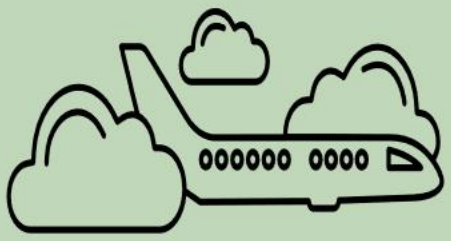
CowBoy Story

eazy bytes



Java supports Object-oriented programming (OOP)

eazy bytes

Java is an object-oriented programming (OOP) language. Object-oriented programming is a programming paradigm that is based on the concept of objects which means, all the logic is going to be imagined and built with the help of Objects.

Look around you, you will see number of objects such as,

Trees

Humans

Vehicles

Animals

Books

Each Object that you see will have properties and behaviours. For examples, a human may have properties like name, gender, age, height, weight and behaviours like walk, talk, run, study, work, sleep etc. Can these properties will be same for all humans? Offcourse not.

But all humans have common definition. To define or represent a Human, in OOP, we are going to create classes. Classes are the basic units of programming in the object-oriented paradigm.

Each object that you see belongs to a class. For example, I am an object of Human class with the name as “Madan” along with other properties that are unique to me. From a single definition, we can create any number of Objects with their own properties and behaviours.

Java supports Object-oriented programming (OOP)

eazy bytes

In a scenario where you are going to work on a bank project or application, you may write Java logic that revolve around classes and objects like Customer, Account, Loan, Card, Transactions etc.

Where as the developer who works on a health care project or application, may write Java logic that revolve around classes and objects like Patient, Doctor, Treatment, Medicines, Reports etc.

Some of the key concepts of Java OOPs are:

Class	*Object*	*Interface*	*Inheritance*	*Encapsulation*	*Polymorphism*	*Abstraction*

Object-oriented languages prioritize data over imperative commands, in contrast to non-OOPs languages that emphasize providing computers with "Do this/Do that" commands. Although object-oriented programs still issue commands to the computer, they first organize the data with the help of class & objects before executing any commands. Due to this, OOPs has many advanatges like Reusability, Modularity, Easy maintenance, Encapsulation, Inheritance & Polymorphism.

So, what is a Java class ?

eazy bytes

In Java, a class is a blueprint or a template for creating objects. It defines the attributes and methods that an object will have.

A class can contain instance variables, which represent the state of an object, and methods, which represent the behavior of an object.

The instance variables are declared at the class level and are unique to each object created from the class. The methods are also declared at the class level, but they can be called by objects created from the class.

Let's try to understand Java class by taking Car Manufacturing analogy:

In this analogy, a "Class" is like a blueprint or a template for building cars. It defines the common characteristics and behaviors that all cars of a certain type should have.

```java
public class Car {
    // Variables
    String model;
    String color;
    int horsePower;

    // Methods
    public void start() {
        // Code to start the car
    }

    public void stop() {
        // Code to stop the car
    }

    public void accelerate() {
        // Code to accelerate the car
    }
}
```

> In order to create a Java class, you need to think about what are the common characterisitcs of an Car. For example, a car will have model, color, horsePower. So create **instance variables** for the same

> Once the instance variables are declared, define the **methods** which represent the behavior of an object. The methods are also declared at the class level & they can be invoked by objects created from the class

So, what is a Java class ?

eazy bytes

A class in Java may consists below components,

- Fields (also known as variables, attributes)
- Methods (also known as functions)
- Constructors
- Instance initializers
- Static initializers

Fields and methods are also known as members of the class. A class can have zero or more class members. A class (inner or nested class) inside another class is also allowed in Java.

Below is the syntax to declare a class in Java

```
[modifier] class <class-name> {
    // Body of the class
}
```

What you see inside [ ] is optional

- ✓ modifier is optional. As of now we are using public which will make a class to use with out any restrictions. More details about modifiers follows
- ✓ The keyword class is used to define a class
- ✓ The class-name is a user-defined name of the class, which should be a valid Java identifier following Pascal case style.
- ✓ Each class has a body with different components like fields, methods etc. All the conents of a class should be specified inside a pair of braces { }.

Declaring fields in a Java class

eazy bytes

In the context of a "Car" class, the properties (also known as attributes) of objects belonging to that class are represented by fields. The "Car" class must include declarations of fields like 'model' within its body. The standard syntax for declaring a field within a class is as follows:

```
[modifier] class <class-name> {
    // Field declaration
    [modifier] <data-type> <field-name> [= <initial-value>];
}
```

* What you see inside [ ] is optional

The data type of the field precedes its name. Optionally, each field can be initialized with a specific value. If initialization is not desired, the field declaration should conclude with a semicolon after its name.

Field names should follow camelCase style

So, what is a Method ?

eazy bytes

A method within a class defines either the behavior of objects belonging to that class or the behavior of the class itself. It serves as a named block of code that can be invoked to execute its specified actions. The entity initiating the method's execution is referred to as the caller.

Optionally, a method may receive input parameter values from the caller and, in turn, return a value. A method can have zero parameters, indicating that it does not accept any input.

In the context of Java, a method is always encapsulated within the body of a class or an interface; it cannot exist independently. This encapsulation ensures that the behavior defined by the method is closely associated with the structure and purpose of the class to which it belongs.

The return type of a method signifies the data type of the value that the method will provide when called by the invoking entity. This return type may be a primitive data type, such as int or boolean, or a reference type, like Car or String. In cases where a method does not yield any value to the caller, the return type is specified as void, denoting the absence of a return value.

The name of the method must be a valid Java identifier. Similar to variable names, a Java method name should starts with a lowercase, and subsequently a word cap is used. For example, moveVehicle, applyBrake, getSpeed, and stopVehicle are valid method names. mOVevehiCLE is also a valid method name; however, it just doesn’t follow standard camelCase naming conventions.

Declaring methods in a Java class

eazy bytes

Below is the syntax to declare a method in Java,

```
[modifiers] <return-type> <method-name> (<parameters-list>)
  [throws-clause] {
    // Body of the method goes here
    [return statement;]
}
```

In a method declaration, four essential components must be included: the return type, the method name, a set of parentheses (opening and closing), and a pair of braces (opening and closing).

Methods in Java

eazy bytes

```
[modifiers] <return-type> <method-name> (<parameters-list>)
  [throws-clause] {
    // Body of the method goes here
    [return statement;]
}
```

1. **Access modifier (optional)**: It specifies the level of access to the method. Java has four access modifiers: public, private, protected, and default. If no modifier mentioned, default will be considered. For now let's use public. We can talk in detail later

2. **Return type:** It specifies the data type of the value returned by the method. If the method does not return a value, its return type should be void.

