



# Object-Oriented Programming - Abstraction

**Course:** Object-Oriented  
Analysis, Design and  
Programming

**Lecture On:** Object-Oriented  
Programming - Abstraction

# Topics covered in the previous class...

1. OOA, OOD and OOAD and their need
2. Procedural Programming vs Object-Oriented Programming
3. Introduction to the Unified Modeling Language (UML)
4. How to read requirement docs and identify objects, attributes and methods
5. The concept of a class and how to draw one using diagrams.net
6. The concepts of visibility and multiplicity for class members
7. Different class relationships: Association, Generalization, Aggregation and Composition

## Poll 1 (15 sec)

Suppose there are two classes in an application—Animal and Dog. You can say that a Dog is an Animal. What would be the relationship between them?

1. Association
2. Generalization
3. Aggregation
4. Composition

## Poll 1 (15 sec)

Suppose there are two classes in an application—Animal and Dog. You can say that a Dog is an Animal. What would be the relationship between them?

1. Association
- 2. Generalization**
3. Aggregation
4. Composition

# Homework Discussion

# Today's Agenda

- **Provide classes for the Inventory Management System**
  - Understand Abstraction: the first fundamental principle of OOP
  - How to declare classes, attributes and methods
  - How to create objects based on the classes
  - How objects are stored in the memory
  - The concept of a constructor and how to provide one
  - Different types of constructors
  - How to use the 'this' keyword

# First Fundamental Principle of OOP: Abstraction



- In Object-Oriented Programming, Abstraction is defined as follows:

***Abstraction denotes the essential characteristics of an object relative to the perspective of the viewer.***

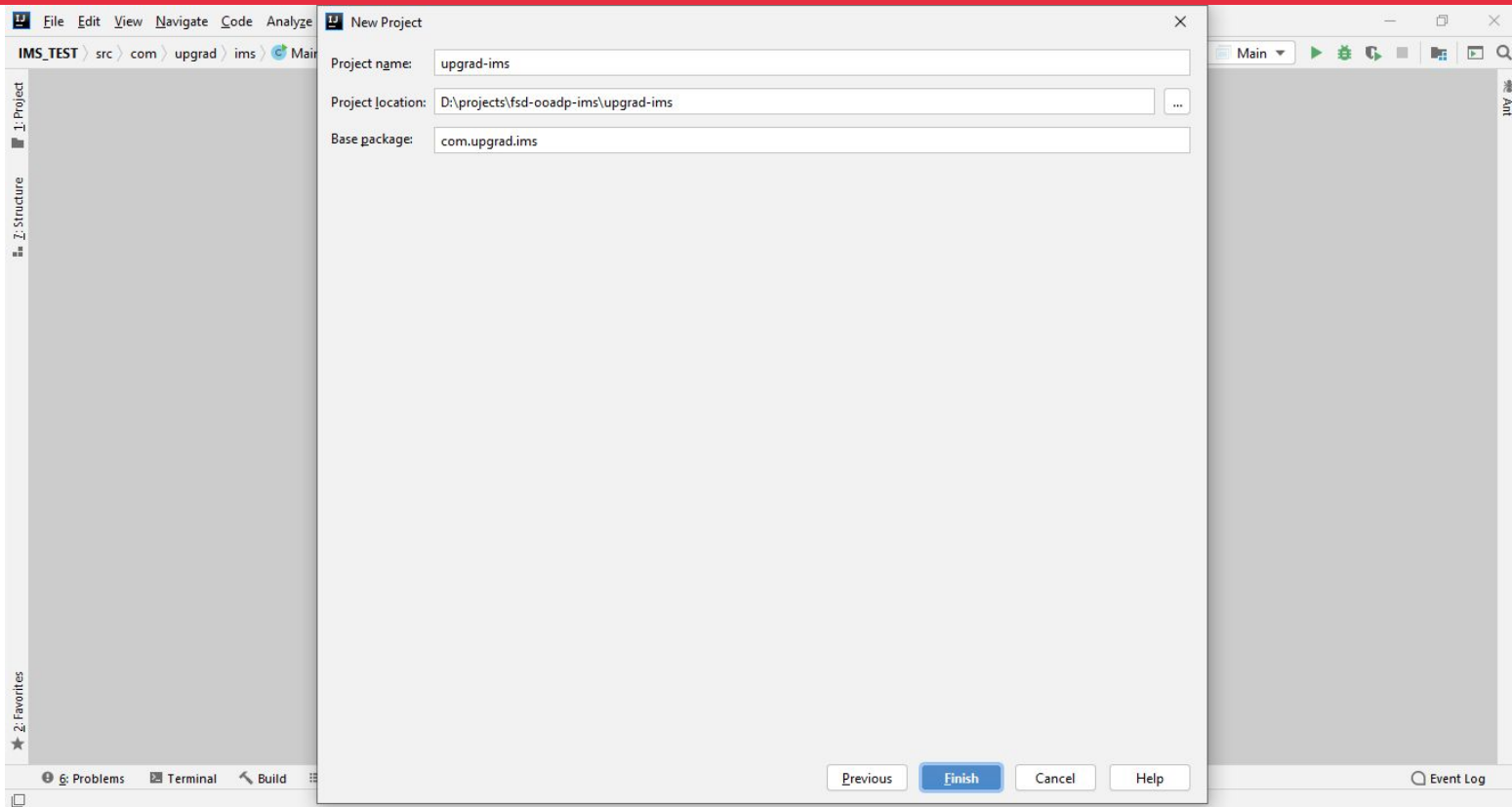
- For example, as humans, we have several important characteristics such as name, height, weight and date of birth.
- However, our essential characteristics may differ based on the role that we are playing.

- If we are playing the role of a Student, then our essential characteristics would be our roll number, gender, courses in which we are enrolled, degree we are pursuing and so on.
- If we are playing the role of an Employee, then our essential characteristics would be our Employee Code, PAN, designation and so on.
- If we are playing the role of a Vendor, then our essential characteristics would be vendor name, credit, product list that we sold and so on.
- Thus, you can observe that ***Abstraction focuses on the essential characteristics of an object based on the context or the role that the object is playing.***

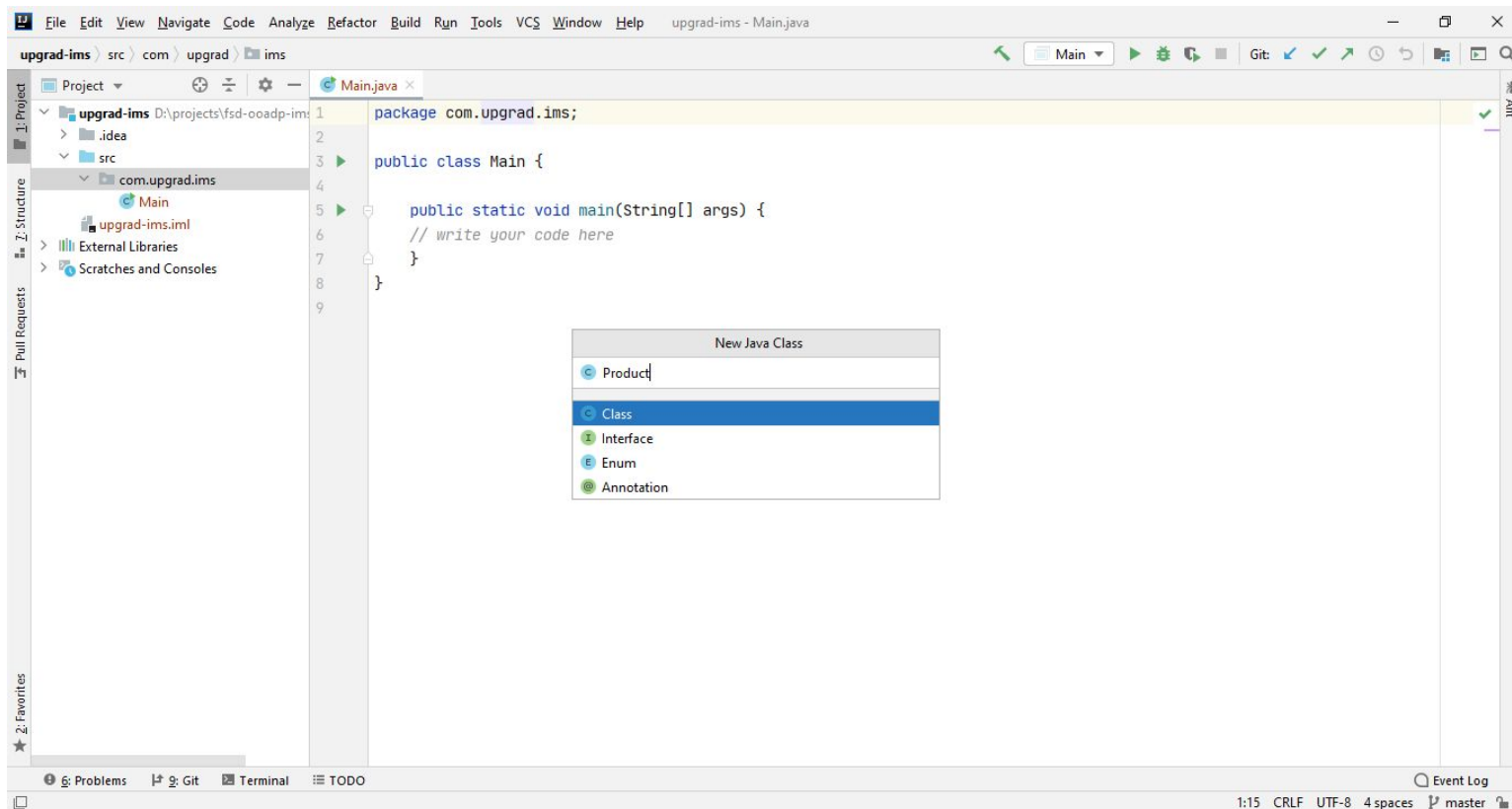
- In Object-Oriented Programming, Abstraction is achieved mainly in the following two ways:
  - **Entity Abstraction:** This refers to modelling an entity in the system based on the essential characteristics. This is done by providing classes for the different entities of our system.
  - **Action Abstraction:** This is done by providing abstract methods (methods with only the name, return type and parameter list, without a body). We will discuss this in detail while learning about inheritance, which is the third fundamental principle of OOP.

