Object-Oriented Programming

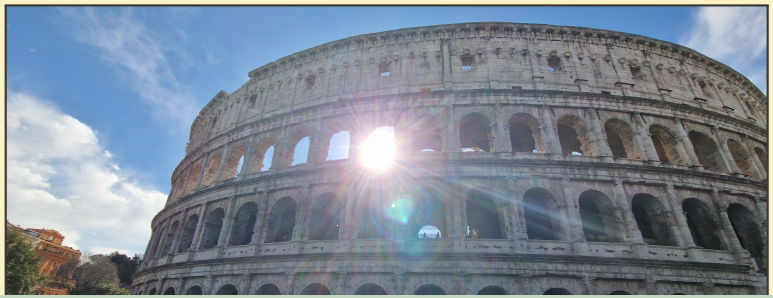
Flocea Dominic

Technical University of Moldova

October 16, 2025

Lecture 2: Classes, Objects, Constructors


POST PAVLVS
SIVS ROMANVS ONT MAX AN MD



- Top Left: *Dolmabahçe Palace*
- Middle Left: *Hagia Sophia*
- Top Right: *Saint Peter's Square*
- Middle Right: *Rome's pantheon*
- Bottom: *The Colosseum*

Lecture Quote

We who cut mere stones must always be envisioning cathedrals.

— Creed of Quarry workers, *from the book "The Pragmatic Programmer"*

Course disclaimer

The lecture slides and materials, including the content and theme were created by the author. Please do not distribute and use without prior permission.

Previously on OOP:

- Uncovered Objects-Oriented programming adoption and history;
- Listed and defined common programming paradigms and software quality factors;
- Explored general concepts and features of Object-Oriented programming;

Previously on OOP...

- Object-Oriented programming is a commonly used paradigm, especially to tackle complex applications;
- It does so by modeling the domain as a collection of modular, reusable interacting objects;
- Software quality is important and internal quality factors predict external quality factors;
- The four pillars of Object-Oriented Programming are:
  - → Encapsulation
  - → Abstraction
  - → Inheritance
  - → Polymorphism

Lecture Goals

By the end of this lecture, you should be able to:

- Understand the relationship between **objects** and **classes**;
- Explain the process of **instantiation** and the role of **constructors**;
- Differentiate between **concrete** and **abstract** classes;
- Describe and apply the concept of **static** attributes and methods;
- Explore how programming languages approch the same concepts;
- Understand the lifetime of an object;

Paradigm and Object Basics

We left on the case for Object-Oriented programming being a programming paradigm centered around objects.

- Yes, and *objects* are the primary building blocks of every Object-Oriented program;
- As discussed before all objects have a *state* (data, or attributes) and defined behavior (methods);
- In OOP, *state*, and *methods* cannot exist outside or separate of objects or classes*;
- An OOP practitioner defines the attributes and the methods of an object with *Classes*;

Classes and Methods vs Function

So a class is the constructs that we use to define objects. Is it that simple? And the set of variables and functions are bundled into objects?

- Yes, it is that simple. Classes are blueprints for objects.
- But in OOP, using the term function could be misleading, as functions do not belong to objects.
- To call a function, we do not need to *instantiate* an object...
- But to call a *method*, we always need an in memory object.

Instantiation and Concrete Classes

Instantiation - I heard about it, when we call the *new* keyword in Java/C#/C++ we *instantiate* the object. Is it that simple or is there more to it?

- Your intuition is correct. Objects are complex data structures, unlike *integers*, *floats*, or *arrays* which are primitives (generally)*.
- When we call *new* we instantiate the object in memory.
- A *concrete class* is all we need to instantiate objects.
- The limit to how many objects we can instantiate with the same class is the hardware or if a limit was specified;
- While a *concrete class* is used to instantiate the object, an object could represent multiple classes, more on this later in the course;

Concrete vs Abstract and Singleton

Woah, woah... What is a *concrete class*? Are there classes that cannot be *instatiated*? How can we restrict the number of objects? Also objects could represent multiple classes?