3. **Method name:** It is the name of the method, which should be a meaningful and descriptive name that indicates the purpose of the method.

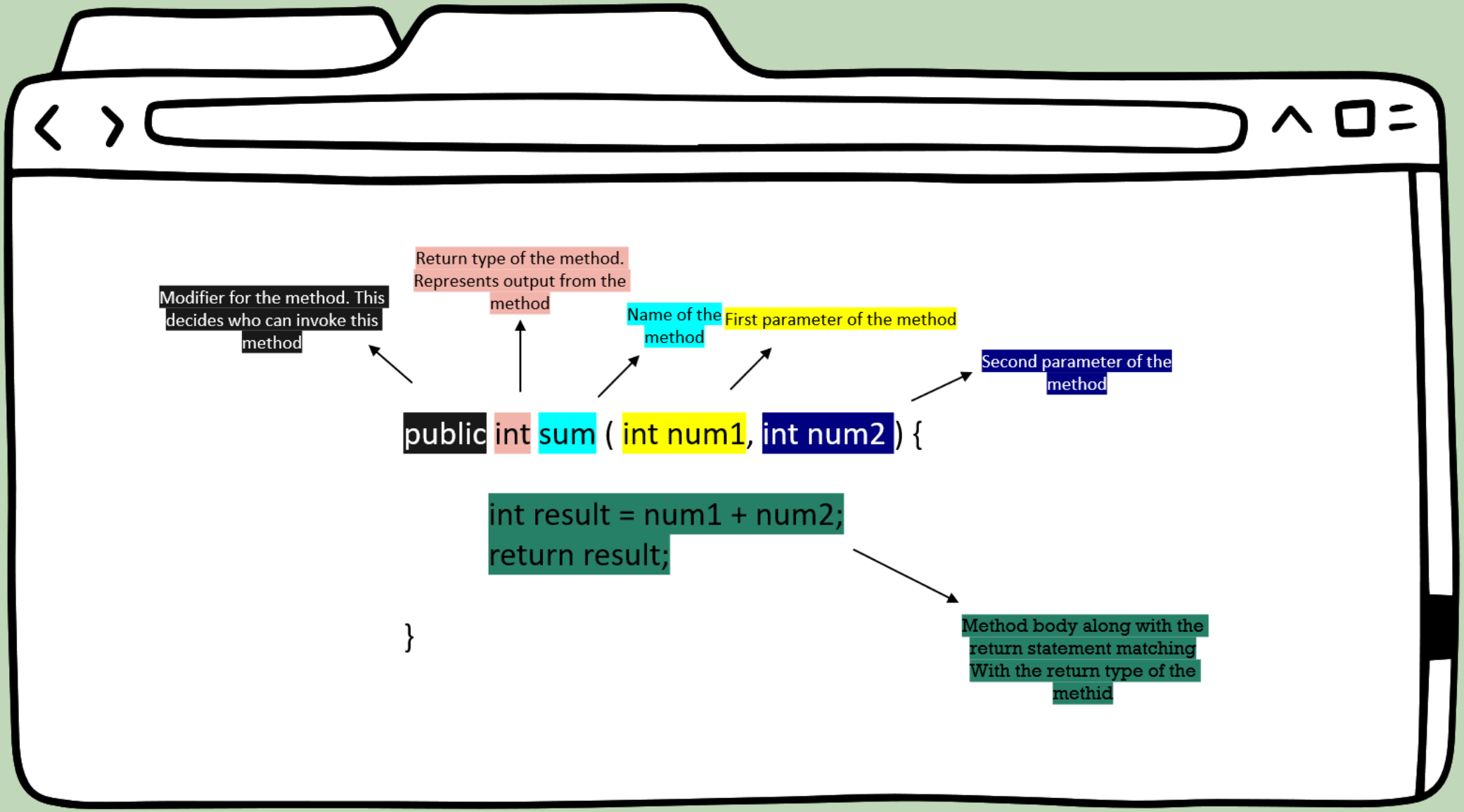
4. **Parameter list (optional):** It is a list of input parameters passed to the method. Each parameter consists of a data type and a variable name, separated by a comma. If a method takes no parameters, an empty pair of parentheses is used.

5. **throws:** The list of parameters may optionally be followed by a throws clause that declares the types of exceptions that the method may throw.

6. **Return statement (optional):** It specifies the value to be returned by the method. The return statement is optional in void methods.

Methods in Java

eazy bytes



Parameters & Arguments of Java methods

eazy bytes

Parameters & arguments are often used interchangeably by developers.

When you define a method, you can specify one or more parameters that the method accepts. A parameter is a variable that is used to store an argument that is passed to the method.

When you call a method, you provide one or more arguments that are passed to the method. An argument is a value that is passed to a method when it is called.

For example, consider the following method definition:

```
public void printMessage(String message) {
   System.out.println(message);
}
```

This method takes a single parameter, message, which is of type String. When you call this method, you provide an argument that is passed to the method. For example:

```
printMessage("Hello, world!");
```

In this case, the argument "Hello, world!" is passed to the printMessage method as the value of the message parameter.

Method signature

eazy bytes

A method has a signature that serves to uniquely identify it within a specific context. This signature is composed of four integral elements:

- ✓ Name of the method
- ✓ Types of parameters
- ✓ Number of parameters
- ✓ Order of parameters

Modifiers, return types, parameter names, and the throws clause are not part of the signature of a method.

The signature of a method uniquely identifies the method within a class. It is not allowed to have more than one method in a class with the same signature.

Method signature

eazy bytes

Method Delcaration	Method Signature
public int add (int num1, int num2)	add (int, int)
public int add (int num3, int num4)	add (int, int)
int add (int num1, int num2)	add (int, int)
public int add (int num1, int num2) throws Exception	add (int, int)
void printMessage (String msg)	printMessage (String)
public double add (int num1, double num2)	add (int, double)
public double add (double num1, int num2)	add (double, int)
public void sayHello ( )	sayHello ( )

It is crucial to recognize that both the type and arrangement of a method's parameters constitute elements of its signature. For instance, the methods double add(int num1, double num2) and double add(double num1, int num2) possess distinct signatures due to differences in the order of their parameters, even though the number and types of parameters remain the same.

return statement in a method

eazy bytes

A return statement is used to return a value from a method. It starts with the return keyword. In cases where a method yields a value, the return keyword must be succeeded by an expression that resolves to the specific value intended for return.

```
return <Java expression>; // The Java expression must evaluate to a value matching with the return type of the method
return num1 + num2 ; // Sample example where a Java expression defined
return 30 ; // Sample example where a value directly returned
```

When a method doesn't produce a value (specified by a return type of void), inclusion of a return statement is optional. If, however, a method with a void return type does incorporate a return statement, the return keyword is directly succeeded by a semicolon, indicating the conclusion of the statement and affirming the absence of an expression.

```
return ; // In case of void return type of method, we can end a method with a simple return or ignore it completely
```

The purpose of a return statement is to give control back to the method's caller, as implied by its name. When accompanied by an expression, the return statement evaluates that expression and transmits the resulting value to the caller. In cases where no expression is present, the return statement simply transfers control back to the caller. Importantly, a return statement consistently marks the conclusion of a method's execution, serving as the final statement within the method's body.