- In this session, you will learn about Entity Abstraction, which is achieved by providing classes for the problem domain entities (the entities that we need to represent in the system).
- So, the Product class is an abstraction for all the products that will be present in the Inventory Management System.
- Similarly, the Vendor class will be an abstraction for all the vendors that will be present in the Inventory Management System.
- We have already identified most of the classes. As the course progresses, we will create more classes as requirement arises.
- Now, you will learn how to create classes while writing code.

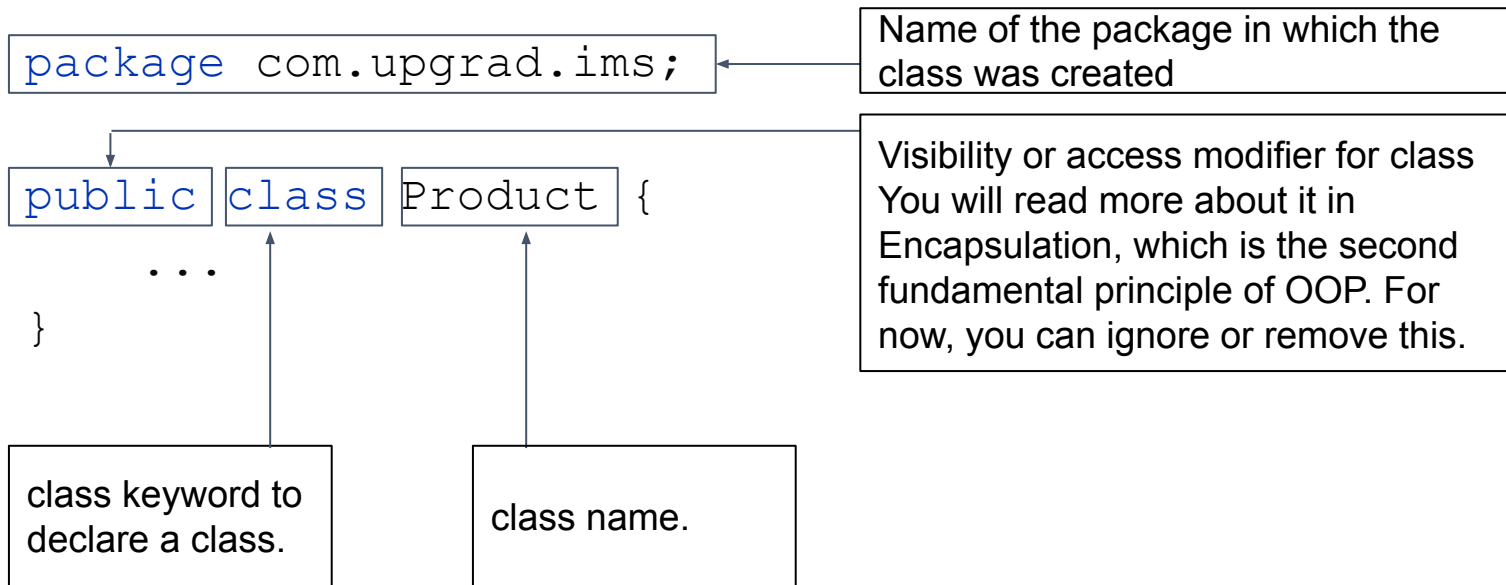
# New Project



- Let's start with the Product class.
- We have already identified all the attributes and methods related to the Product class.
- We will be adding or customising existing methods based on the need.
- Each class is provided in a separate file with the following two rules:
  - The name of the file should be the same as the class name.
  - The file extension should be '.java'.
- We can also provide more classes in the same file, but there are some limitations to that; you will learn about these later in the course.
- Let's create the Product class in the 'com.upgrad.ims' package.



- When you hit enter, IntelliJ will create a new file Product.java in the 'com.upgrad.ims' package with the following code snippet:





- So, the basic syntax to create an empty class is as follows:

```
class ClassName {  
}
```

- The class body (attributes and methods) will be between the curly braces after the ClassName.

```
class ClassName {  
    // attributes and methods  
}
```

- The attributes to a class are declared in the following way:

```
class ClassName {  
    int attribute1;  
    float attribute2;  
}
```

- We simply list down all the attributes along with their type.
- Each attribute declaration needs to end with a semicolon.

- Let's provide attributes to the Product class that we identified while drawing the class diagram.

```
public class Product {  
    int id;  
    String name;  
    String category;  
    float salesPrice;  
    float cost;  
    int quantity;  
    boolean active;  
}
```

- The methods to a class are declared in the following way:

```
class Product {  
    int attribute1;  
    ...  
  
    float methodName(int parameter1) {  
        return parameter1 * 2.0;  
    }  
}
```

- Note that method definitions must **NOT** end with a semicolon.

- Methods and attributes can be declared in any order.
- You can declare attributes before the methods, after the methods or a mix of both.
- As per convention, all the attributes are declared first, followed by the method definitions.
- Let's provide all the methods for the Product class.

```
public class Product {  
    int id;  
    ...  
  
    float getProfitOrLoss() {  
        return salesPrice - cost;  
    }  
  
    void activate() {  
        active = true;  
    }  
  
    ...  
}
```

[Code Reference](#)

## Poll 2 (15 sec)

What is Abstraction?

1. Identifying objects from the requirement doc
2. Providing classes for the objects

## Poll 2 (15 sec)

What is Abstraction?

1. Identifying objects from the requirement doc
- 2. Providing classes for the objects**



## Poll 3 (15 sec)

What is incorrect in the following class syntax?

```
class User {  
    userId int;  
}
```

1. The 'class' keyword should be 'Class'.
2. There are no methods.
3. int should be written before userId.
4. The class name should be followed by parentheses.

## Poll 3 (15 sec)

What is incorrect in the following class syntax?

```
class User {  
    userId int;  
}
```

1. The 'class' keyword should be 'Class'.
2. There are no methods.
- 3. int should be written before userId.**
4. The class name should be followed by parentheses.

## Poll 4 (15 sec)

Which of the following is the correct way to define a method inside a class?

1. `changeNameTo(newName: String) : void;`
2. `void changeNameTo(String newName);`
3. `void changeNameTo(String newName) { ... }`
4. `void changeNameTo(String newName) { ... };`

## Poll 4 (15 sec)

Which of the following is the correct way to define a method inside a class?

1. `changeNameTo(newName: String) : void;`
2. `void changeNameTo(String newName);`
3. **`void changeNameTo(String newName) { ... }`**
4. `void changeNameTo(String newName) { ... };`

You learnt how to create classes, declare attributes and define methods. Now, provide a class for the Customer objects.

## **TODO:**

- Use the following command to check out to the current state:  
`git checkout d3a727d`
- Create a new Java class and name it Customer.
- Use the class diagram to provide attributes and methods to the Customer class.
- For now, do not create separate classes for BusinessPartner, Contact and Address and provide all the attributes and methods inside the Customer class.

## TODO:

- The calculateDiscount() methods should be implemented as shown below:
  - $\text{Discount} = \text{num of transaction} * 0.1;$
  - However, the discount cannot be more than 10% in any case.
- The getAddressDetails() method should return the following value:
  - street, city, state
- The getContactDetails() method should return the following value:
  - name, phone, email
- For now, do not pay attention to the updateAddressDetails() and updateContactDetails() methods.

