```

Instantiation: `new` keyword

The `new` operator creates an object of a class using the class constructor and returns the object reference.

```
public class Shelter{
 public static void main(){
        Dog bob = new Dog();
        bob.breed = "Golden Retriever";
        bob.setname("Bob");
        bob.bark();
 }

}
```

The constructor method

- Special method with the **same name as the class**
- Automatically called when object is created with the `new` keyword
- Objective is to initializes the instance variables if any
- Class can have no constructor, one constructor or many
- When no constructor is specified by the coder, a constructor with no arguments is generated and variables are to the default values
  - Numerical values = 0 (or 0.0)
  - Boolean = false
  - Objects and reference type variables = `null`

Reference Type Variables

To manipulate the object, its reference has to be stored in a reference variable.

The type of the reference variable has to match the type of the object created.

Example:

```
Dog D1;
D1 = new Dog();

Cat C1;
C1 = D1 ; // Type Mismatch ERROR!
```

Be mindful: don’t overuse main

The main method should only be used:

- to test your real class
- to launch/start your Java application

```
public class Shelter{
 public static void main(){
       String dog = “Bob”;
       String  breed = “Golden Retriever”;
       System.out.println(“Woof Woof”);
  }
}
```

The pillars of OOP

Abstraction

Inheritance

Polymorphism

Encapsulation

Encapsulation

Intuition

By now, you all must have used String type to store textual data. Let's say you want to know if the given word is palindrome or not.

String x = "abba"

String y = "dancing queen"

While you come up with a solution, focus on the different String methods you are thinking of using …

x.length() , x.equals(y) => do you know how **exactly** these work? Do you really need to know?

Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods	Deprecated Methods

Modifier and Type	Method and Description
`char`	**`charAt`**`(int index)` Returns the `char` value at the specified index.
`int`	**`codePointAt`**`(int index)` Returns the character (Unicode code point) at the specified index.
`int`	**`codePointBefore`**`(int index)` Returns the character (Unicode code point) before the specified index.
`int`	**`codePointCount`**`(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this `String`.
`int`	**`compareTo`**`(String anotherString)` Compares two strings lexicographically.
`int`	**`compareToIgnoreCase`**`(String str)` Compares two strings lexicographically, ignoring case differences.
`String`	**`concat`**`(String str)` Concatenates the specified string to the end of this string.

What is Encapsulation?

How an object performs its duties is hidden from the outside world. This idea of separation of implementation and public interface is called Encapsulation.

Methods can be used without the knowledge of the inner workings.

Inner workings can be modified without impacting use (as long as the interface is unchanged).

A **public interface** is the set if all methods in the class along with their descriptions.

Why is it useful?

Makes it easier for users and fellow software developers to read and use the code.

It helps you protect your data and protect your right to modify your implementation.

It reduces system complexity and increases robustness by preventing external entities from depending on internal details.

Example: A class!

Types of Methods

Accessor: A method that queries the object without changing it's state.

- Often has a non-`void` return type
- Example: `getBreed()`

Mutator: A method that modifies an object's state.

- Example: `compute_age()`

Helper: Assists some other method in performing its task.

- Often declared as private so outside clients cannot call it

Access Modifiers

Public

- Any code anywhere can access the public methods/variables

Private

- Methods/variables within the same class can access the private component. Keep in mind it means private to the class, not private to the object.

Protected

- Methods/variables in the same package have access, EXCEPT it also allows subclasses outside the package to inherit the protected component.

Static variables and methods

Normally, every object of a particular class gets it's own copy instance variables and methods.

However, in some cases it is required that the class only has one copy of a variable across all it's objects. This would be a **static variable**.

Example: Let's say we want to assign a ID to all pets that are part of a shelter.

static pet_id = 0; as instance variable in Dog class

Static methods are methods that can be called on the Class directly without creating an object.

Example: Math.sqrt(x);

Example

```
public class Dog
{
     private String Myname;
     private String Mybreed;
     private static int pet_id = 0;

     public void bark()
     {
          System.out.println("Woof Woof");
     }
     public void setname(String name)
     {
          Myname = name;
     }

}
```

Going back to the Starting Problem

We have to write a program for a shelter. The program has to allow the owner/user to keep a track of all pets that are currently in in their shelter, to add new pets, to remove pets from the inventory when adopted, etc.