- Yes, there are classes that cannot be instantiated, called *abstract classes* and we will cover them later;
- A *concrete class* has all the necessary instructions in order to instantiate an object*;
- Usually, we just refer to them as classes, unless we want to specify or make a distinction.

Concrete vs Abstract and Singleton

Woah, woah... What is a *concrete class*? Are there classes that cannot be *instatiated*? How can we restrict the number of objects? Also objects could represent multiple classes?

- And yes, you can restrict number of objects of a class. This can come in handy when want only one instance of an object ex. Global Settings/Configuration. This is a Creational Design Pattern called a Singleton, but design patterns are not covered in this course.
- an object could represent multiple classes leading us to the concept of Polymorphism, which we will cover later as well as abstract classes.

Instantiation and Constructors

I feel like *instantiation* is an interesting word, and maybe there is more to it than meets the eye.

- Again your intuition is correct, objects are not just allocated in memory, they are complex.
- When an object is instantiated, two general things are happening - the memory is allocated, and it tries to load instructions for its instantiation.
- Every object has its instructions for instantiation.
- We specify these instructions with a special *method* called the constructor.
- If we do not instantiate it, (i.e. calling te *new* keyword), objects point to a null pointer*;

Abstraction

With class attributes, and methods and the constructor, we have everything to construct objects

- Right, and remember, all that can be named can be an object.
- All you need is come up with a good abstraction - which is not easy.
- A good abstraction includes only the valuable information, and all that is unnecessary is ignored.

Classes as Objects

All that can be named can be an object? Does that include classes?

- Yes, classes are objects as well;
- This is true for most OOP languages, including Java, C#, Python, Ruby;
- In C++ classes are not objects, but they have a similar construct called *type¡nfo*;
- It is an object which keeps information characterizing its instances - and the only one that can create these instances;

Summary

- **Object** — an entity that combines *state* (data) and *behavior* (methods).
- **Class** — a blueprint or template for creating objects.
- **Instantiation** — the process of creating an object from a class, usually involving memory allocation and initialization through a constructor.
- **Method** — a function defined within a class that operates on the object's data.

Types of Classes

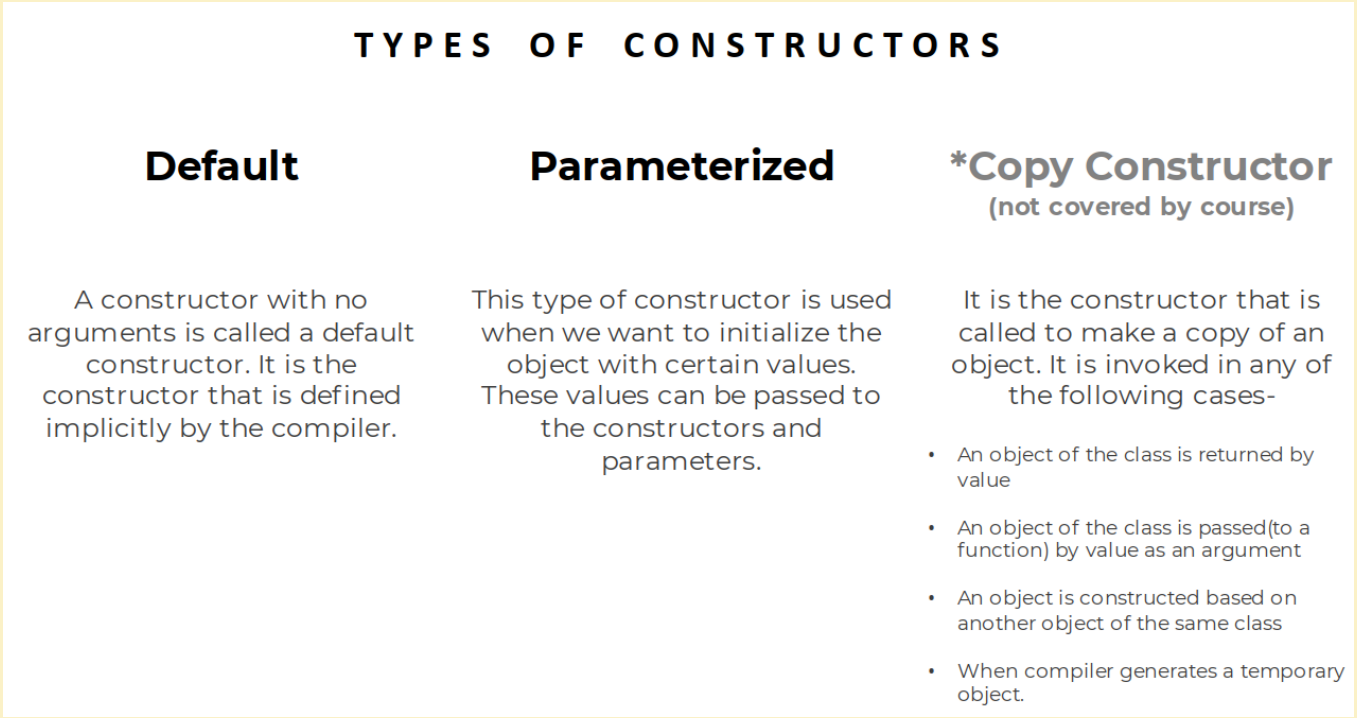
- **Concrete Class** — can be instantiated; provides full implementation of its methods.
- **Abstract Class** — cannot be instantiated;