[Code Reference](#)

# Object Creation

- So, we provided the Product class and the Customer class.
- Now, you will learn how you can create an Object out of these classes.
- We can create an object of a class using the following syntax:

```
ClassName variableName = new ClassName();
```

- So, if you want to create an object of the Product class, you need to do it as shown below:

```
Product product = new Product();
```

Object Declaration

Object Instantiation

Object Initialization

*Note: You will learn about these three later in the session.*



- Now, you can access any property or method of the newly created objects using the dot operator.

```
product.id = 10;
```

```
float getProfit = product.getProfitOrLoss();
```

[Code Reference](#)

You learnt how to create objects out of classes and access their attributes and methods. Now, create the Customer object and access its attributes and methods.

## **TODO:**

- Use the following command to check out to the current state:  
git checkout bc14c90
- Create Customer objects.
- Use the dot operator to access its attributes and methods.
- Print values of various attributes and methods on the console.

[Code Reference](#)

# Stack and Heap Memory

- Earlier, you learnt about the following three parts of object creation: object declaration, object instantiation and object initialization.
- Object declaration is the same as primitive data type declaration. It provides the variable type followed by the variable name.



- Now, you will learn about the object instantiation part.
- For that, you first need to understand two types of JVM (Java Virtual Machine) memories—Stack and Heap.

- Whenever you call a method, JVM will create a Stack data structure and push all the attributes into that Stack along with any variable that you declare while that method is getting executed.
- Consider the following method:

```
int add (int a, int b) {  
    int result = a + b;  
    return result;  
}
```

- Suppose you call the add() method as shown below:

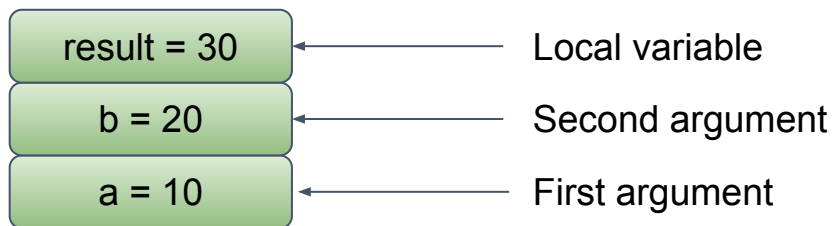
```
int sum = add(10, 20);
```

- When JVM encounters this statement, it will create a stack data structure as shown below:



- Each entry in the stack contains the data and the type of that data (here, int type).

- During the execution of the add() method, when JVM notices that you are declaring one more variable named result, it will push one more entry in the stack corresponding to that method.



- When JVM hits the closing curly braces for the method, it deletes this data structure along with all the entries inside it. This is the reason why you cannot access the local variable once the method is completed.

- When you pass primitive variables to the method call, the variables themselves are not passed to the method call—only their value gets copied.
- This is also called ***Passed By Value***, as the variables themselves are not passed to the method call—only their value is.



```
void method() {  
    → int a = 10, b = 20;  
    swap(10, 20);  
    System.out.println("a: " + a + ", b: " + b);  
}
```

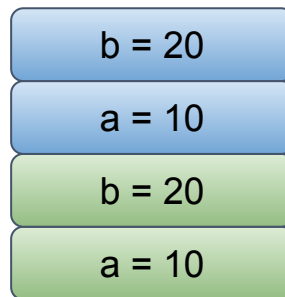
```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

b = 20


a = 10

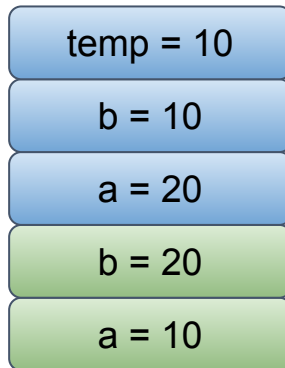
```
void method() {  
    int a = 10, b = 20;  
    → swap(10, 20);  
    System.out.println("a: " + a + ", b: " + b);  
}
```

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```



```
void method() {  
    int a = 10, b = 20;  
    swap(10, 20);  
    System.out.println("a: " + a + ", b: " + b);  
}
```

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
     b = temp;  
}
```



```
void method() {  
    int a = 10, b = 20;  
    swap(10, 20);  
    System.out.println("a: " + a + ", b: " + b);  
}
```

```
void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

b = 20

a = 10

- Thus, for primitive variables, whenever you declare a new variable and assign another variable to it, the value gets copied.

```
int temp = a;
```

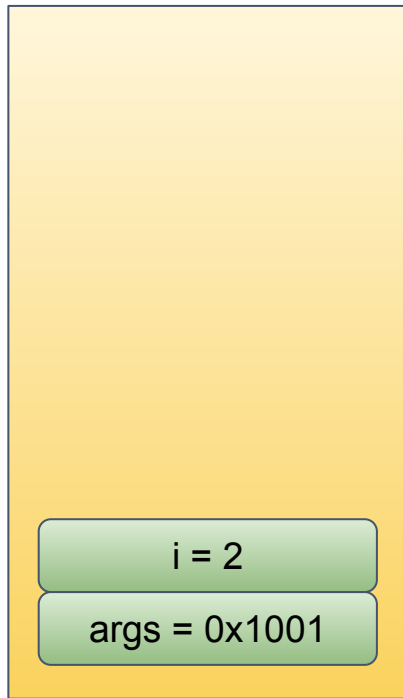
Here, the value of a is copied to the temp.

- This is okay for primitive data types, as they acquire a small piece of the memory. So, copying a value every time is not that big a memory overhead or performance overhead.
- However, for objects, which themselves contain numerous primitive data types and probably other objects, copying the value every time would cost both in terms of memory and performance.

- Therefore, objects are stored in the Heap memory, which is common for the entire application.
- When you create a new object, the object is created in the heap, and its reference is stored in the stack of the method where it was created.
- When you pass this object variable to other methods, only the reference is copied and passed to the method. So, the new method can access this object. This is called ***Passed By Reference***, as the reference is getting copied, not the value itself.
- You will learn about this through an example.

```
public static void main(String[] args) {  
    ➔ int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