```
public class Dog
{
     private String Myname;
     private String Mybreed;
     private static int pet_id = 0;


     public void bark()
     {
          System.out.println(“Woof Woof”);
     }
     public void setname(String name)
     {
          Myname = name;
     }

}
```

Inheritance

Intuition

So far for our problem, we have created the following classes:

Shelter class with the main method

Dog, Cat, Snake, Hamster, Chicken Class

They all have the same variables and the same methods defined except the different noise methods (bark, meow, hisss, etc.)

It is because they are all animals. All animals share some traits like some physical characteristics, belonging to a breed, ability to make a sound, age, etc.

How can we make use of this information?

What is Inheritance?

An OOP principal that allows us to extend existing classes and adding methods and fields specific to the subclass.

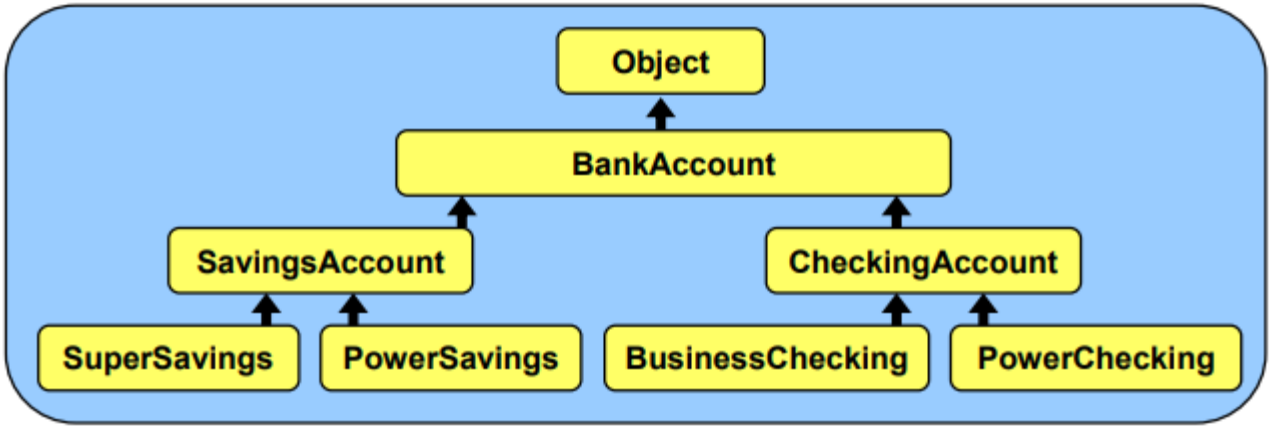
Creates Class hierarchies**:**

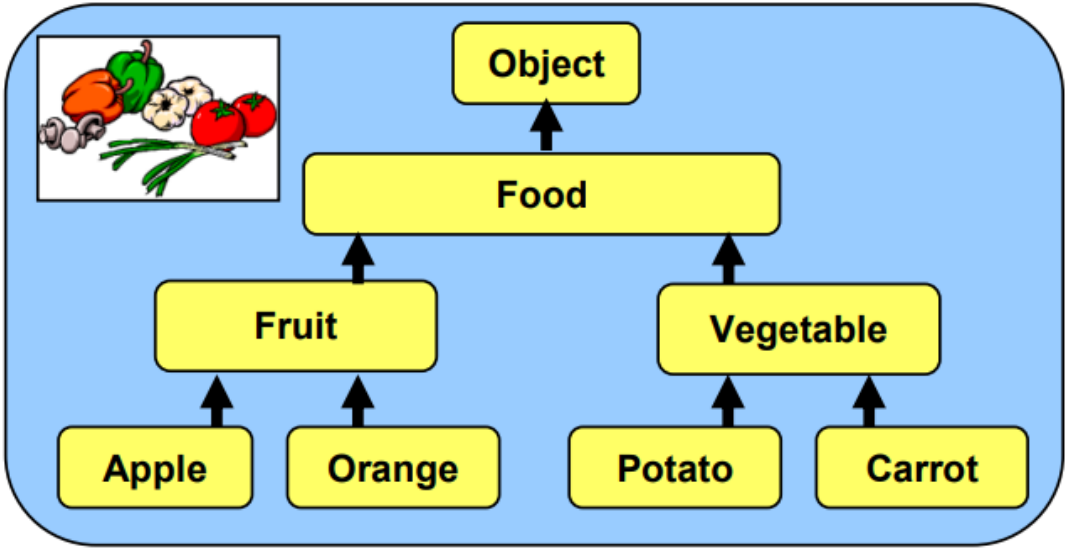
- **Superclass**: Parent class being extended. This is the more general class.
- **Subclass**: Child class that inherits behavior from superclass. Gets a copy of every field method from superclass. The more specialized class.

Is-a relationship: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.

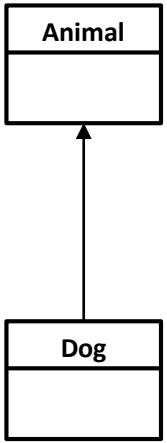
Use the keyword `extend` to create hierarchies and implement inheritance

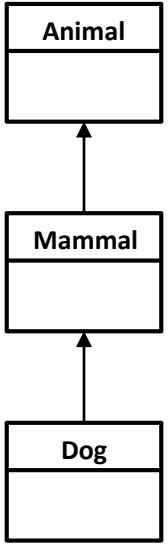
Examples of class heirarchies

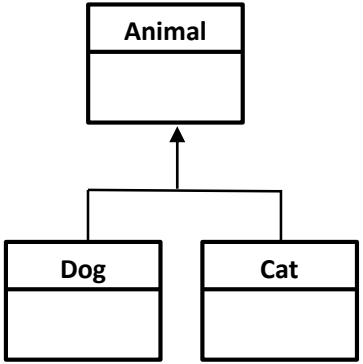


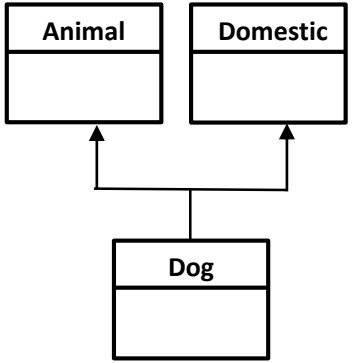

Object
Food
Fruit
Vegetable
Apple
Orange
Potato
Carrot

Types of Inheritance









Superclass and Subclass methods

When writing a subclass of superclass:

You can override methods of the superclass

- If you have a method with same name and signature then method from subclass overrides

You can inherit methods of the superclass