Method invocation

eazy bytes

The process of running the code associated with a method is referred to as "calling a method" or "invoking a method." This entails using the method's name, followed by the values for its parameters (if any), enclosed within parentheses. To invoke the `sum` method, we can use the following statement:

```
sum(6, 24);  // If the method is in the same class, we can simply invoke using the method name
objectName.sum(6, 24); // If the method is in another class, we can invoke the method using the object of the class and dot(.) operator
ClassName.sum(6, 24); // If the method is in another class and it is a static method, we can invoke the method using the class
                      name and dot(.) operator
```

When we call a method by passing the values 6 and 24, Java will bind these values to the method parameters by the time it invokes the method for execution.

Incase if you want to catch the output from the method, we can change the above method invocations to like shown below,

```
int result = sum(6, 24);
int result = objectName.sum(6, 24);
int result = ClassName.sum(6, 24);
```

Let’s say hi to main method again

eazy bytes

Here is the declaration of main method that will be used by JVM to start the Java program,

```
public static void main(String[] args) {
    // Method body goes here
}
```

In the declaration of the main() method, two modifiers, namely public and static, are used. The public modifier facilitates access to the method from any part of the application, provided the declaring class is accessible.

Declaring the main() method as static is essential because the main() method acts as the entry point for a Java application, initiated by the JVM when a class is executed. Since the JVM doesn't have the capability to create an instance/object of a class, a standardized approach is necessary to commence Java applications. By specifying the main() method details and designating it as static, a consistent and standard mechanism is established for the JVM to initiate Java applications. The static nature of the main() method allows the JVM to invoke it using the class name.

The return type of the main() method is void, indicating that it doesn't yield a value to its caller.

The method is named "main" and takes a parameter of type String array (String[]), traditionally referred to as "args." It's noteworthy that while "args" is commonly used, you have the flexibility to choose any parameter name, such as ”myArgs”. However, the selected name must be consistently used within the method's body if reference to the passed parameter is necessary.

Tricky question about main method

eazy bytes

What happens if we don't mention static inside the main method signature ?

The method and program will compile with out any issues. But if we try to execute the program, we will receive an error or exception.

Can we create multiple main methods inside a class ?

Yes we can create multiple main methods inside a class, as long as the main method signatures are different. Here is the example of the same.

```
public class HelloWorld {

    public static void main(String[] myArgs) {
        /* used as the program entry point */
    }
    public static void main(String[] myArgs, float num) {
        /* Another main() method */
    }

    public double main() {
        /* Another main() method */
    }
}
```

Creating and Initializing Objects

eazy bytes

Once you have defined a class, you can create an object of that class using the new keyword. The new keyword allocates memory for the object and returns a reference to it.

```
Employee myObj = new Employee();
```

This creates a new object of the Employee class and assigns it to the myObj variable. The new keyword calls the constructor of the class to initialize the object.

Employee Java class

```
Employee employee1 = new Employee();
employee1.firstName= "John";
employee1.lastName= "Doe";
employee1.age=30;
employee1.gender='M';
```

```
Employee employee2 = new Employee();
employee2.firstName= ”Maria";
employee2.lastName= ”Bess";
employee2.age=25;
employee2.gender=‘F';
```

```
Employee employee3 = new Employee();
employee3.firstName= ”Kiran";
employee3.lastName= ”Kumar";
employee3.age=28;
employee3.gender='M';
```

The Dot Operator (.)

The dot operator (.) gives you access to an object’s state and behavior (instance variables and methods).

Like you can see above, three Employee objects are created with new operator. Post that we set different values related to them. This way from single blue print class, we can create any number of objects with their data

eazy bytes

Class, Objects, methods in action


```java
public class Employee {

    String firstName;
    String lastName;
    byte age;
    char gender;

    public byte getAge() {
        return age;
    }

    public char getGender() {
        return gender;
    }
}
```

```java
public class EmployeeMain {

    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.firstName = "John";
        employee.lastName = "Deo";
        employee.age = 30;
        employee.gender = 'M';

        System.out.println(employee.getAge());
        System.out.println(employee.getGender());
    }

}
```

As demonstrated above, using a EmployeeMain class with a main method, we instantiated an employee object, populated it with data, and then called its methods to display the information contained within the object.

What is a Constructor ?

eazy bytes

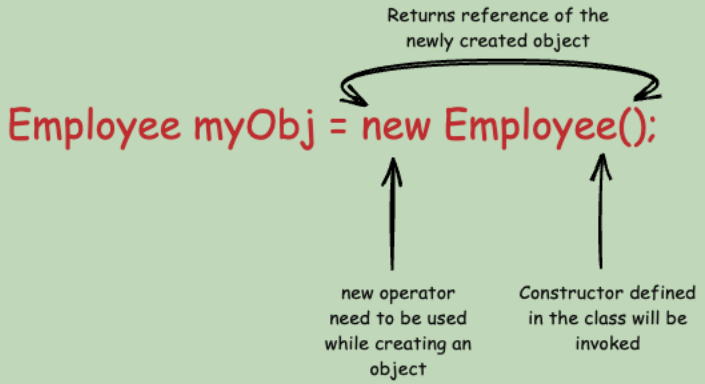
A constructor is a code block that is essential for creating an object, which is an instance of a class. Similar to a method, it will have a name and input parameters. However, it is distinct because it shares the same name as its corresponding class and does not return any value, not even void.

When an object is instantiated in Java, the JVM invokes a constructor that serves the purpose of initializing the object's state by assigning values to its declared properties & and allocating memory.

Below is a sample class & it's default constructor,

```
public class Employee { // Class declaration
    public Employee ( ) { // Constructor declaration
        // Constructor logic to be executed while object creation
    }
}
```

> If a class does not have any constructors defined explicitly, then Java automatically provides a default constructor with an empty body.
>
> Employee myObj = new Employee();
>
> With the above code, we created an object of Employee class, but the values inside this object will be of default values, since we leveraged default constructor provided by JVM



The default constructor will not have any parameters. Default constructor is also called as no-args constructor.
The access level of the default constructor is the same as the access level of the class.