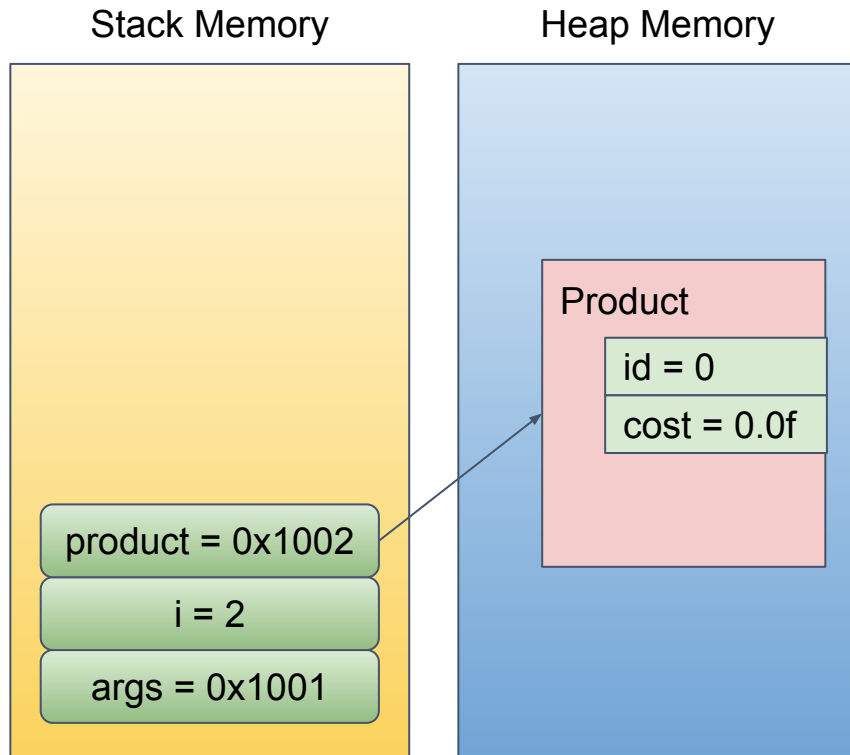
Stack Memory



Heap Memory

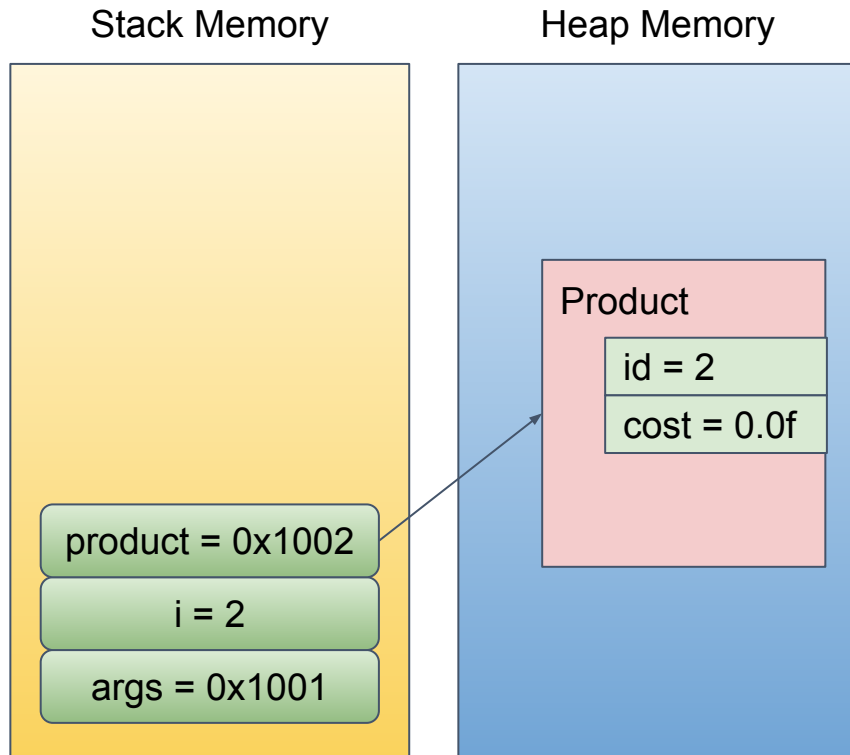


```
public static void main(String[] args) {  
    int i = 2;  
    ➔ Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

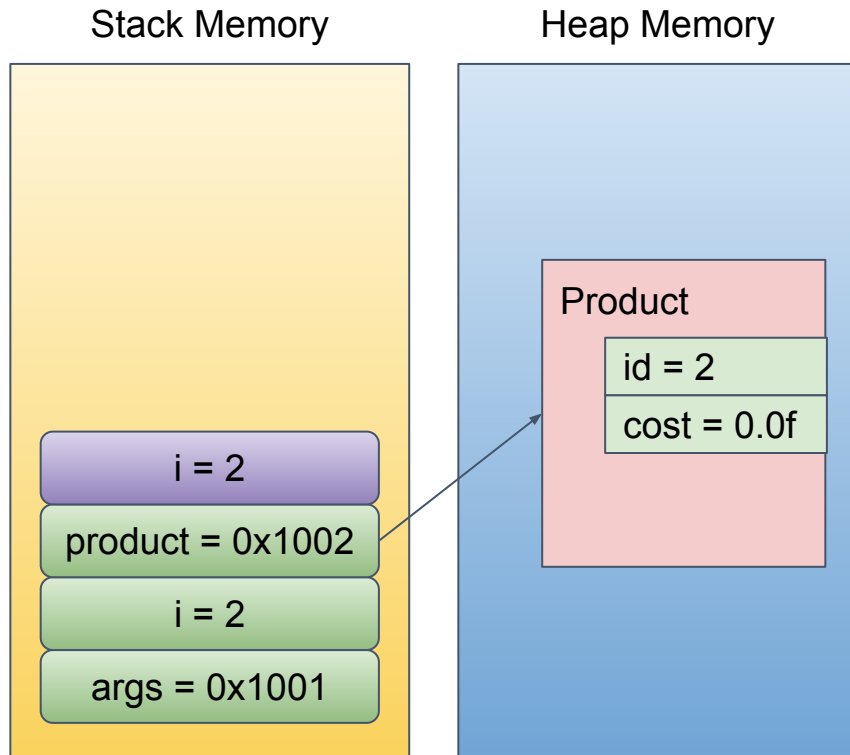




```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    ➔ product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

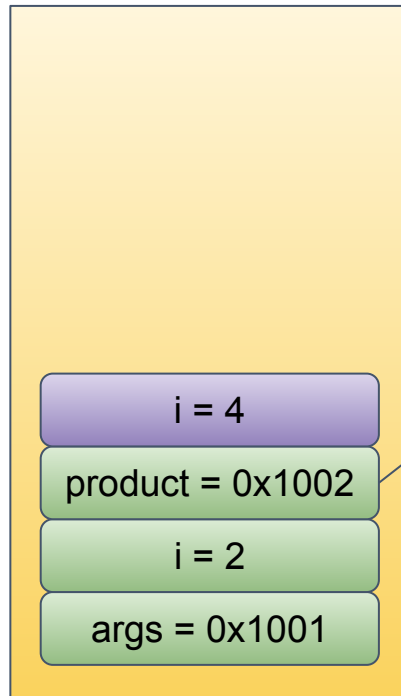


```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    ➔ makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

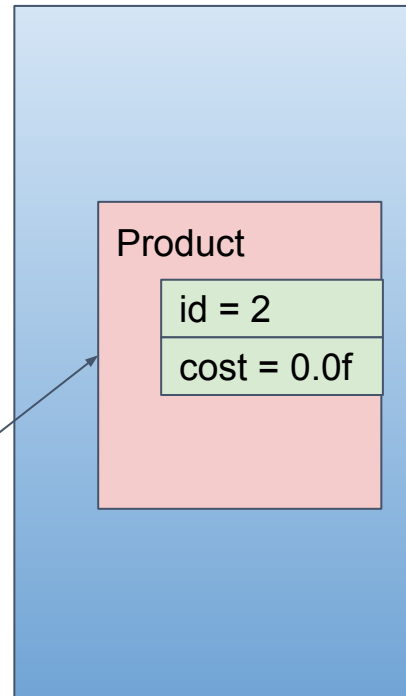


```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    → i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

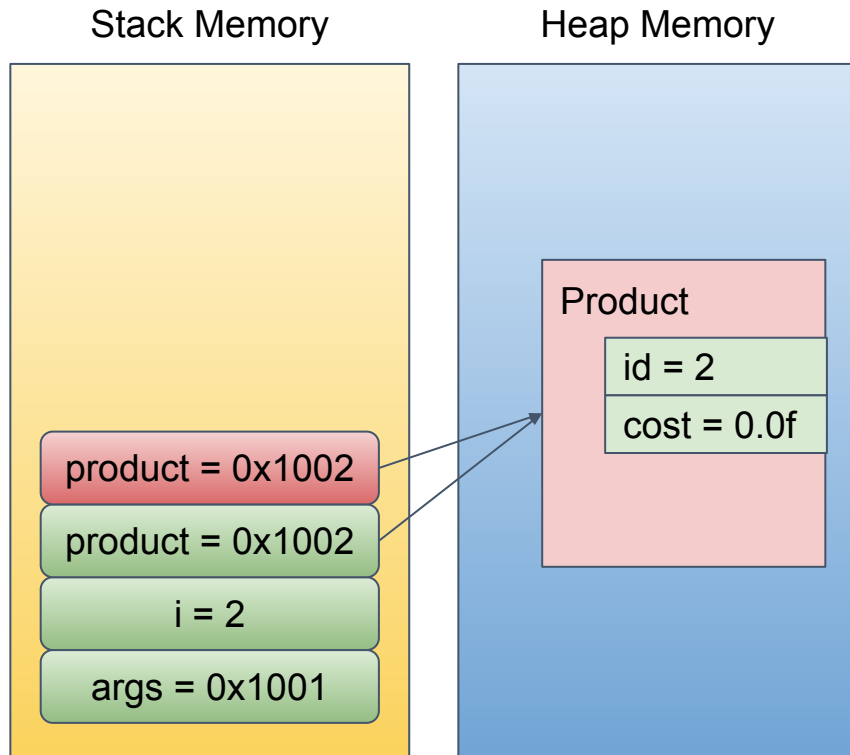
Stack Memory



Heap Memory

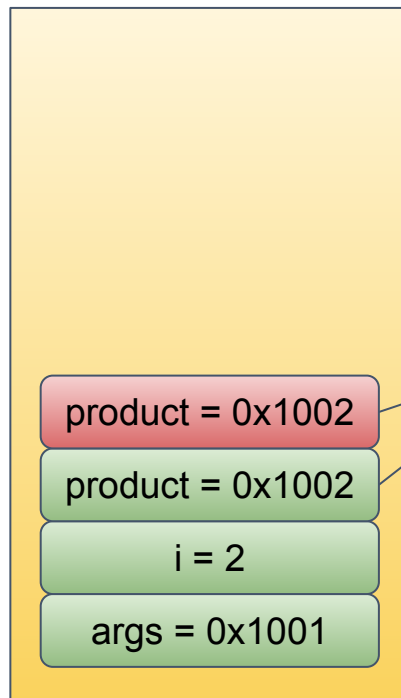


```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    ➔ makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

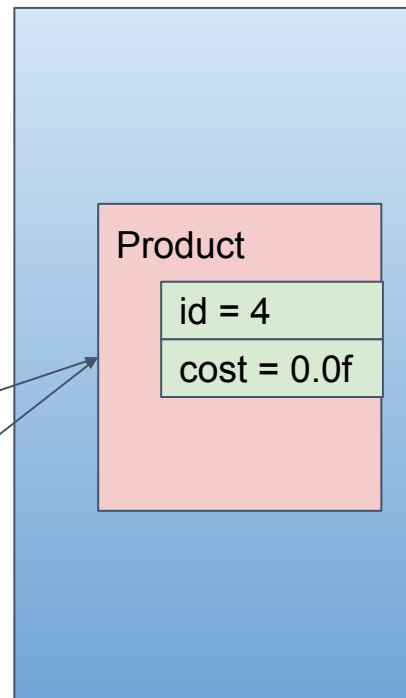