- If no same signature methods exist, then the superclass methods are inherited by the subclass

You can write new methods

- If you write new methods in the subclass, they can be called upon only suing the objects of the subclass

Constructors in Inheritance

The subclass constructor can call a superclass constructor using the keyword `super`.

The call has to be in the first line.

If the subclass constructor doesn't call the superclass constructor, the default constructor of the super-class is called implicitly.

Can also use the keyword super to call a method from the superclass that was overridden in the subclass.

The `this` keyword

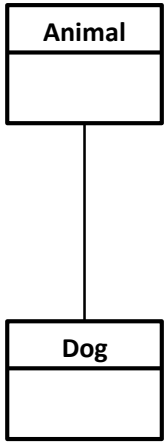
The keyword is used to refer to the current object of the class.

It's useful to differentiate local variables from global variables, to pass the reference of the current object, to call on the instance variables and methods.

`this()` can be used to call the constructor of the current class

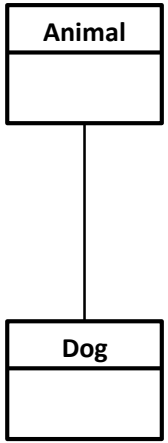
Example

```
public class Animal
{
     String Myname;
     String Mybreed;
     int age;

     public setName(String name) {…}
     public setBreed(String name) {…}
     public setAge(String name) {…}


}
```



Example

```
public class Dog extends Animal
{
     public void bark()
     {
          System.out.println("Woof Woof");
     }
}

public class Shelter{
  public static void main(){
      Dog bob = new Dog();
      bob.breed = "Golden Retriever";
      bob.setName("Bob");
      bob.bark();
   }
}
```



Polymorphism

What is Polymorphism?

An OOP principal that allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

It is the ability of an object to take many forms.

In Java, this can be achieved in two ways:

- Method overloading
- Method overriding

Method Overriding

When a child class overrides i.e. provides a different implementation of a method that it inherits from the parent class.

