Unit 4 – C# Programming

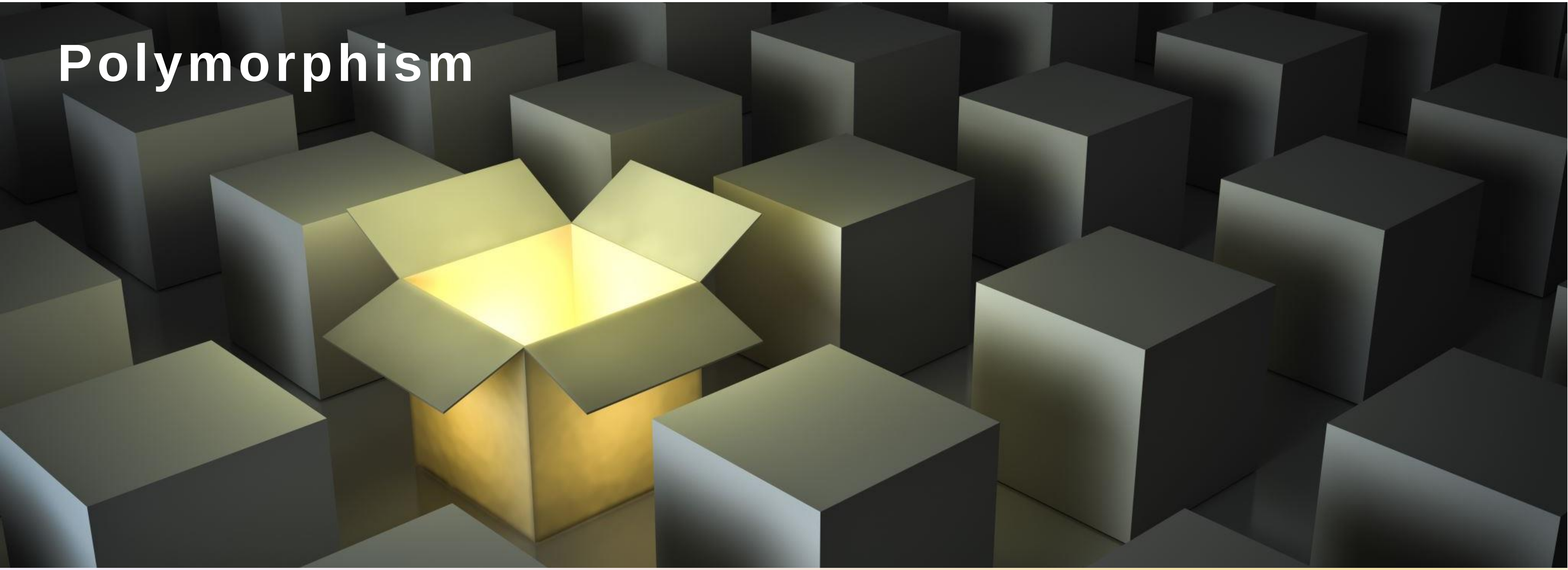
Object Oriented Programming - Polymorphism

Learning Aim: Polymorphism

By the end of the session learners will be able to:

- Be able to describe and implement Polymorphism, Overloading and Overriding.
- Be able to create and use OVERLOADED CONSTRUCTORS/METHODS.