What is a Constructor ?

eazy bytes

One important aspect to note is that constructors, although they look like methods, are not methods or even members of the class. A constructor doesn't have a return type and always has the same name as the class. Its only purpose is to be called when a new instance of the class is created.

What kind of code should you write in a constructor? In a constructor, the focus should be on initializing the instance variables of the newly created object. It's essential to limit the code within a constructor to tasks related to the initialization of these variables. It's crucial to recognize that when a constructor is called, the object is still in the process of creation, and assuming a fully formed object in memory can lead to unexpected outcomes.

If necessary, it is possible to declare any number of constructors explicitly, each taking a different set of parameters to set the initial state. This is known as constructor overloading.

A general structure representation of a Java constructor can be written as follows:

```
[modifiers] <constructor-name>(<parameters-list>)
                    [throws-clause] {
    // Constructor body
    // Assign values to instance variables
    // Perform other initializing tasks
}
```

Access modifier: The constructor may have an access modifier such as public, private, protected, or package-private (default). For now let's use public. We can talk in detail later

Constructor Name: The name of the constructor must match the name of the class it belongs to.

Parameter List: The constructor may take zero or more parameters, which are declared in a comma-separated list within the parentheses.

throws: Optionally, the closing parenthesis may be followed by a throws clause, which in turn is followed by a comma-separated list of exceptions.

Constructor Body: The constructor body is a block of code that is executed when the object is created using new operator. It may contain any valid Java code, including assigning values to instance variables, performing calculations, if statements, loops, and method calls etc.

Constructor Overloading

eazy bytes

Below is the code where we created a custom constructor of Employee & how to create a object of Employee using the same constructor.

```java
public class Employee {

    String firstName;
    String lastName;
    byte age;
    char gender;

    public Employee(String firstName, String lastName,
                    byte age, char  gender) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.gender = gender;
    }

    public byte getAge() {
        return age;
    }

    public char getGender() {
        return gender;
    }
}
```

```java
public class EmployeeMain {

    public static void main(String[] args) {


        Employee employee1 = new Employee("John", "Doe", (byte) 30, 'M');
        Employee employee2 = new Employee("Max", "Steve", (byte) 25, 'F');

        System.out.println(employee1.getAge());
        System.out.println(employee1.getGender());

        System.out.println(employee2.getAge());
        System.out.println(employee2.getGender());
    }
}
```

this keyword: In Java, "this" is a keyword that refers to the current object instance. It can be used to refer to the instance variables and methods of the current object.

If a custom constructor is defined in a Java class, the JVM will not generate a default constructor automatically. In such cases, if a default constructor is required, the developer must explicitly create it.

Constructor Chaining

eazy bytes

Constructor chaining is a technique used in object-oriented programming to invoke one constructor from another constructor in the same class.

For example, consider a class ”Employee" that has two constructors - one that takes no parameters and another that takes parameters:

```
public class Employee {

    String firstName;
    String lastName;
    byte age;
    char gender;

    public Employee () {

    }

    public Employee(String firstName, String lastName,
                    byte age, char gender) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.gender = gender;
    }

    public byte getAge() {
        return age;
    }

    public char getGender() {
        return gender;
    }
}
```

Let’s try to call the constructor with parameters from the constructor with no parameters. As a naive developer you may try like shown below,

Constructor Chaining

eazy bytes

In Java, there's a unique method for invoking one constructor from another: employ the keyword this as if it were the actual constructor name. The following code calls the constructor with parameters from the constructor with no parameters using the this keyword,

```java
public class Employee {


    public Employee () {
        this("John", "Deo", (byte) 30, 'M');
    }

    public Employee(String firstName, String lastName,
                    byte age, char gender) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.gender = gender;
    }

  // other code removed for brevity


}
```

When calling a constructor from another constructor in Java, three important rules must be followed,

1. The call to another constructor must be the very first thing in the constructor
2. A constructor cannot call itself
3. Same constructor can't be called more than once

The following constructor invocations are not valid,

```java
public Employee () {
    System.out.println("Compilation fails");
    this("John", "Deo", (byte) 30, 'M');
}
```

Compilation fails as the constructor invocation is not the first statement

```java
public Employee () {
  this();
  System.out.println("Compilation fails");
}
```

Compilation fails as the constructor can't invoke itself

return statement in constructor

eazy bytes

In Java, constructors cannot declare a return type, which means they can't return any value. While it's technically possible to use a return statement without an expression inside a constructor, doing so is discouraged as it's considered poor practice.

Compilation fails as the constructor can’t return any value from it’s body

```
public Employee () {
  System.out.println("Compilation fails");
  return "SomeValue";
}
```

Valid code but it is a bad practice to mention return inside the constructor body

```
public Employee () {
  System.out.println("Compilation fails");
  return;
}
```

Instance Initialization Block

eazy bytes

In Java, an Instance Initialization Block (IIB) is a block of code within a class that is executed each time an instance of the class is created. The primary purpose of the Instance Initialization Block is to initialize instance variables of an object. This block is particularly useful when you want to perform some initialization logic for each object of a class.

An Instance Initialization Block is defined within a class, but outside of any method or constructor. An instance initializer does not have a name. Its code is simply placed inside an opening brace and a closing curly braces { } .

```
public class Car {

    String model;
    String color;
    int horsePower;

    {
        this.model = "Camry";
        this.color = "Black";
        this.horsePower = 200;
    }
}
```

The code inside an Instance Initialization Block is executed whenever an object of the class is instantiated.

You can have multiple Instance Initialization Blocks within a class, and they are executed in the order in which they appear in the class definition.

An instance initializer cannot have a return statement.

```
public class Car {

    String model;
    String color;
    int horsePower;

    {
        this.model = "Camry";
        this.color = "Black";
    }

    {
        this.horsePower = 200;
    }
}
```

But why Java provides two options to initialize the objects, one using Constructor and other using Instance Initialization Block ?

Instance Initialization Block

eazy bytes

Reason 1:

It may surprise for you to hear that not all Java classes have constructors. This is especially true for anonymous classes, which, as the name suggests, lack a formal name. Since constructors need a name matching the class, anonymous classes can't have them. So, how do you initialize an object of an anonymous class? Enter the instance initializer, a handy tool not limited to anonymous classes — any class can use it to initialize its objects.

Reason 2:

It's important to note that Instance Initialization Blocks are executed every time an object is created. If you have common initialization logic across multiple constructors, using an Instance Initialization Block can help avoid code duplication.