Break or Quiz...

Constructors

- Every *concrete class* has a default constructor if one is not specified;
- Generally the constructor method has the same name as the Class itself (except for example: python);
- If a constructor has been specified - the default one generally is no longer available, it has to explicitly declared.

Types of Constructors



Code

Enough talk - let's code

Java Constructor (1) - Greeter class

```java
class Greeter {
  // STATE
  String thing;
  int times;

  // CONSTRUCTOR - same name as class
  // no return type
  // called when object is instantiated
  // defines needed attributes to create the object
  Greeter(String thing, int times) {
    this.times = times;
    this.thing = thing;
  }

  // BEHAVIOR
  String createGreeting() {
    String result = "";
    for (int i = 0; i < times; i++)
      result += "hello ";
    return result + thing;
  }
}
```

Java Constructor (2) - Greeter usage

```java
public class Main {
  public static void main(String[] args) {
    // declaration
    Greeter greeterFaf;
    // instantiation
    greeterFaf = new Greeter("FAF", 5);
    // declaration + instantiation
    Greeter greeterReader = new Greeter("reader!", 2);
    // calling object method, which returns an object of
        type String
    var fafGreeting = greeterFaf.createGreeting();

    // output -> hello hello hello hello hello FAF
    System.out.println(fafGreeting);
    // output -> hello hello reader!
    System.out.println(greeterReader.createGreeting());
  }
}
```

Python Constructor

```
1
2  class Greeter:
3      def __init__(self, thing, times): # constructor
4          self.thing = thing
5          self.times = times
6
7      def __str__(self): # Special method, tells print how
           to interpret object
8          return "hello " + self.thing * self.times
9
10     def greet(self, postfix):
11         result = ""
12         for _ in range(self.times):
13             result = result + "hello "
14         return result + " " + self.thing + postfix
15
16 # run and print those, understand the code
17 greet = Greeter("Worlda!", 3)
18 greeting = greet.greet("nice")
19 Greeter.greet(self=greet, postfix="postfix")
```

C++ Constructor

```
1
2 class Greeter {
3 private: // optional declaration, private is the default
4   std::string thing;
5   int times;
6
7 public:
8   Greeter() : thing("World"), times(1) {} // shorthand
        type constructor
9
10  std::string getThing() { return thing; }
11  void setThing(std::string newThing) { thing = newThing
        ; }
12  % code will go here
13 }
```