```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    → product.id = 2 * product.id;  
}
```

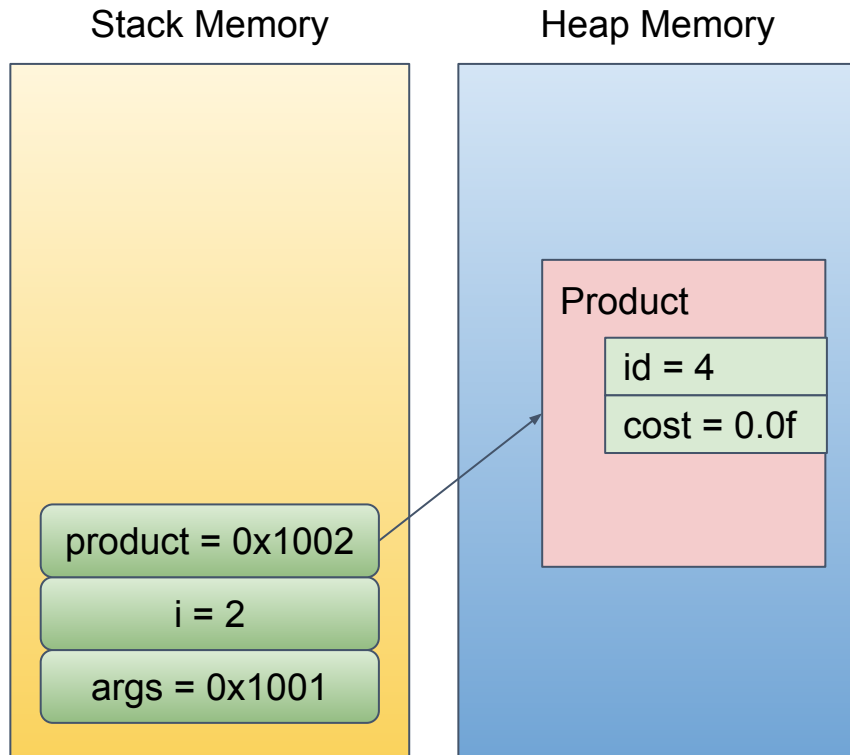
Stack Memory



Heap Memory



```
public static void main(String[] args) {  
    int i = 2;  
    Product product = new Product();  
    product.id = 2;  
  
    makeValueDouble(i);  
    makeIdDouble(product);  
  
    System.out.println(i);  
    System.out.println(product.id);  
}  
  
static void makeValueDouble(int i) {  
    i = i * 2;  
}  
  
static void makeIdDouble(Product product) {  
    product.id = 2 * product.id;  
}
```

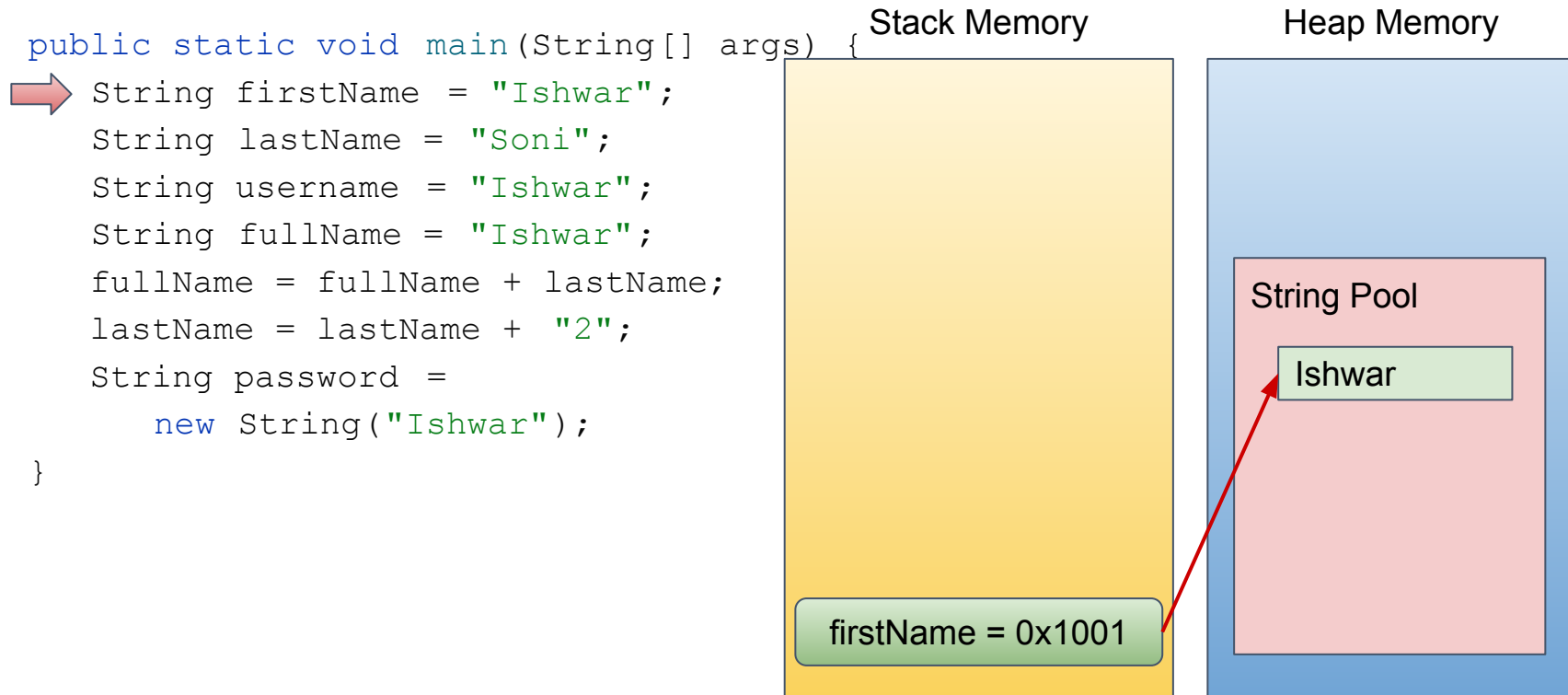


- As you can observe, you can only declare and initialise primitive data types, as memory allocation for them is done in the stack.
- However, for the object type, you need to first declare them (create a variable), instantiate them using the new operator (allocate memory in the heap) and then initialise them (provide values to the object attributes).
- You will learn how you can initialise objects when you learn about constructors.

- String is also a class that is the most used to create objects.
- In some big programs, there can be thousands or even millions of String objects at any moment.
- So, to optimise this, JVM only creates one String object for any given String literal.
- Also, all of the Strings are stored in the ***String Pool***, which is a special memory area in the heap memory.
- If you create a new String variable with the same value, JVM will search whether the same String literal exists in the String Pool; if yes, then it returns the reference to it. Otherwise, it creates a new String object and returns the reference to that.

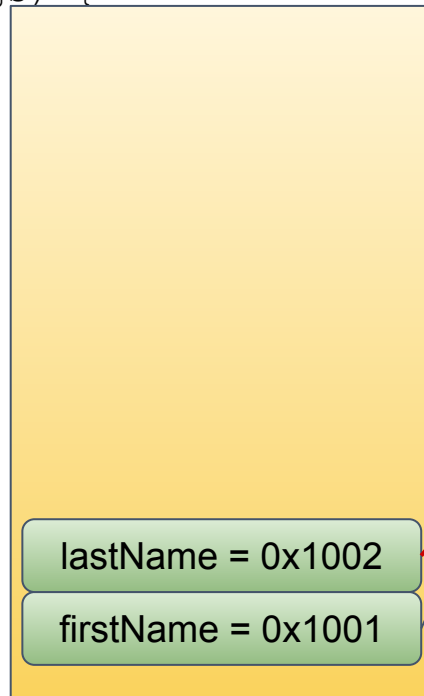


- However, if multiple String variables are pointing to the same object and one of them changes the value assigned to it, will the value be changed for other variables as well?
- This is where the immutable nature of Strings comes into play. Whenever you try to change the value of a string, you are actually creating a new String object.
- However, there is an exception. When you create String objects using the new keyword, JVM will create a new object whether or not a String object exists with the same value. That is why you should always create String objects using the String literals.
- You will learn about this through an example.

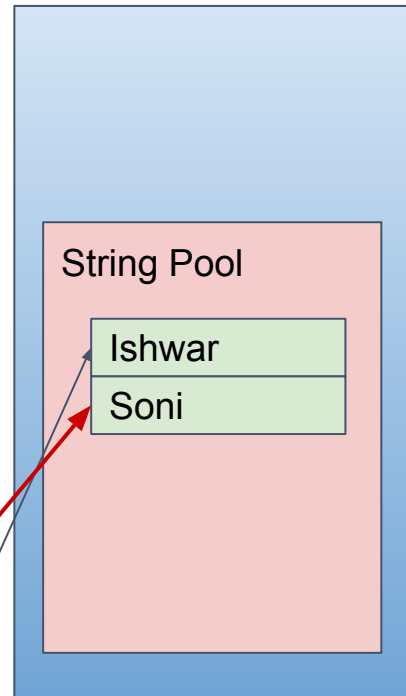