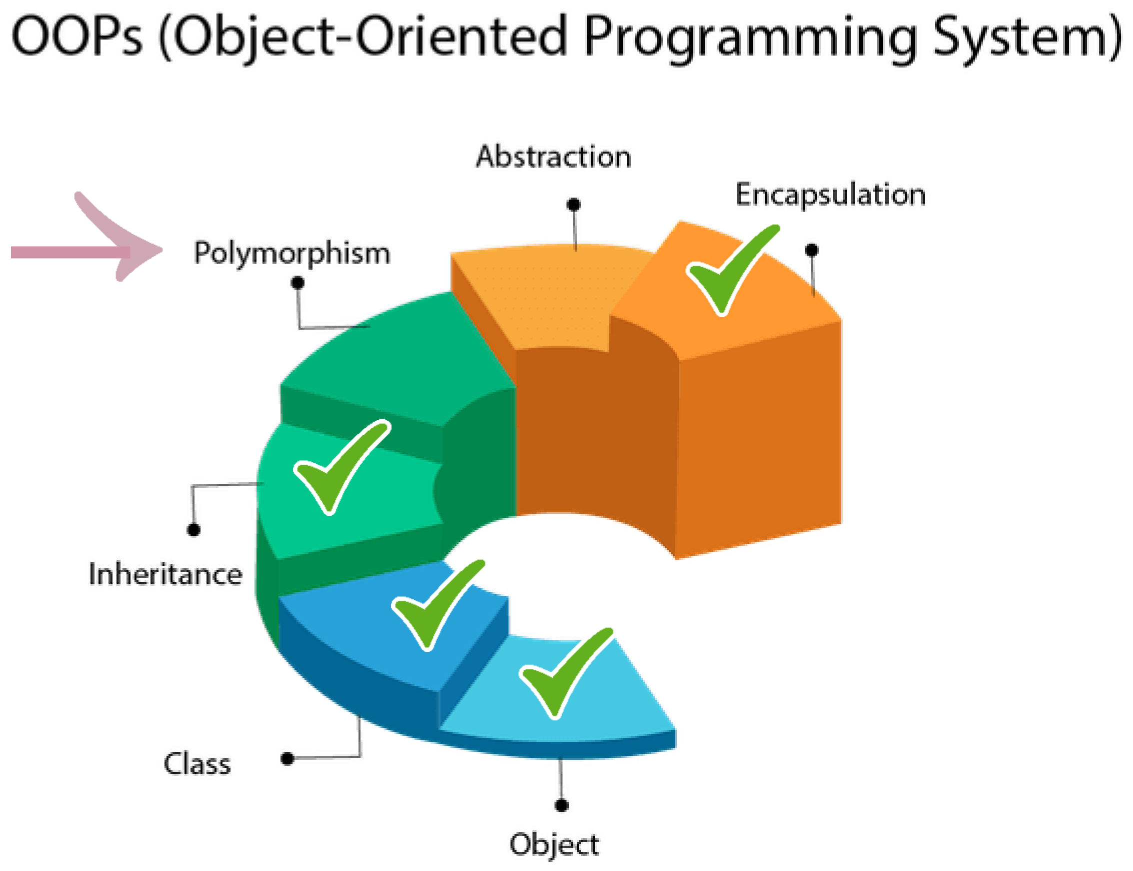
Polymorphism



Four major principles of OOP

The Objects Oriented Programming (OOP) is constructed over four major principles:

1. Encapsulation,
2. Inheritance,
3. Polymorphism,
4. Data Abstraction.



Polymorphism - Real World Examples of Polymorphism

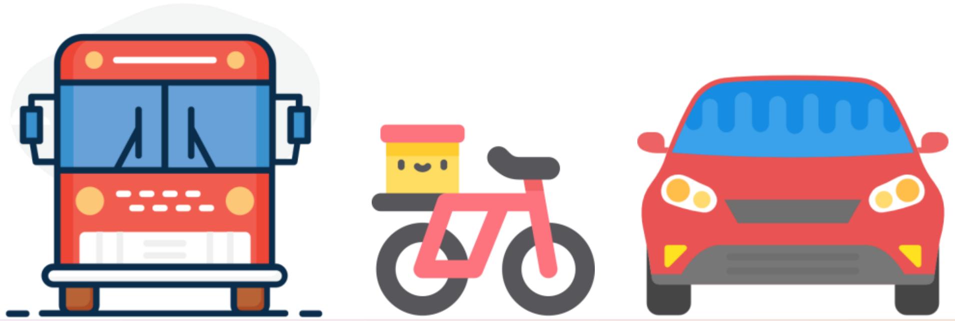
Polymorphism means one name, many forms. Polymorphism means having multiple methods, all with the same name, but different functionality.

Example 1

Vehicles

In the context of a transportation system, vehicles like cars, buses, and bicycles can all be considered as objects of a more general "Vehicle" class.

They share common characteristics like **moving**, **stopping**, and **carrying passengers**, but **each has its own specific implementation of these behaviours**.