Kotlin Constructor

```kotlin

// constructor example in Kotlin - concise yet verbose
// what we see in the parenthesis is the primary constructor
class Greeter(thing: String = "", times: Int = 0) {
  // your code goes here
}

fun main() {
  // providing default parameters, allows us to construct the object in 4 ways
  val greeter1 = Greeter("World", 3) // primary constructor
  val greeter2 = Greeter(times = 3) // named arguments and default thing parameter
  val greeter3 = Greeter() // default arguments
  val greeter4 = Greeter(thing = "World") // order of named arguments !matters
}
```

Static Methods and Attributes vs Instance Methods and Attributes

We learned that attributes and methods belong to objects, and if we want to call a method or access an attribute we always need to instantiate the object first.

- Well, not exactly. Some attributes and methods may belong to the class itself, not the object;
- These are created using the *static* keyword in Java, C#, and C++;
- In Python, we use the *@staticmethod* decorator;
- In Kotlin, we use the *companion object* construct;
- Static members are created once per class, not per object — they exist even before any object is instantiated.

Static Methods and Attributes vs Instance Methods and Attributes

We learned that attributes and methods belong to objects, and if we want to call a method or access an attribute we always need to instantiate the object first.

- That means we can access static attributes and methods without instantiating the object;
- They are accessed through the class itself;
- Instance objects of the class *can* access static attributes and methods of the class;
- Even if declared in the same class - static methods cannot access non-static *members*;

Static Methods and Attributes vs Instance Methods and Attributes

We learned that attributes and methods belong to objects, and if we want to call a method or access an attribute we always need to instantiate the object first.

- Static members are useful for constants, counters, utility functions, and factories;
- A common pitfall from people experienced in functional/procedural programming is to overuse it;
- They quicky find out that some state variables might be inaccesible from static methods that they use as functions, ending up with a chain of static methods -> similar to procedural/functional programming;
- Heavy static use breaks encapsulation and object-oriented design principles;

Code

Enough talk - let's code

Java Static Constant, Counter

```
1   class StaticClass {
2     private int memberCounter; // cannot be accessed in
          static methods
3     private static int staticCounter = 0; // constant -
          can be accessed in static methods
4
5     public static void increment() {
6       staticCounter++; // allowed
7       memberCounter++; // not allowed: will not compile
8     }
9
10    public void increment() {
11      staticCounter++; // allowed
12      memberCounter++; // allowed
13    }
14    // getters
15  }
```

Java Singleton using Static Method

```
1   class Singleton {
2     private static Singleton instance = null;
3     private int counter = 0;
4
5     private Singleton() {} // private constructor
6
7     public static Singleton getInstance() {
8         if (instance == null) {
9             instance = new Singleton();
10        }
11        return instance;
12    }
13    public void increment() {
14        counter++;
15    }
16    // getter
17  }
```

Kotlin Singleton

```
1   object Singleton {
2     var counter = 0
3     fun increment() {
4         counter++
5     }
6     // getter
7   }
```

Java Static factory method

```
1   class Person {
2     private String name;
3     private int age;
4     private Person(String name, int age) { // private
          constructor
5         this.name = name;
6         this.age = age;
7     }
8     public static Person createAdult(String name) { //
          static factory method
9         return new Person(name, 18); // default adult
              age
10    }
11    public static Person createChild(String name) { //
          static factory method
12        return new Person(name, 0); // default child age
13    }
14  }
```

Object lifetime

Alright, but I think we are missing something... you said that memory is being allocated everytime we instantiate an object. We have created all these objects, but when do we deallocate the memory, in other words, when do we free this memory? Does it stay there?

- It depends on the language, but some things are consistent, one of them is the lifetime of an object:
  - → **Object declaration** - no memory is allocated, the object points to a null pointer*;
  - → **Object instantiation** - memory is allocated, and the constructor is called;
  - → **Object usage** - methods are called, attributes are accessed, problems are being solved, system is working;
  - → **Object deallocation (deletion)** - object goes out of scope and memory is freed;

Object lifetime

Okay, so how can I free the memory when the objects have outlived their purpose?