```
public static void main(String[] args) {  
    String firstName = "Ishwar";  
    ➔ String lastName = "Soni";  
    String username = "Ishwar";  
    String fullName = "Ishwar";  
    fullName = fullName + lastName;  
    lastName = lastName + "2";  
    String password =  
        new String("Ishwar");  
}
```

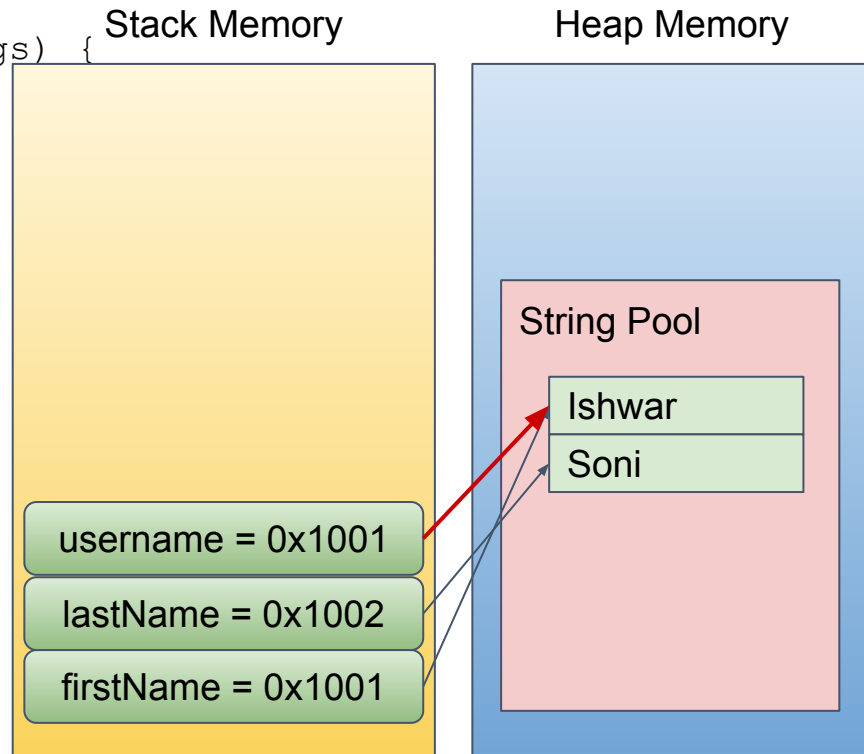
Stack Memory



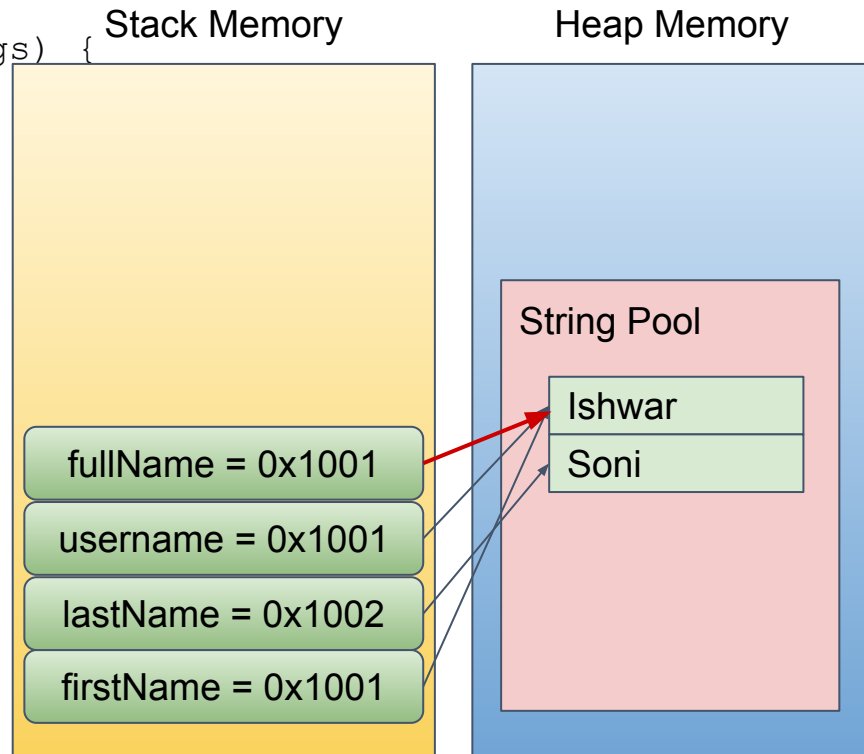
Heap Memory

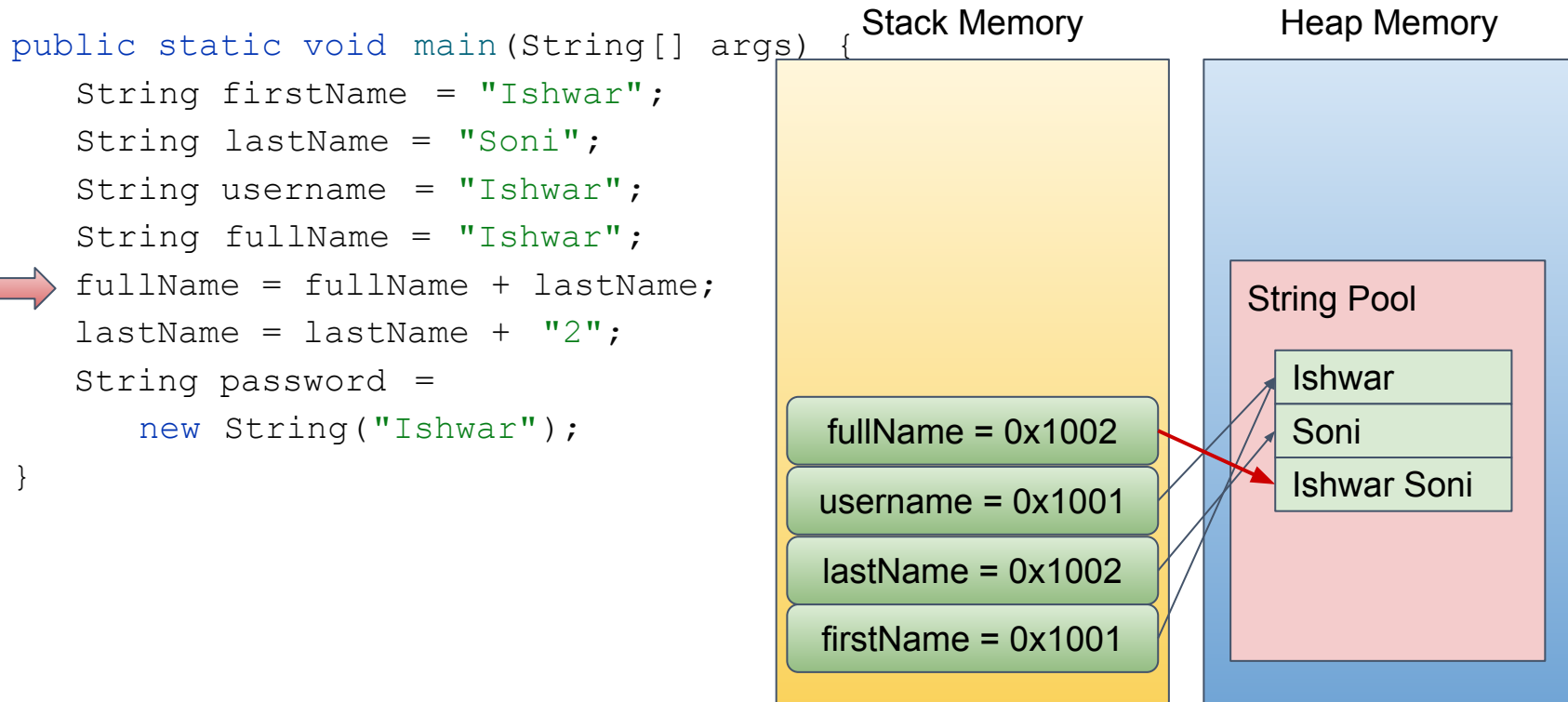


