While folks are joining

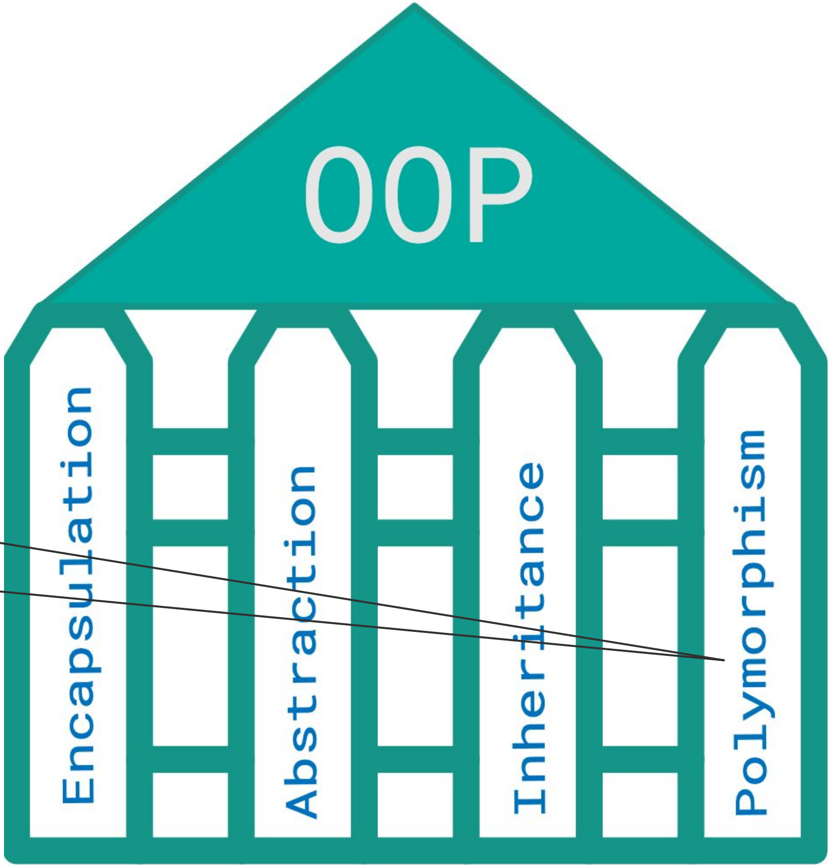
Get you laptops ready and login to your **replit** accounts.

We will be coding away in the session!

Crio Sprint : DEV-OOP-1

Session 4

Four Pillars of OOP



Will be Discussed Today!

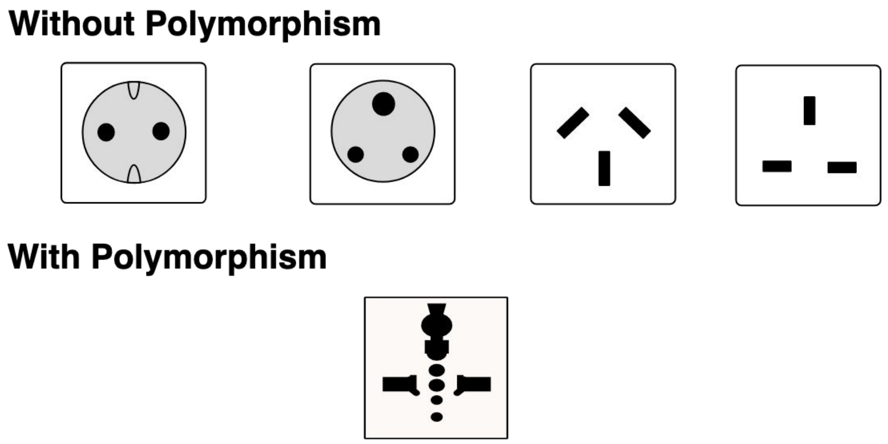
Why Polymorphism? - Scenario #1 Electric Socket

Ever done this while travelling?

You don't want to pack all these in your travel bag either!