In this example, the prinln statement defined inside instance initialization block is going to be executed when ever an object of Car is being created regardless of which constructor being used.

```java
public class Car {

    String model;
    String color;
    int horsePower;

    {
        System.out.println("Car object is being created...");
    }

   public Car () {

   }

   public Car (String model, String color, int horsePower) {
        this.model = model;
        this.color = color;
        this.horsePower = horsePower;
   }

}
```

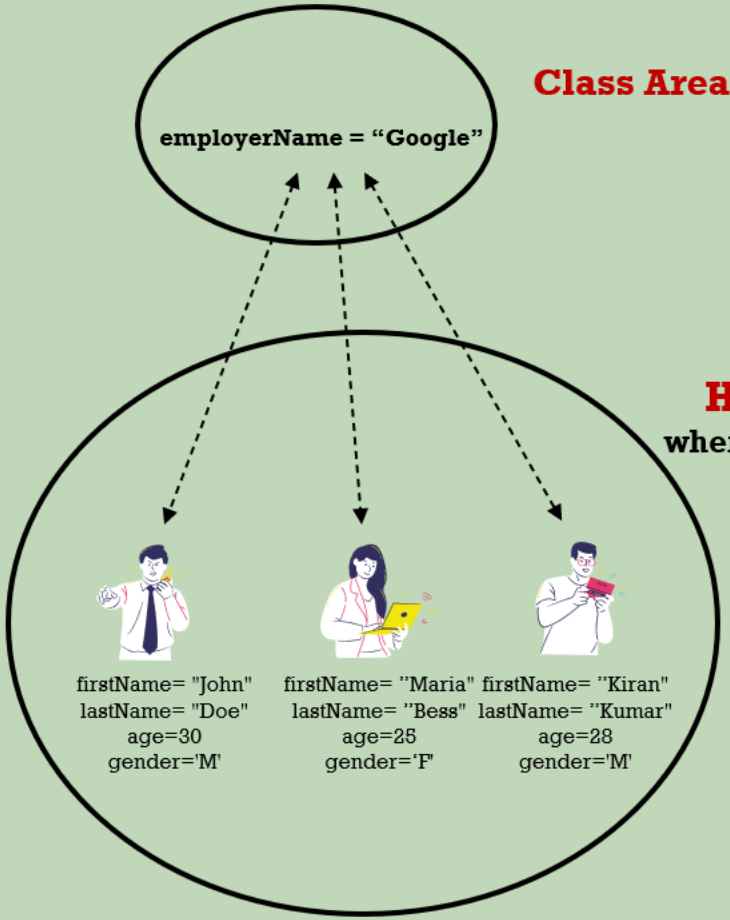
static variables/fields

eazy bytes

In Java, a static variable is a variable that belongs to a class rather than an instance/object of the class. A static variable is declared using the static keyword and is shared among all objects of the class.

static String employerName = "Google";

They are typically used to store data that is common to all instances/objects of the class, such as a constant value or a read-only value. Here in our scenario, since all the employees are going to work in a same company, it makes sense to create employerName variable as static. They can be accessed using the class name without creating an object of the class.



In Java, static variables are stored in the memory area known as the "Method Area" or "Class Area". This area is a part of the JVM's memory and is shared by all threads. When a Java program is executed, the JVM loads the class into the memory, and the static variables of the class are initialized in the Method Area. The static variables remain in the memory until the program terminates.

Since the static variables are shared by all instances of the class, they can be accessed without creating an instance of the class. The JVM ensures that only one copy of the static variable exists in the memory, and all instances of the class refer to the same copy. It is worth noting that the size of the Method Area is limited, and if a program creates too many static variables or classes, it can cause an OutOfMemoryError.

constant static variables

eazy bytes

Some times, we want a static variable to be final or constant. So that no developer can change it during the execution of the program. For example, we can have same employerName as a constant or final value. For these kind of scenarios, we can create a constant static variable. In Java, a constant static variable is a static variable that has the final keyword and is used to store a constant value that will not change during the lifetime of the program. These variables are typically declared with all capital letters and underscores to separate words (snake_case), following the convention for naming constants in Java.

```java
public class Employee {

    public static final String EMPLOYER_NAME = "Google";
    String firstName;
    String lastName;
    byte age;
    char gender;

    public Employee(String firstName, String lastName,
        byte age, char gender) {

        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.gender = gender;
    }
}
```

In this example, the EMPLOYER_NAME variable is a constant static variable that is set to the value of Google using the final keyword. Since EMPLOYER_NAME is a static variable, it can be accessed directly through the class name without needing to create an instance of the class.

static & instance variables

eazy bytes

static variable

Static fields are also known as class variables because they are associated with the class rather than with instances of the class. They are declared using the static keyword and are initialized only once, when the class is loaded into memory. All instances of the class share the same static field.

Static fields are accessed by using the

ClassName.StaticFieldName. For example, in the below example class, the static field can be accessed using Example.staticField

```
public class Example {
    public static int staticField = 0;
    public int instanceField = 0;
}
```

instance variable

Instance fields, on the other hand, are associated with each instance of a class. They are declared without the static keyword and are initialized each time a new instance of the class is created. Each instance has its own copy of the instance field.

Instance fields are accessed by using the

objectVariable.InstanceFieldName. For example, in the below example class, the static field can be accessed using the following code,

```
Example exObj = new Example();
exObj.instanceField
```

```
public class Example {
    public static int staticField = 0;
    public int instanceField = 0;
}
```

static methods in Java

eazy bytes

In Java, a static method is a method that is associated with a class, rather than with an instance of the class. This means that you can call a static method directly on the class itself, without needing to create an instance of the class first. To declare a static method in Java, you use the static keyword in the method declaration. Here is an example of a class with a static method:

```
public class ArithmeticOperations {

    public static int addition(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }

}
```

In this example, the ArithmeticOperations class has a static method named addition that takes two integer parameters and returns the sum of the two values. Here, the addition method is called directly on the ArithmeticOperations class, without needing to create an instance of the class first. The returned value, 25, is assigned to the result variable.

```
int result = ArithmeticOperations.addition(20, 5);
```

static methods in Java

eazy bytes

Static methods are commonly used for utility functions that perform a specific operation and do not require access to instance variables or methods. They can also be used for factory methods that create new instances of a class, or for methods that perform a global operation that affects all instances of the class.

Static methods cannot access instance variables or methods of a class, only other static variables or static methods or static variables & methods of other classes can be accessed. To access instance variables or methods, you must create an instance of the class and call the method or access the variable through the instance.

In Java, both static and non-static methods are stored in the same memory area as the class itself, which is called the "method area" or “class area". This area is allocated when the class is loaded by the JVM and it contains information about the class, including its fields, methods, and static variables.