Example 2

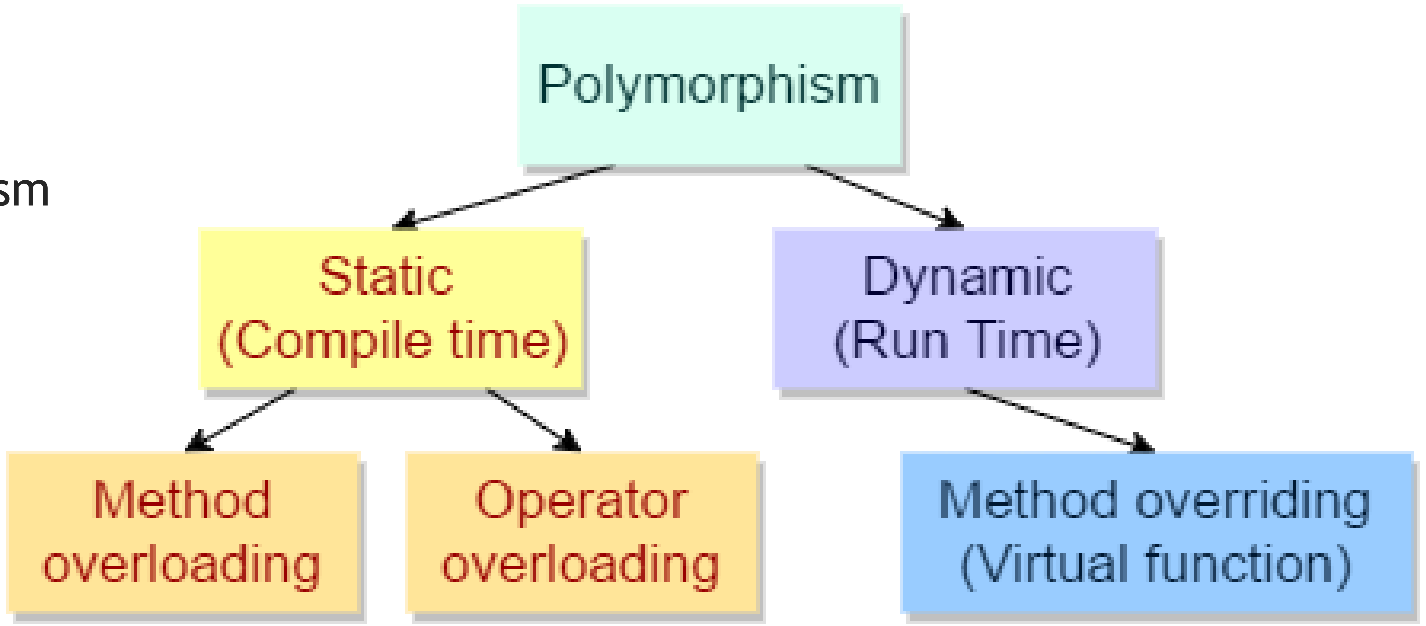
Your mobile phone, one name but many forms:

- As phone
- As camera
- As mp3 player
- As GPS

Polymorphism

There are two types of polymorphism:

- **Static** or compile time polymorphism
- **Dynamic** or runtime polymorphism

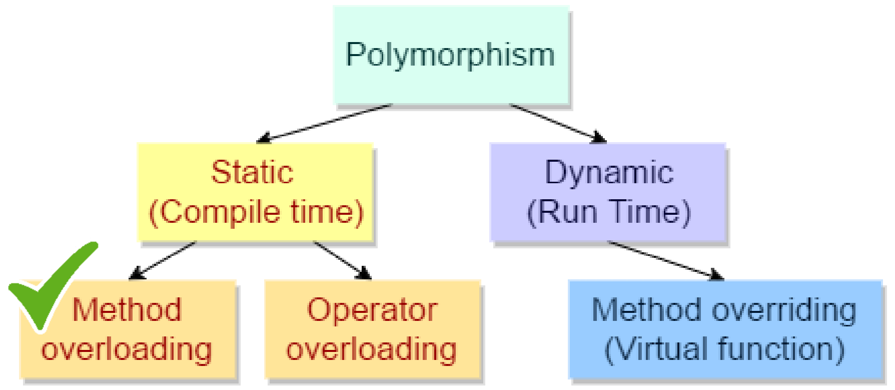


You already used **overloading** in the previous session, for the Constructors.