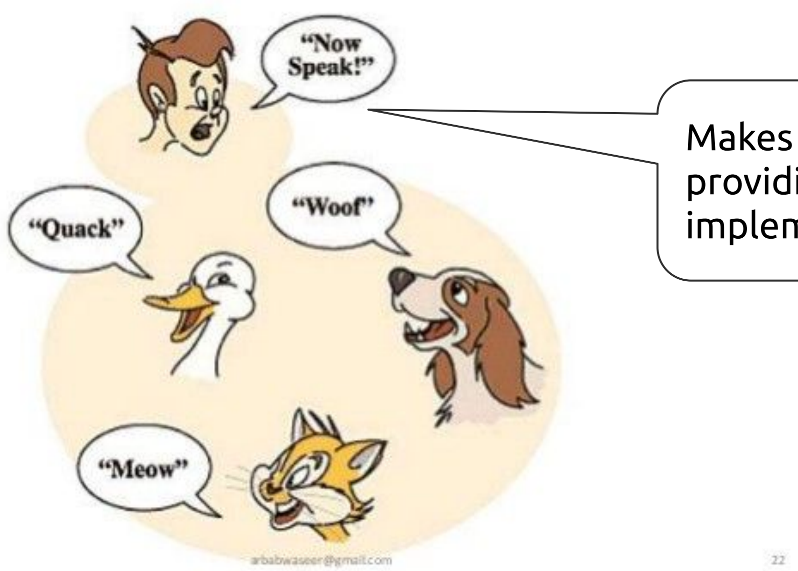
Why Polymorphism? - Scenario #1 Electric Socket

Wouldn’t it will be better if we had sockets that could accept many different types of plugs.



What is Polymorphism ?

- The ability for an object or function to take **many forms.**
- Helps reduce complexity and write reusable code.



Makes your code flexible by providing multiple ways to implement similar functionality.

Types of Polymorphism

- Compile Time Polymorphism
  - Method Overloading (Static Binding)
- Runtime Polymorphism
  - Method Overriding (Dynamic Binding)

Activity 1 - Addition

- Perform the addition of the given numbers. But user can enter any number of arguments.
- Possible Solution:
  - **addTwo(int, int)** method for two parameters
  - **addThree(int,int,int)** for three parameters
  - So on.
- What’s the problem with the above technique?
  - Difficult to understand the behaviour of the method due to strange naming convention.
  - Difficult to track how many such methods are performing addition in the class due to different names.
- Can we avoid this problem?
  - Yes. Method Overloading.

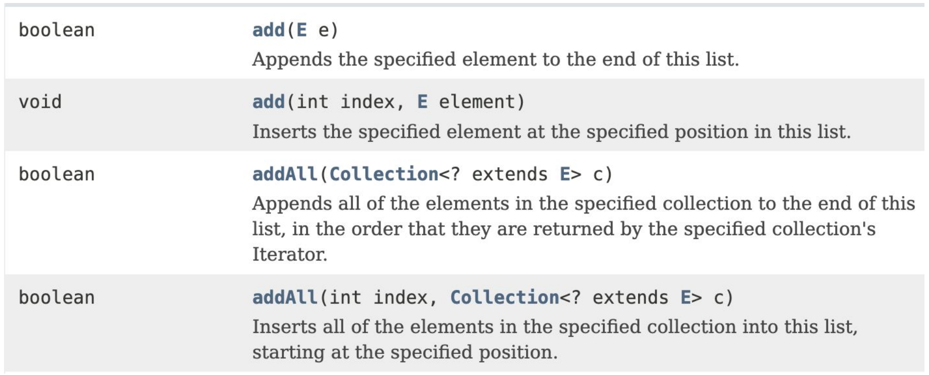
Method Overloading

- What is Method Overloading?
  - Multiple methods having the **same name but difference in parameters**.
  - A class can **hold several methods** having the **same name.**
- **Three ways to overload methods:**
  - By changing the number of arguments/parameters.
  - By changing the data type of arguments.
  - By changing the Order of arguments.
- **Solution for Addition Activity**
      - **addition(int, int)**
      - **addition(int,int,int)**

1. By Changing the number of arguments / parameters

```
class SimpleCalculator
{
   int add(int a, int b)
   {
      return a+b;
   }
   int  add(int a, int b, int c)
   {
      return a+b+c;
   }
}
public class Demo
{
  public static void main(String args[])
  {
     SimpleCalculator obj = new SimpleCalculator();
     System.out.println(obj.add(10, 20));
     System.out.println(obj.add(10, 20, 30));
  }
}
```