The famous main () method in Java is a static method. Because since it is the first method that needs to be invoked by the JVM without needing to create an instance of the class first.

static & instance methods

eazy bytes

static method

Static methods are also known as class methods because they are associated with the class rather than with instances of the class. They are declared using the static keyword and can be called directly on the class, without the need to create an instance of the class.

They cannot access instance variables or methods directly, and can only access static variables or methods.

Static methods can be accessed by using the ClassName.StaticMethodName(). For example, in the below Example class, the static method can be accessed using Example.staticMethod()

```
public class Example {
   public static void staticMethod() {
      // Logic
   }
}
```

instance method

Instance methods, on the other hand, are associated with each instance of a class. They are declared without the static keyword and can be called only on instances of the class.

They can access both instance variables and methods, as well as static variables and methods.

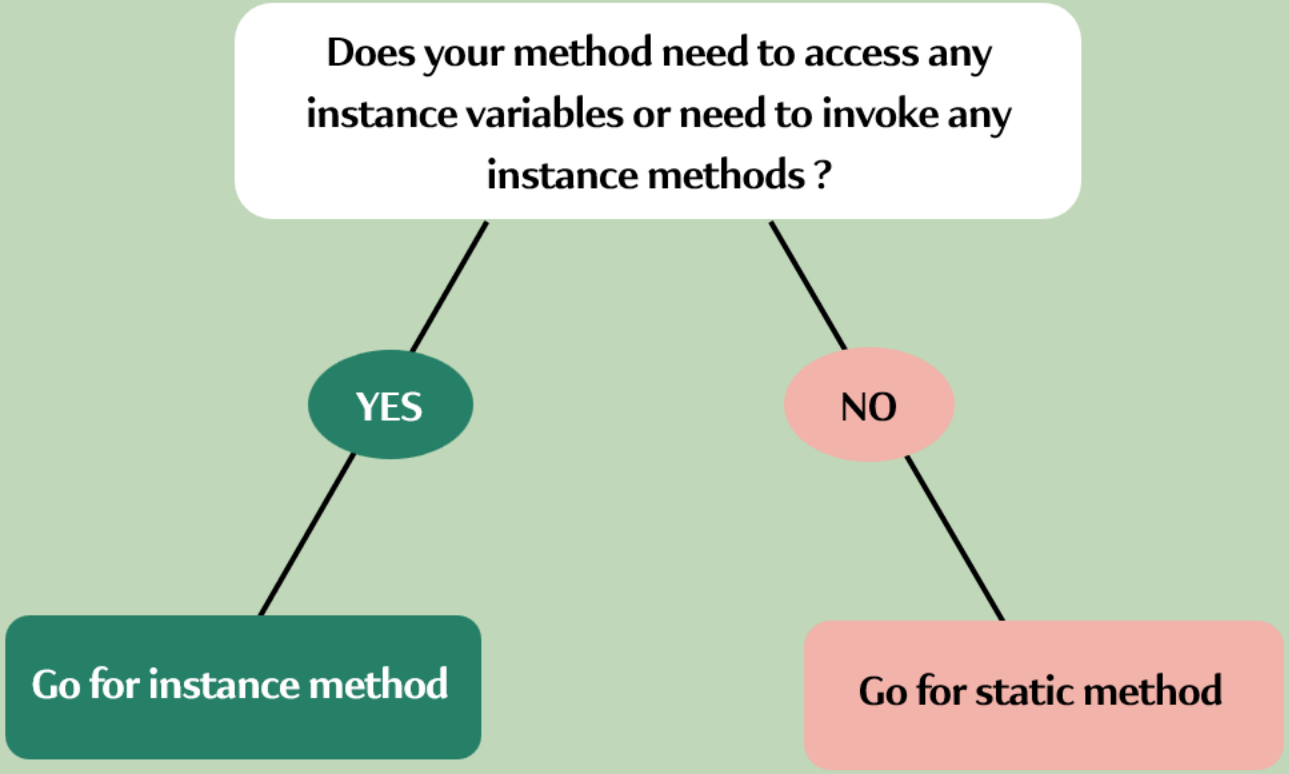
Instance methods can be accessed by using the ObjectVariable.InstanceMethodName(). For example, in the below Example class, the static field can be accessed using the following code,

```
Example exObj = new Example();
exObj.instanceMethod();
```

```
public class Example {
  public void instanceMethod() {
        // Logic
  }
}
```

should I create a static or instance method ?

eazy bytes



static Initialization Block

eazy bytes

In Java, a Static Initialization Block (SIB) is a block of code that is associated with a class rather than with instances of that class. It is executed when the class is loaded into the Java Virtual Machine (JVM) for the first time. The primary purpose of the Static Initialization Block is to initialize static variables of a class.

A Static Initialization Block is defined using the static keyword and is enclosed in curly braces { }. It is placed within the class body but outside of any method or constructor.

```
public class Car {

    static byte noOfEngines;

    static {
        noOfEngines = 1;
    }
}
```

The code inside a Static Initialization Block is executed when the class is loaded into memory by the JVM. It is executed only once, no matter how many objects/instances of the class are created.

A static initializer cannot have a return statement.

Static Initialization Blocks are useful when you need to perform one-time initialization for static variables or when the initialization logic is more complex and cannot be accommodated within a single line.

You can have multiple static Initialization Blocks within a class, and they are executed in the order in which they appear in the class definition.

The java command loads the definition of a given class before it tries to execute its main() method. When the definition of the class is loaded into memory, at that time the class is initialized, and its static initializer will be executed. This is the reason that you see the code from the static initializer executed before you see the code from the main() method executes.

eazy bytes

static initializer block

Static blocks starts with a keyword static followed by { }

Static blocks executes during class loading

It can only refer static variables

It can not use this keyword as there are no object instances as part of static block

It executes only once during the life time of the program when the class loads into the memory

instance initializer block

Instance initializer blocks starts with { }

Instance initializer executes during object instantiation

It can refer both static & non-static variables

It can use this keyword

It runs everytime there is a call to the constructor or when we try to create a object using new operator

Where does **Java store** classes, objects, variable, methods etc.

eazy bytes

In Java, memory management is handled by the JVM (Java Virtual Machine), which is responsible for allocating and deallocating memory for Java programs. Java programs are executed in a runtime environment that includes several different types of memory:

Heap memory: This is the area of memory where objects are allocated. Each object is allocated on the heap, and its memory is reclaimed by the JVM's **garbage collector** when it is no longer being used.

Stack memory: This is the area of memory where local variables and method calls are stored. When a method is called, a new stack frame is created on the stack, which contains the method's local variables and parameters. When the method returns, the stack frame is removed from the stack.

Method Area: This is the area of memory where class definitions, method definitions, and other runtime constants are stored.