```
public static void main(String[] args) {  
    String firstName = "Ishwar";  
    String lastName = "Soni";  
    ➔ String username = "Ishwar";  
    String fullName = "Ishwar";  
    fullName = fullName + lastName;  
    lastName = lastName + "2";  
    String password =  
        new String("Ishwar");  
}
```

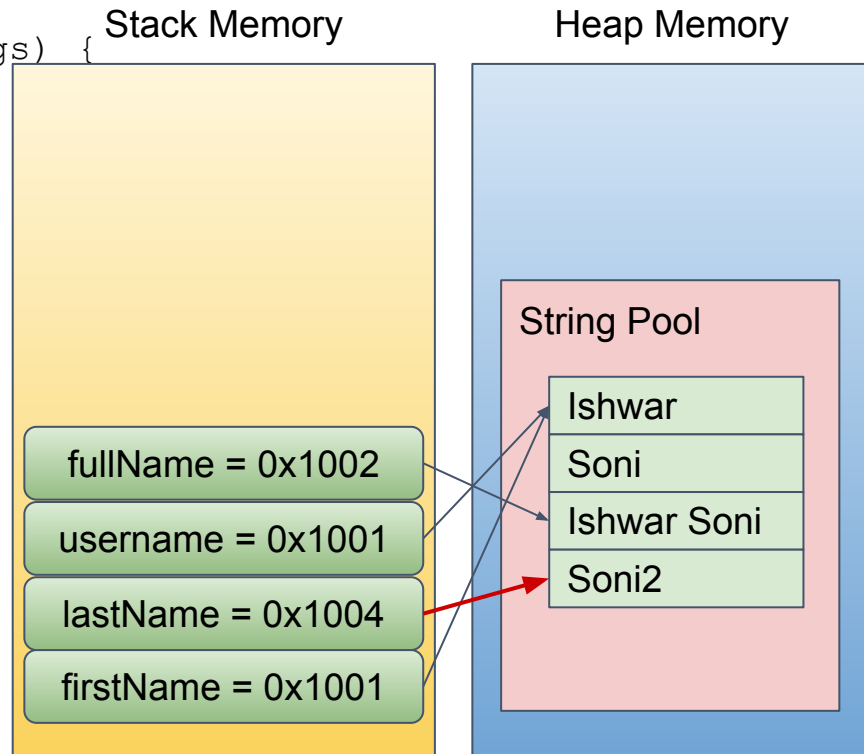


```
public static void main(String[] args) {  
    String firstName = "Ishwar";  
    String lastName = "Soni";  
    String username = "Ishwar";  
    ➔ String fullName = "Ishwar";  
    fullName = fullName + lastName;  
    lastName = lastName + "2";  
    String password =  
        new String("Ishwar");  
}
```

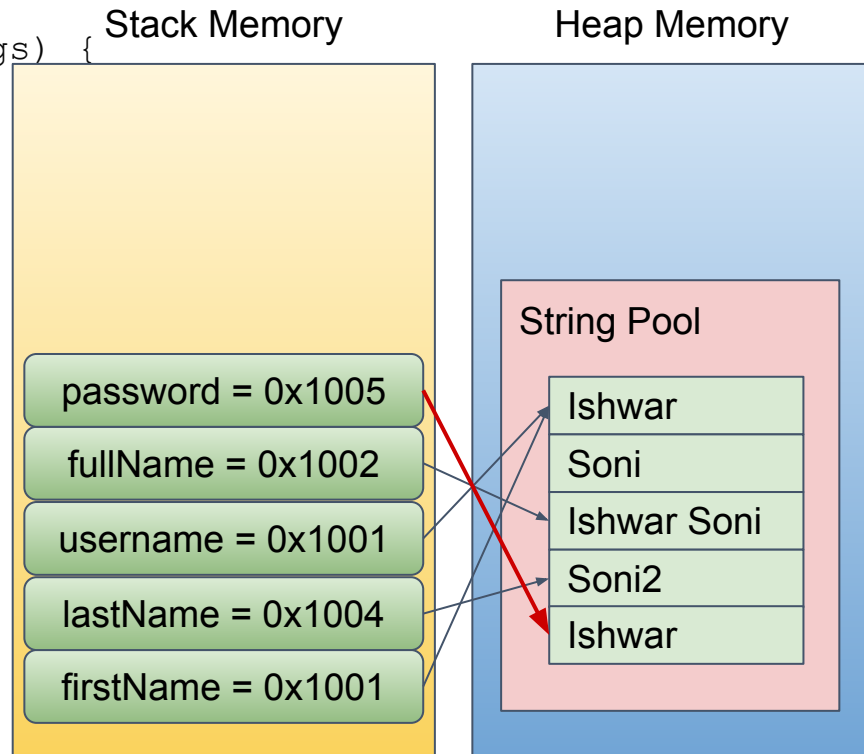




```
public static void main(String[] args) {  
    String firstName = "Ishwar";  
    String lastName = "Soni";  
    String username = "Ishwar";  
    String fullName = "Ishwar";  
    fullName = fullName + lastName;  
    ➔ lastName = lastName + "2";  
    String password =  
        new String("Ishwar");  
}
```



```
public static void main(String[] args) {  
    String firstName = "Ishwar";  
    String lastName = "Soni";  
    String username = "Ishwar";  
    String fullName = "Ishwar";  
    fullName = fullName + lastName;  
    lastName = lastName + "2";  
    ➡ String password =  
        new String("Ishwar");  
}
```





- You can assign the null to object variables, either during initialisation or once it has been assigned any other value.
- When you assign the null value, that object variable will point to 'no memory' in the heap.

```
Product product = null;
```

- In the example given above, the product variable points to the 'no memory' location in the heap.
- You can think of null as a special literal that can be used to specify that the object variables do not point to any memory in the heap.

- In Java, Objects and Instances are used interchangeably, as both mean the same because you cannot create objects without first instantiating a class using the new operator.
- In other languages such as JS, the meanings of objects and instances are different, as you can create objects without first declaring a class.
- For example, suppose you create five objects of the Product class by instantiating the Product class five times using the new operator.
- Then, we say that all of them are product objects (as they represent the product entity), but they are five different instances (as they refer to five different memory locations in Heap).
- As per convention, in Java, you should use 'instance' instead of 'object'.

## Poll 5 (15 sec)

Which of the following is true with respect to the Stack and Heap memory? (Note: More than one option may be correct.)

1. The Stack memory is separate for each method.
2. The Heap memory is common for the entire application.
3. Primitive data variables are stored in the heap memory.
4. Objects are stored in the stack memory.

## Poll 5 (15 sec)

Which of the following is true with respect to the Stack and Heap memory? (Note: More than one option may be correct.)

- 1. The Stack memory is separate for each method.**
- 2. The Heap memory is common for the entire application.**
3. Primitive data variables are stored in the heap memory.
4. Objects are stored in the stack memory.

## Poll 6 (15 sec)

Which of the following is true with respect to the String pool?

1. The String pool memory is allocated inside the stack memory.
2. In String pool, there can exist only one String object with the same value.
3. You cannot change the value of the String objects inside the String pool.
4. A String object inside the String pool can only be referenced by one String variable.

## Poll 6 (15 sec)

Which of the following is true with respect to the String pool?

1. The String pool memory is allocated inside the stack memory.
2. In String pool, there can exist only one String object with the same value.
3. **You cannot change the value of the String objects inside the String pool.**
4. A String object inside the String pool can only be referenced by one String variable.

# Constructors

- Earlier, you learnt about the following three parts of object creation: object declaration, object instantiation and object initialization.

```
Product product = new Product();
```

Object Declaration

Object Instantiation

Object Initialization

- Object declaration is the same as the primitive data type declaration, wherein you provide the variable name preceded by the variable type.
- You also learnt about Object instantiation using the new operator, which allocates memory in heap and returns the reference to that memory.



- Now, you will gain an understanding of object initialization and learn how it is performed using a constructor.
- A constructor is a special type of class member that is used to initialise the object (provide initial values to the attributes for an object).
- Each class has a constructor, whether you provide one or not. If you do not provide one, it is provided by the compiler.
- A constructor looks like the following:

```
ClassName (/*parameter list*/) {  
    //body  
}
```

- Constructors look similar to methods, with only the following three differences:
  - They do not have a return type.
  - They have the same name as the class name.
  - You cannot call them using the dot operator. They are called automatically during class instantiation and initialization.
- When you execute the following statement, it calls the constructor of the Product class:

```
Product product = new Product();
```

↑  
It calls the constructor that matches with the signature, which means a constructor with no arguments (no args constructor).

- As you did not provide any constructor inside the Product class, the compiler will provide a no arg empty constructor as shown below:

```
Product () {  
  
}
```

- The constructor that is provided by the compiler is also called the **default constructor**.
- Remember that the default constructor is only provided for the classes without a constructor.
- If you provide a constructor for the Product class with a different syntax (with some parameter), then your code would fail.

- Let's provide a constructor with one parameter, as shown below, and check what happens to the code:

```
Product (int id) {  
  
}
```

- Now, the compiler will throw the following error:

```
Product appleMacBookAir = new Product();
```

Expected 1 arguments but found 0

Create constructor Alt+Shift+Enter

More actions... Alt+Enter

- Mainly, constructors are of the following three types:
  - Default constructors: Provided by the compiler for those classes where the constructor is missing
  - No arg constructor: Provided by the developer, with no arguments
  - Parameterized constructor: Provided by the developer, with at least one argument (or parameter)
- You can provide any number of constructors for a class with different signatures. Providing more than one constructor is called **Constructor Overloading**.
- Since the signature needs to be different for each constructor, you can only provide one no arg constructor.

- Let's provide a no arg constructor for the Product class, and you will learn how it works.