- In Java's ArrayList class, we can see some of the overloaded methods below:-

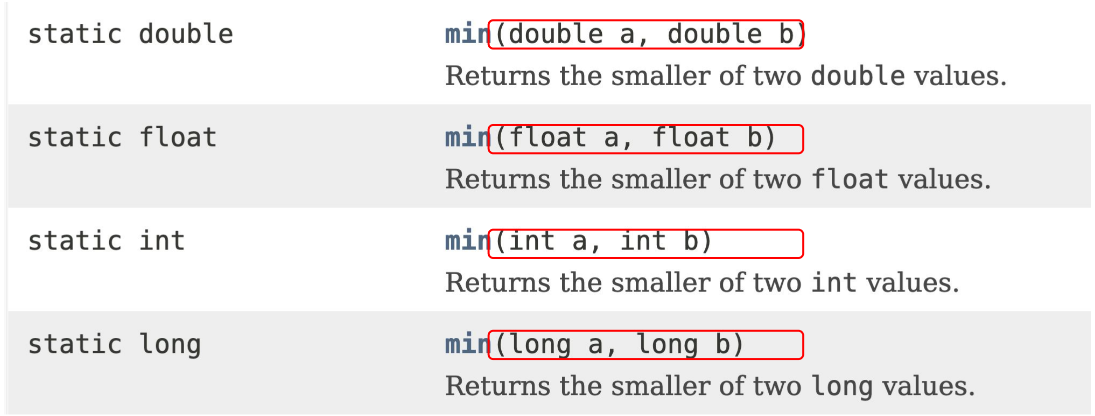


`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.

- Click on above link to explore more such methods.

Activity 2 - By changing the data type of arguments

- In Java's Math class, you will find many examples of overloaded methods.
- min() is overloaded with different data types.



static double	min(double a, double b) Returns the smaller of two double values.
static float	min(float a, float b) Returns the smaller of two float values.
static int	min(int a, int b) Returns the smaller of two int values.
static long	min(long a, long b) Returns the smaller of two long values.

3. By changing the Order of Arguments

```
class Student
{
   public void show(String name, int age)
   {
      System.out.println("Name of person = "+name+ " and age is = "+ age);
   }
   public void show(int age, String name)
   {
      System.out.println("Name of person = "+name+ " and age is = "+ age);
   }
   public static void main (String [] args)
   {
     Student s = new Student();
     // If student providing parameter of String and int  type then first method called
     s.show("Ram", 25);
     // If student providing parameter of int and String type then second method called
     s.show(25, "Ram");
   }
}
```

Curious Cats

- Why is method overloading by changing the return type of a method, not possible?
    - Compiler only checks method signature for duplication and not the return type.
- When do we use Static Polymorphism?

Summary - Method Overloading

- When a class has two or more than two methods which are having the **same name but different types of order or number of parameters,** it is known as Method Overloading.
- Method overloading is resolved during **compile time.**
- Three ways to overload methods:
    - By changing the **number of arguments/parameters.**
    - By changing the **data type of arguments.**
    - By changing the **Order of arguments.**
- Changing only return type with same parameters of method is not Method Overloading.

5 minute break

Activity 3 - Bank Interest Rates

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI and ICICI banks could provide 8% and 7% rate of interest.

What would be the output from this program?

- Interest rate will be printed as 5 for every bank.

What can we do to fix it?

- Create new methods in each bank which will give the expected rate.

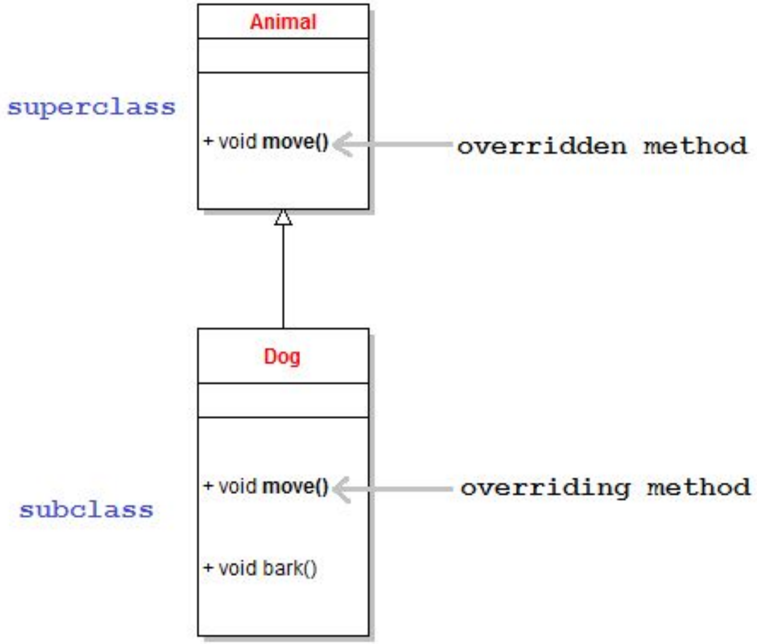
Can we use the same method name - *getRateOfInterest* in each bank subclass?