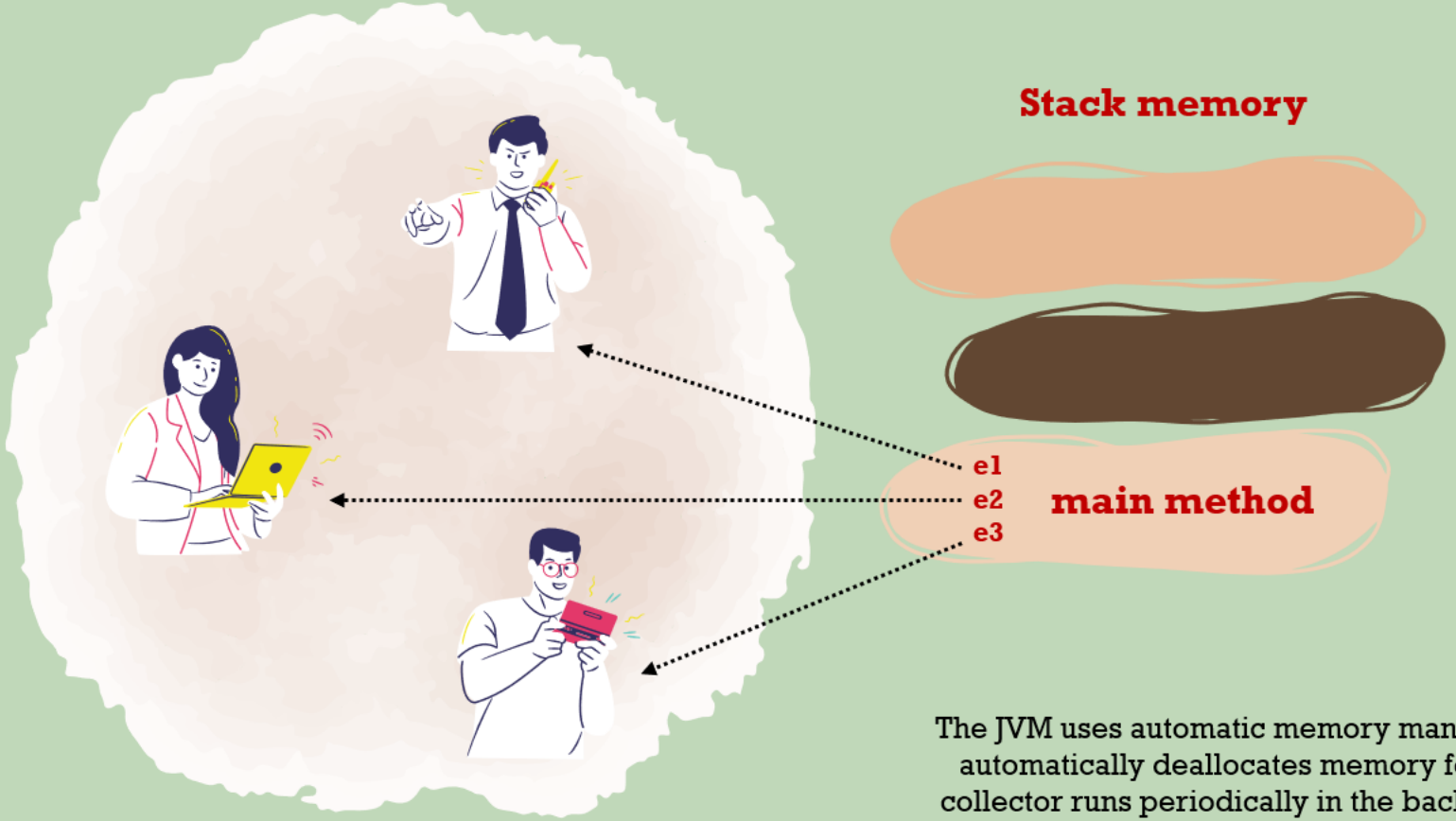
Native Heap: This is the area of memory where native libraries and code are loaded.

When your Java program is launched on a JVM, it receives a portion of memory from the underlying operating system, which is dependent on the JVM version and platform. Using the same memory JVM is going to create all the above memories. The amount of memory and the ability to modify it may vary, but typically you won't have control over it. However, if needed, the same can be customized

What happens when we create **Objects**

eazy bytes

The below statement creates the objects of Employee class in heap memory whereas the references like 'employee1' gets pushed to the stack under the method where the objects are being created. For example, if I create these objects inside my main() method, then this is how it looks inside Heap & Stack memory



1) Make a new reference of a class

3) Link the object & reference b/w heap & stack memory

```
Employee e1 = new Employee();
Employee e2 = new Employee();
Employee e3 = new Employee();
```

2) Create object using new. Magic happens here

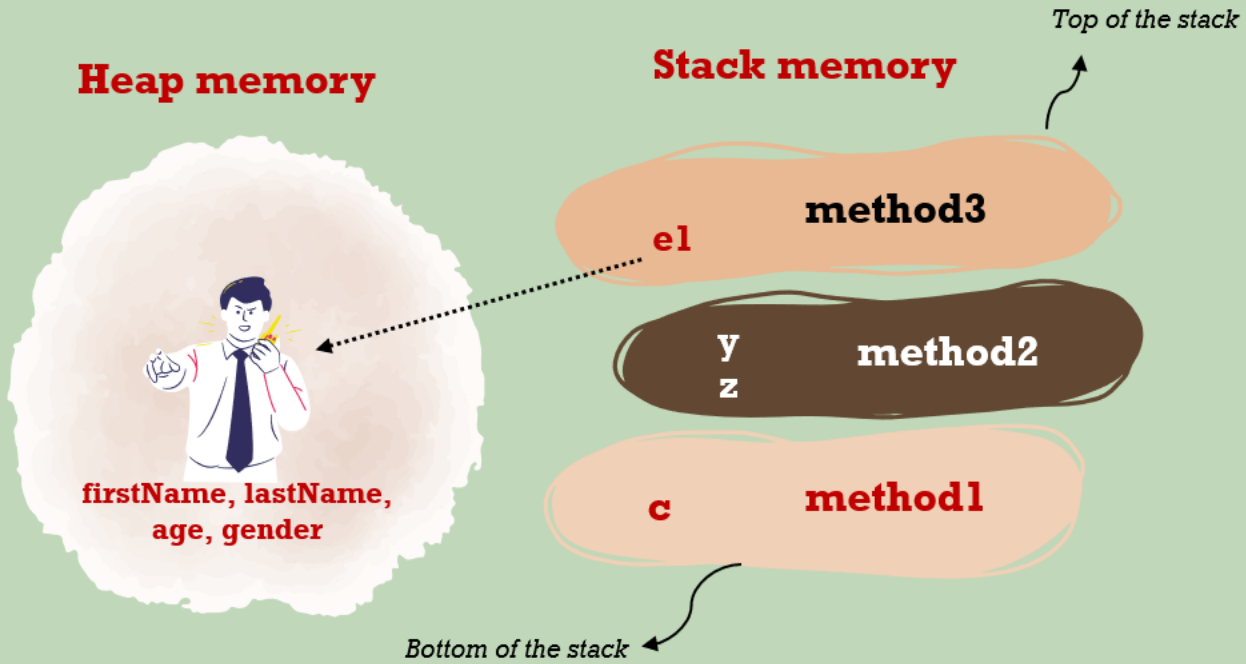
The JVM uses automatic memory management, which means that the JVM's garbage collector automatically deallocates memory for objects that are no longer being used. The garbage collector runs periodically in the background and identifies objects that are no longer being referenced by the program. It then deallocates the memory used by those objects.