- In Java, Kotlin, Python and C# you dont;
- A built-in system of automatic cleaning does it for you - **the garbage collector**;
- The garbage collector (GC) frees the memory when objects are no longer referenced;
- In C++ you do;
- You have to explicitly free the memory using the *delete* keyword;

Object end of lifetime

So the GC automatically frees the memory once the objects are no longer referenced? Why cant C++ create its own GC?

- Well, maybe I wasnt specific enough about the GC;
- GC may delay collection; no longer referenced doesn’t mean immediately freed;
- GC *eventually* frees the memory of objects that are no longer referenced a process that could slow down the system;
- C++ primary focus is on performance and fine control over application resources;

Garbage Collector

Kōtotsu monster Reference from popular shonen - Bleach.

Object end of lifetime: Destructors

So C++ does not have a garbage collector, and I have to deallocate the memory everytime manually

- Remember, C++ allows you to create objects on the stack.
- C++ has a special method called the *destructor*, unlike the constructor, it ”deinstantiates” the object;
- The destructor is called automatically when the object goes out of scope;
- The destructor has the same name as the class, but is prefixed with a tilde ( );

Code

Enough talk - let's code

C++ Destructor (delete & out of scope) (1)

```
1 class Greeter {  // same as previous Greeter class
2   // ...
3   public:
4   // ...
5   // DESTRUCTOR - same name as class, prefixed with ~ (
        tilda)
6   ~Greeter() { std::cout << "bye-bye" << std::endl; }
7 };
```

C++ Destructor (delete & out of scope) (2)

```cpp
int main() {
  // out-of-scope auto destructor
  {
    auto greetOnStack = Greeter(); // stack object, auto-call destructor
    auto greetOnHeap = std::make_unique<Greeter>(); // smart pointer, auto-call destructor
  } // both objects go out of scope here, destructors called automatically

  // manual delete destructor call
  auto greetOnHeapNew = new Greeter(); // remember to delete if not using smart pointer
  delete greetOnHeapNew;
}
```

C++ Automatic destructor calls

So we are calling delete, the same way we are calling free in C... I did this in C enough, I dont want to do this again, C++, no thanks... Give me the GC instead!

- Well, C++ has a different approach to resource management, and...;
- And you dont have to call *delete* everytime you deallocate memory;
- This concept called *RAII* - Resource Acquisition Is Initialization does it for you;

C++ RAII

Sounds complicated, give me GC instead!

- No, it is not complicated, it is actually simple and elegant;
- Just create objects on the stack, and when they go out of scope, the destructor is called automatically;
- For the heap use and smart pointers like `std::unique`$_p$*tror*

Code

Enough talk - let's code

C++ Smart Pointers and RAII

```cpp
class Greeter {  // same as previous Greeter class
  // ...
  public:
  // ...
  // DESTRUCTOR - same name as class, prefixed with ~ (tilda)
  ~Greeter() { std::cout << "bye-bye" << std::endl; }
};

int main() {
  {
    auto greetOnHeap = std::make_unique<Greeter>(); // smart pointer, preffered to new keyword
  } // object goes out of scope here, destructor called automatically
  return 0;
}
```

RAII vs Garbage Collection

- Garbage Collection (Java, Python, etc.) automatic delayed cleanup.
- RAII (C++) automatic cleanup, resources released exactly when objects go out of scope.
- RAII (C++) no background GC process, and no performance overhead, also everything happens predictably, you dont have to guess.

Key Takeaways

- OOP models software as interacting objects built from reusable class blueprints.
- Constructors define how objects are created; destructors define how they are destroyed.
- Static members belong to the class, not instances.
- Memory management strategies are different for language to language, GC vs developer-managed.
- RAII and smart pointers make C++ both safe and performant, removing most need for manual deletion and GC.

Object-Oriented Programming

Or click me! (Not a scam)