- Yes. Method Overriding.

```
class Bank{
  int getRateOfInterest(){return 5;}
}
//Creating child classes.
class SBI extends Bank{
}
class ICICI extends Bank{
}

class Test{
  public static void main(String args[]){
    SBI s=new SBI();
    ICICI i=new ICICI();
    System.out.println("SBI Rate of 
Interest: "+s.getRateOfInterest());
    System.out.println("ICICI Rate of 
Interest: "+i.getRateOfInterest());
  }
}
```

Method Overriding



```
class Bank{
 //Overridden Method
 int getRateOfInterest(){return 5;}
}
//Creating child classes
class SBI extends Bank{
 //Overriding Method
 @Override
  int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
  //Overriding Method
 @Override
  int getRateOfInterest(){return 7;}
}

class Test{
 public static void main(String args[]){
   SBI s=new SBI();
   ICICI i=new ICICI();
   System.out.println("SBI Rate of Interest"+ s.getRateOfInterest());
   System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
  }
}
```

Activity #4 - Find a Bug

```
public class Calculator{

    public int add(int a, int b){
        return a + b;
    }
    public int substract(int a, int b){
        return a - b;
    }
    public int multiply(int a, int b){
        return a * b;
    }
    public int divide(int a, int b){
        return a / b;
    }
}
```

```
public class ScientificCalculator
extends Calculator{

    @Override
    public int add(int a, int b){
        return a - b;
    }
    @Override
    public int substract(int a, int b){
        return a + b;
    }
    public int square(int a){
        return a * a;
    }
    public double divide(int a){
        return Math.sqrt(a);
    }
    // several other methods
}
```

- What's the issue with ScientificCalculator Class?
- How can we stop overriding the methods?
  - Mark the methods as **final.**
- A method marked as final **cannot be overridden by subclasses.**

Curious Cats

- When to prefer Runtime Polymorphism over Compile time Polymorphism?
  - Use Compile Time Polymorphism if,
    - Need to provide different ways to input for the same functionality
  - Use Run Time Polymorphism if,
    - Need to change few method implementations, but keep rest of core functionality same.

Summary - Rules for Method Overriding

1. **Only inherited methods** can be overridden.
2. The overriding method must have **same argument list**.
3. The overriding method must have **same return type**.
4. The overriding method **must not have more restrictive access modifier**.
    a. If the overridden method has *default* access, then the overriding one must be *default*, *protected* or *public*.
    b. If the overridden method is *protected*, then the overriding one must be *protected* or *public*.
    c. If the overridden method is *public*, then the overriding one must be only *public*.

Activity 5 (Take home - Optional)

- NumberGame - Replit (Try it out by yourself and then check the Solution given in speaker notes below)