Heap memory or **Garbage-collectible heap**

What exactly stored inside Stack memory

eazy bytes

Sample code snippet with three methods (let's not worry what the rest of the class looks like). The first method (method1())) calls the second method (method2()), and the second method calls the third (method3()). The methods has local variables of both primitive type & reference type, pass input to other methods. For this code, the heap & stack memory will look like as shown below



Sample method invocation

```
public void method1() {
  boolean c = true;
  method2(2);
}

public void method2(int y)
{
  int z = y + 24;
  method3();
}

public void method3() {
  Employee e1 = new
Employee();
}
```

Local Variables

Local variables are declared inside a method, including method parameters. They're temporary and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace). If the local variable is of type primitive, it store inside stack memory itself, where as if it is reference type, it will be refered to heap memory location using the object reference

Instance Variables

Instance variables are declared inside a class but not inside a method. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to which means inside the heap memory

eazy bytes

null keyword

In Java, null is a reserved keyword that denotes a variable or attribute that has a type but doesn't reference any object. Essentially, null signifies the absence of an instance or object associated with that variable or field.

Below are the examples of assigning a variable to null in Java:

```
String myString = null; // myString holds null value that is assigned
Employee employeeObject; // employeeObject holds null
```

It's important to note that trying to access a null reference will result in a NullPointerException at runtime. Therefore, you should always check whether a reference is null before using it to avoid these runtime errors.

Primitive data type fields cannot be null since they always have a default value assigned to them.

```
int number = null; // compilation fails
boolean isValid = null; // compilation fails
```

Object Destruction

eazy bytes

When you create things (objects) in a programming language like Java, you also need to know how to get rid of them when you're done using them. In other languages like C or C++, this can be tricky and can lead to memory leaks, where the computer's memory gets used up.

Java is different. You don't have to stress about manually getting rid of things you create. In some other languages, programmers have to carefully manage memory to avoid leaks, but Java handles it automatically. So, you can focus on more interesting parts of your code instead of worrying about memory management.

Garbage Collection in Java:

Garbage collection is like having a helper (garbage collector) in Java that keeps an eye on things you create. It looks at all the objects you've created and checks if they're still being used. If not, it removes them and frees up the memory they were using.

Initially, Java used a technique called "mark and sweep." It marked objects that were still being used, then swept away the ones not marked. It had some issues, like causing pauses in programs.

Nowadays, Java's garbage collectors are smarter and work continuously without causing delays. They divide memory into sections, placing short-lived objects separately for quicker cleanup. They can also adjust how they work based on how your program is running.

As a programmer, you usually don't need to worry about garbage collection. Java handles it for you. But if you want to force a cleanup, you can use System.gc(). It's not a guarantee, but it gives Java a nudge to clean up if needed.

Class vs Object vs Instance vs Reference

eazy bytes

In Java, "Class," "Object," "Instance," and "Reference" are four important concepts that are often used interchangeably, but they have distinct meanings.

Class: A class is a blueprint or template for creating objects that define the behavior and properties of objects. It describes the data members and methods that an object of that class will have. A class can be thought of as a template that defines the common attributes and behaviors of a group of objects.

Object: An object is an instance of a class. It is created from a class blueprint and has a unique set of values for its data members. It is a concrete entity that can be manipulated using its methods.

Instance: An instance is a single occurrence of an object at runtime. When an object is created, it is called an instance of the class. Object and instance are interchangeable terms

Reference: A reference is a variable that holds the memory address of an object. It is used to refer to the object and access its properties and methods. A reference does not hold the object itself but points to its location in memory.

Class vs Object vs Instance vs Reference

eazy bytes

A good analogy for understanding the concepts of Class, Object, Instance, and Reference in Java is to think of a house construction plan.

A class is like a house blueprint or construction plan. It defines the structure, design, and features of a house. It provides a blueprint for constructing a house that can be replicated many times.

An object or instance is like a physical house built from the construction plan. Each house built from the same plan has the same structure, design, and features. However, each house may have different decorations, furniture, or occupants, making them unique.

A reference is like a GPS coordinate that points to a particular house. It does not contain the house itself, but it provides a way to access the house's location and interact with it. Similarly, a reference in Java points to a specific instance of an object, allowing access to its properties and methods.

Class vs Object vs Instance vs Reference

eazy bytes

Heap memory


```
House house1 = new House(“Red”);
House house2 = new House(“Black”);
House houseOne = house1;
new House(”Green”);
```

When we write House house1 = new House(”Red"), we are actually creating a new object of the House class, which serves as a blueprint. The same applies for house2 as well.

Following the previous line, the code House houseOne = house1; generates another reference that points to the same object in memory as the house1 variable. Therefore, we end up with two references pointing to the exact same object in memory. This does not create a new house object, but rather a new way to refer to the same house object.

Since both houseOne & house1 are referring to the same memory address, any change that we make with one of the reference will impact the other reference as well.

The code "new House("Green");" creates a new instance of the House object and sets its color to green. However, we are not assigning this object to any variable in our code. Even though this code compiles without any errors, we cannot interact with this object after this line of code is executed because we have not created a reference to it. The object exists in memory, but our code has no way of accessing it since we did not assign it to any variable.

Encapsulation

eazy bytes

In Object Oriented Programming, encapsulation generally refers to two concepts.

Firstly, it involves grouping attributes(fields) and behaviors(methods) together within a single object (Class). This is very similar to capsules which contain the active ingredient, along with other substances that help to stabilize and preserve the drug.

Secondly, it involves the practice of hiding certain fields and methods from public access using access modifiers like private, protected etc. This is very similar to a brief case where we hide certain data inside it in a secured manner.