```
public class Animal
{
    String Myname;
    String Mybreed;
    abstract int age;

    public void setName(String name) {…}
    public void setBreed(String name) {…}
    public voidsetAge(String name) {…}

    public void eat(){
    System.out.println("I eat food");

    }
}
```

```
public class Dog
{
    @override
    public void eat(){
    System.out.println("I eat dog food");

    }

}
```

Method Overloading

Methods with same name but different parameters.

```
public class Dog {
private String myName;
private String myBreed;
private int myAge;

public Dog() {…}
public Dog(String name) {…}
public Dog(String name, String breed) {…}
public Dog(String name, String breed, int age) {…}

}
```

Abstraction

What is Abstraction?

It is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

Abstraction simplifies complex reality by modeling classes based on the essential characteristics relevant to the context.

Effective abstraction helps manage complexity by reducing and isolating change impacts

In Java, this can be achieved in two ways: Abstract Classes and Interfaces

Abstract Class and methods

When there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly, you can force programmers to override the method .

An Abstract class is a class that cannot be instantiated.

An abstract method is a method whose implementation is not specified.

Example

```
public abstract class Animal
{
  String Myname;
  String Mybreed;
  abstract int age;

  public setName(String name) {…}
  public setBreed(String name) {…}
  public setAge(String name) {…}

  public abstract makeSound();
}
```

Interfaces

- Interface is a named collection of:
  - abstract methods (methods without implementations), and
  - constants declarations.
- All the methods are abstract; that is, they have a name, parameters, and a return type, but they don’t have an implementation
- All the methods are implicitly public
- All data declarations are implicitly constant declarations; i.e., public static final. This implies that instance fields cannot be declared in interfaces
- interfaces are not classes; i.e., they cannot be instantiated.
- To realize an interface, a class must implement all the methods that the interface requires.

Example

```
public interface Mammal
{
  public void makeNoise();
  public void setFur();

 }


public interface Reptile
{
  public String makeNoise();
  public void setScales();

 }
```

```
public class Snake extends Animal implements Reptile{

public void makeNoise(){
          System.out.println("Hisssss");
  }

public void setScales(){
          System.out.println("Colorful scales");
  }


}


public class Shelter{
  public static void main(){
          Snake rango = new Snake();
          rango = "Rosy Boa";
          rango("Rango");
          rango.makeNoise(); // "Hisssss"
}

}
```

Abstract Class and Interfaces

- Used to provide a base class for concrete subclasses to inherit from
- Is a class that cannot be instantiated
- Can contain both abstract and non-abstract methods and variables that can be public, protected, private, final/non-final and static/non-static.
- Represents the type of an object
- A class can inherit from only one Abstract class
- An abstract class can be extended using the keyword **“extends”**

- It is a set of methods that must be implemented by all classes that implement it.
- Interface is not a class
- Only contains abstract methods and constants that are public static final.
- Represents a set of behavior
- An interface can be implemented by many classes
- An interface can be implemented using the keyword **“implements”**

Summary of Object-Oriented Principles

OOP is based on key principles like Encapsulation, Abstraction, Inheritance, and Polymorphism, which work together to create flexible, maintainable, and scalable software.

Understanding and applying these principles allows for designing systems that are robust and adaptable to change.

Remember to consider the balance between complexity and benefit when applying these principles to ensure they provide value to your project.

Topics covered today

Understanding Objects and Classes

Understanding what OOP programming is and why it is useful

Understanding the four pillars of OOP and the differences among them:

- Polymorphism
- Inheritance
- Abstraction
- Encapsulation

Understanding the difference between an Abstract Class and Interface

Understanding Inheritance and Interfaces

Understanding the different Access Modifiers and Types of Methods