Polymorphism Byte Overview

Messaging Application

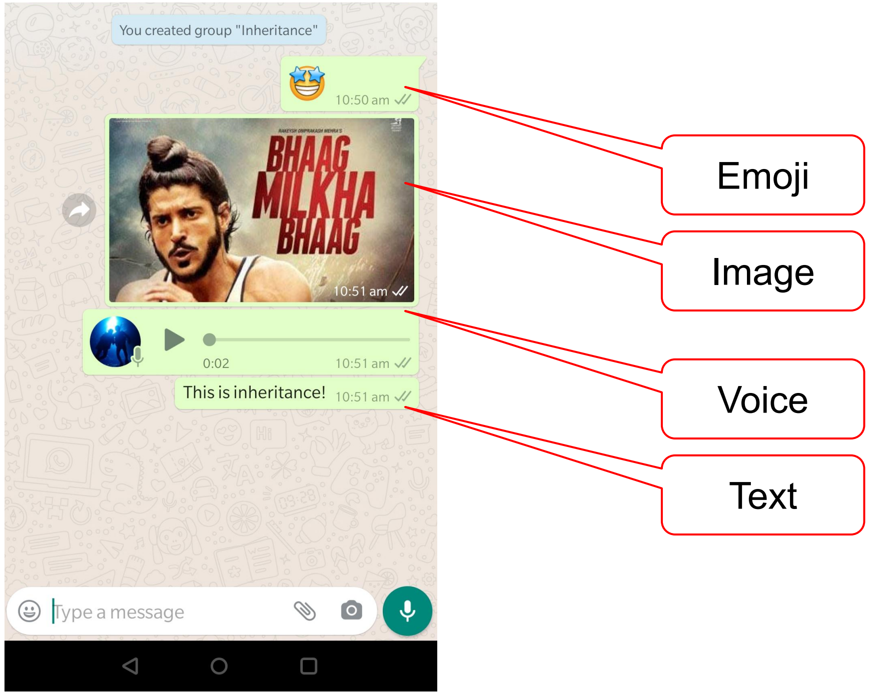
Byte breakdown (for reference)

This Bytes **builds on top of the Inheritance Byte**, please complete that first before attempting this Byte.

- **Overview** - tells you what you will be learning in this Byte and the problem statement.
- **Setup** - Pull the source code required for this Byte from the git repo into your workspace where you can execute it.
- **Milestone 1** - Understand the problem statement and new requirement (each message type is handled by a different team and they want to implement their own functionality).
- **Milestone 2** - Approach the problem in a simple way and enhance the code to handle the requirement. Learn the drawbacks of this simple approach. Think about how you can do this better.
- **Milestone 3** - Use **Method Overriding** for a better solution to one of the requirements
- **Milestone 4** - Use **Method Overloading** for a better solution to one of the requirements
- **Milestone 5** - Takeaways and summary.

Recap - YouChat - Messaging Platform

Now Support different kind of messages



You created group "Inheritance"

10:50 am

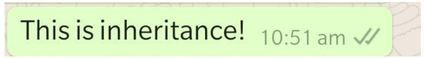
10:51 am

0:02 10:51 am

This is inheritance! 10:51 am

New Requirements

Text Message


This is inheritance! 10:51 am

Check Validity - if length of text is < 100

Image Message

Check Validity - if image is not empty

Possible Solution

- You currently have the required methods in all the different message types.
- Add validation methods in each type of message class and perform validation logic

```
public class TextMessage extends Message {
   //other methods
   public boolean isValidTextMessage(){
      if(this.getTextMessageContentSize() > 100){
         return true;
      }
      return false;
   }
}
```

```
public class ImageMessage extends Message {
   //other methods
public  boolean isValidImageMessage(){
      if(this.getImageMessageContent() != null){
         return true;
      }
      return false;
   }
}
```

- What's the issue with the above approach?
  - Clients (e.g. AndroidHandler.java) need to be aware of the method names used by each of the message types.
  - Every time a new message type is introduced or teams want to have their own implementation, the clients would need to make code changes.

Polymorphism Based Solution

```
public abstract class Message {
   // other methods and fields
   public abstract boolean isValid();
}
public class TextMessage extends Message {
   //other methods
   @override
   public boolean isValid(){
      if(this.getTextMessageContentSize() > 100){
         return true;
      }
      return false;
   }
}
public class ImageMessage extends Message {
   //other methods
  @override
  public  boolean isValid(){
      if(this.getImageMessageContent() != null){
         return true;
      }
      return false;
   }
}
```

Why is this a better solution?

- Each message type class overrides the default base class functionality.
- Clients are not impacted

Take home exercises for the session

- You will explore **Polymorphism** with this real world scenario in the following:
- For Java Learners
  - Polymorphism Byte : Java - Crio.do
- For Python Learners
  - Polymorphism Byte : Python - Crio.do

Feedback

Thank you for joining in today.

We’d love to hear your thoughts and feedback.

Further Reading

- Java - When NOT to call super() method when overriding? - Stack Overflow
- OOP: Everything you need to know about Object Oriented Programming | by Skrew Everything | From The Scratch | Medium

Thank you