Like all methods, Constructors can be **Overloaded**.

This means that you can specify multiple versions of a Constructor for a class, giving the programmer the flexibility of instantiating an object using different data parameters.

Static or Compile Time Polymorphism



In static polymorphism, the decision is made at compile time.

- Which method is to be called is decided at compile-time only.
- Method overloading is an example of this.
- Compile time polymorphism is method overloading, where the compiler knows which overloaded method it is going to call.

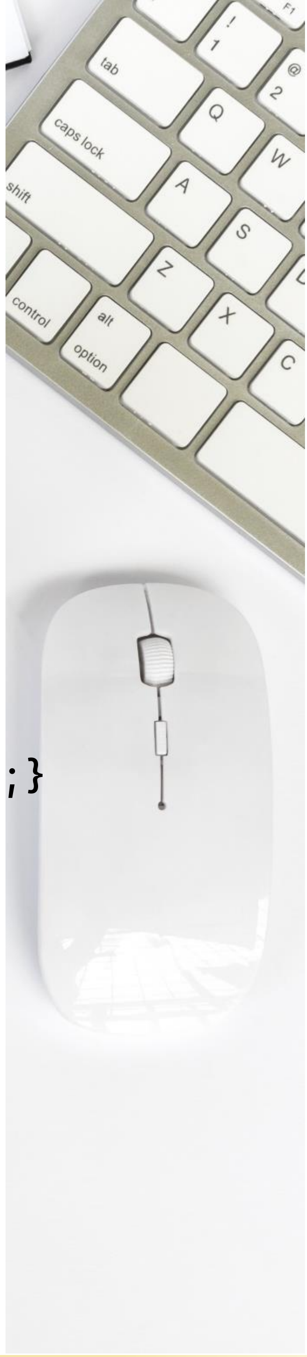
Method overloading = a class can have more than one method with the same name and different parameters.

Compiler checks the type and number of parameters passed on to the method and decides which method to call at compile time and it will give an error if there are no methods that match the method signature of the method that is called at compile time.

Overloading methods – Example

```
using System;

public class Calculator
{
    // Initial Add method for two integers
    public int Add(int num1, int num2) { return num1 + num2;}
    // Overloaded Add method for two strings
    public void Add(string a1, string a2) { Console.WriteLine("Adding Two String :" + a1 + a2);}
    // Overloaded Add method for two doubles
    public double Add(double num1, double num2) { return num1 + num2;}
    // Overloaded Add method for three integers
    public int Add(int a, int b, int c) { return a + b + c;}
    // Overloaded Add method for three doubles
    public double Add(double num1, double num2, double num3) { return num1 + num2 + num3;}
}
```



Overloading methods - Example (cont.)

```
class Program
{ static void Main() {
        Calculator calculator = new Calculator();
        int sumInt = calculator.Add(5, 10);
        double sumDouble = calculator.Add(3.5, 2.7);
        int sumIntThree = calculator.Add(5, 10, 15);
        double sumDoubleThree = calculator.Add(3.5, 2.7, 1.2);

        Console.WriteLine("Sum of two integers: " + sumInt); // Output: 15
        Console.WriteLine("Sum of two doubles: " + sumDouble); // Output: 6.2
        Console.WriteLine("Sum of three integers: " + sumIntThree); // Output: 30
        Console.WriteLine("Sum of three doubles: " + sumDoubleThree); // Output: 7.4

        Calculator obj = new Calculator();
        obj.Add("Apprentify ", "Training");
        Console.WriteLine(obj.Add(5, 10, 3));
        Console.ReadLine();
    }
}
```

Overloading constructors - Example

```
public class Student {
    public string Name { get; set; }
    public int Age { get; set; }

    // Default constructor with no parameters
    public Student()
    {
        Name = "Unknown";
        Age = 0;
    }

    // Constructor with name parameter
    public Student(string name)
    {
        Name = name;
        Age = 0; // Default age
    }

    // Constructor with name and age parameters
    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

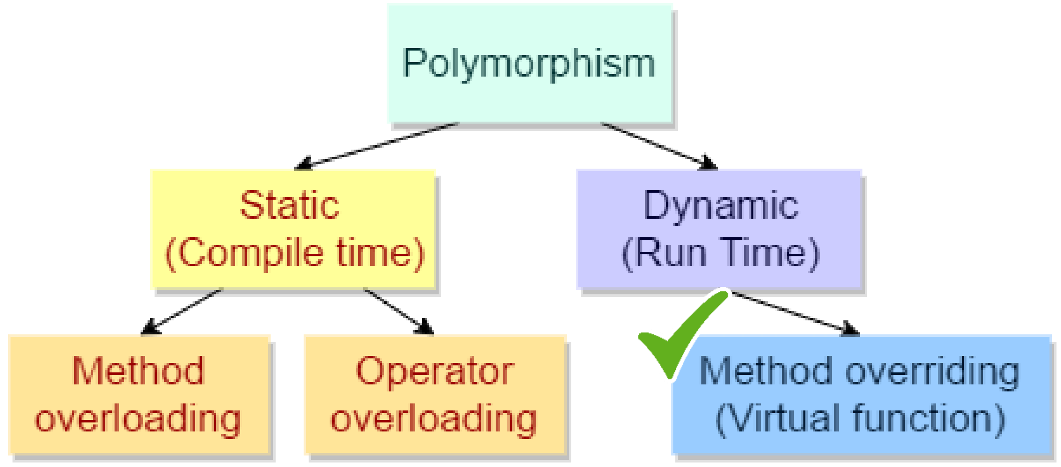
Overloading constructors – Example (cont.)

```
    ...
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
```

```
class Program
{
    static void Main()
    {
        Student student1 = new Student();
        Student student2 = new Student("Joanne");
        Student student3 = new Student("Andrew", 25);

        student1.DisplayInfo(); // Output: Name: Unknown, Age: 0
        student2.DisplayInfo(); // Output: Name: Joanne, Age: 0
        student3.DisplayInfo(); // Output: Name: Andrew, Age: 25
    }
}
```

Dynamic or runtime polymorphism



Run-time polymorphism is achieved by method overriding.

Method overriding allows us to have methods in the base and derived classes with the same name and the same parameters.

Compiler would not be aware whether the method is available for overriding the functionality or not.

So, compiler would not give any error at compile time.

At runtime, it will be decided which method to call and if there is no method at runtime, it will give an error.

Dynamic or runtime polymorphism

```
using System;

// Base class (or parent class)
public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape");
    }
}

// Derived class (or child class) that overrides the Draw method
public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}
```

Dynamic or runtime polymorphism (cont.)

```
class Program
{
    static void Main()
    {
        Shape shape = new Shape();
        Circle circle = new Circle();

        shape.Draw();  // Calls the base class method: "Drawing a shape"
        circle.Draw(); // Calls the overridden method in the derived class: "Drawing a circle"

        // Polymorphism in action
        Shape polymorphicShape = new Circle();
        polymorphicShape.Draw(); // Calls the overridden method in the derived class: "Drawing a circle"
    }
}
```

Dynamic Polymorphism - Summary

1. It is not compulsory to mark the derived/child class function with **override** keyword while base/parent class contains a **virtual** method.
2. Virtual methods allow the derived classes to provide their own implementation of that method using the override keyword, in other words, a virtual methods or property can be overridden in the derived classes.
3. Virtual methods can't be declared as private.
4. You are not required to declare a method as virtual. But, if you don't, and you derive from the class, and your derived class has a method by the same name and signature, you'll get a warning that you are hiding a parent's method.
5. We will get a warning if we won't use **Virtual/New** keyword.
6. Instead of Virtual, we can use **New** keyword.

Learning Aim: Polymorphism

You should now be able to:

- Be able to describe and implement Polymorphism, Overloading and Overriding.
- Be able to create and use OVERLOADED CONSTRUCTORS/METHODS.

Questions?