```
Product () {  
    System.out.println("Created Product Object");  
    id = 1;  
    name = "default name";  
    category = "default category";  
}
```

- Add the following code to the main method:

```
Product appleMacBookAir = new Product();  
System.out.println(appleMacBookAir.name);
```

- On running the code, you will obtain the following output:

*Created Product Object*  
*default name*

- It is printing the “Created Product Object”, which means that the No Arg constructor that we provided is getting called.
- Also, the name is printed as “default name”, which means that we can initialise the attribute values using the constructor.

[Code Reference](#)

- No arg constructors are not very helpful, as you cannot pass the values to initialise attributes.
- This is where the parameterized constructor helps you.
- You can provide any number of parameterized constructors, but all of them should have different signatures.
- Using parameterized constructors, you can pass values for the attributes to initialise.
- Let's provide a parameterized constructor for the Product class and use it inside the main() method to instantiate and initialise the product object.



```
Product (int _id, String _name, String _category,  
        float _salesPrice, float _cost, int _quantity,  
        boolean _active) {  
    id = _id;  
    name = _name;  
    category = _category;  
    salesPrice = _salesPrice;  
    cost = _cost;  
    quantity = _quantity;  
    active = _active;  
}
```

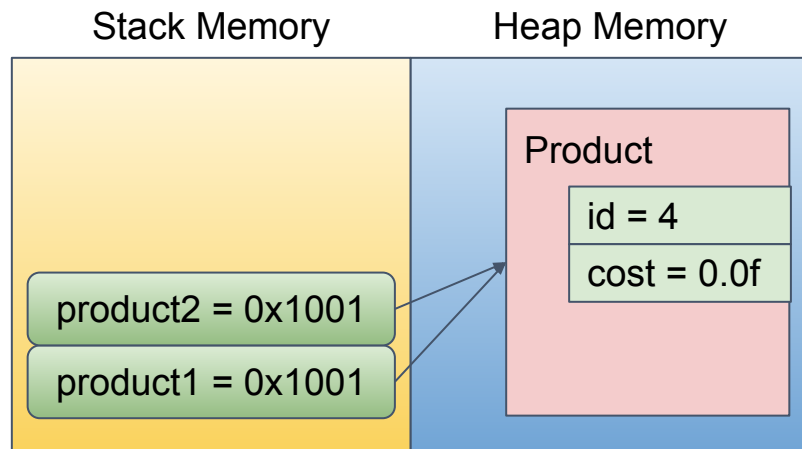
- Let's call the constructor inside the main() method. If the number of constructors is more than one, JVM will match the signature and call the appropriate one.

```
Product appleMacBookAir =  
    new Product(1, "Apple MacBook Air", "Laptop",  
                80000f, 60000f, 1000, true);
```

[Code Reference](#)

- Earlier, you learnt that when you assign an object variable to another one, only the reference gets copied, not the object itself. Thus, both variables end up pointing to the object in the heap memory. This is called ***shallow copying***.
- For example:

```
void method () {  
    Product product1 = new Product();  
    Product product2 = product1;  
}
```



- However, there do arise some situations wherein you want to create a clone of an existing object, which is also called ***deep copying***.
- There are several ways to achieve this, and one of them is using the copy constructor.
- In the copy constructor, you provide the object that you want to clone as an argument, and inside the constructor, you use that object to initialise the attribute to the new clone objects.
- You will now learn how to provide a copy constructor.

```
Product (Product product) {  
    id = product.id;  
    name = product.name;  
    category = product.category;  
    salesPrice = product.salesPrice;  
    cost = product.cost;  
    quantity = product.quantity;  
    active = product.active;  
}
```

- Let's use the copy constructor to create a clone of an existing object.

```
Product clone = new Product(appleMacBookAir);  
System.out.println(clone.name);  
System.out.println(appleMacBookAir.name);
```

```
clone.name = "clone";  
System.out.println(clone.name);  
System.out.println(appleMacBookAir.name);
```

[Code Reference](#)

## Poll 7 (15 sec)

Which of the following statements is true regarding constructors?

1. There can be any number of no arg constructors.
2. JVM provides a no arg constructor when the constructor is missing for a class.
3. A class can have only one copy constructor.
4. Each class has at least one parameterized constructor.

## Poll 7 (15 sec)

Which of the following statements is true regarding constructors?

1. There can be any number of no arg constructors.
2. **JVM provides a no arg constructor when the constructor is missing for a class.**
3. **A class can have only one copy constructor.**
4. Each class has at least one parameterized constructor.



You learnt how to provide constructors for the classes. Now, provide constructors for the Customer class and use them to instantiate customer objects.

## **TODO:**

- Use the following command to check out to the current state:  
git checkout 38275b0
- Provide the following constructors for the Customer class:
  - No Arg Constructor and then print a message stating “object created” through it.
  - Parameterized constructor and copy constructor

[Code Reference](#)

‘this’ keyword

- Earlier, you learnt how to provide constructors for a class. When you do not provide any constructor, the compiler will provide a default constructor.
- If you choose to provide one, you can either provide a no arg constructor or a parameterized constructor.
- One of the special types of the parameterized constructors is the copy constructor, which is used for cloning existing objects.
- Yet, there are two unanswered questions about constructors, which are as follows:
  - Can the constructor argument name be the same as the attribute name?
  - How to call one constructor from another?
- Let's answer them one by one.

- So, the first question is ‘Can you have a constructor argument name that is the same as the attribute name?’
- The answer is **yes**. How to initialise attributes for an object? How can you tell JVM or the compiler whether you are referring to attributes or constructor arguments?
- This is where a special Java keyword helps you—the **this** keyword.
- The ‘this’ refers to the current object in the context of which the methods or the constructor is called.
- You will understand this by learning how methods work in Java.

- You will learn how the activate() method works in the Product class.

```
void activate() {  
    active = true;  
}
```

- Suppose you call this method as shown below:

```
appleMacBookAir.activate()
```

- When the activate() method is called, it will start executing.
- In the activate() method, it will try to set the value of the active variable to true. However, there is no active variable inside the activate() method.

- When JVM is executing a method and it does not find it inside the current method, it will start searching for it in the attribute list of the current object in the context of which the method was called.
- How does JVM know which object is the current one? ***It is always the object before the dot.***

**appleMacBookAir**.activate()

- When you call a method on an object, the reference to that object (object before dot) is copied inside the 'this' keyword and passed to the method.
- Now, JVM has access to all the attributes of the object (before dot). You can use the 'this' keyword directly to access the attributes.

- You can use the 'this' keyword as shown below.

```
void activate() {  
    this.active = true;  
}
```

- Now, JVM will directly start searching for the active variable inside the attribute list of the **object before dot**.
- The same is true for the constructor call as well. The only difference is that as there is no **object before dot** for the constructor; in a constructor, **this** refers to the object that is newly getting created.
- Now, you will learn how to perform this in a constructor.

```
Product (int id, String name, String category,  
        float salesPrice, float cost, int quantity,  
        boolean active) {  
    this.id = id;  
    this.name = name;  
    this.category = category;  
    this.salesPrice = salesPrice;  
    this.cost = cost;  
    this.quantity = quantity;  
    this.active = active;  
}
```

[Code Reference](#)

You can use the 'this' keyword for other constructors and methods.



- Now, let's try to answer the second question—'How to call one constructor from another?'
- You will understand this through an example. Suppose you want to provide a no arg constructor that uses the parameterized constructor to provide some default values.
- You certainly cannot perform the following, as the Constructor can only be called during object creation (with a new keyword):

```
Product () {  
    Product (1, "product", "category",  
            0f, 0f, 0, false);  
}
```

- If you append the 'new' keyword before the Product, then you will end up creating two objects.
- In such a situation, the 'this' keyword is useful.
- You can use the 'this' keyword with parentheses to call another constructor with the matching signature.

```
Product() {  
    this(1, "product", "category",  
        0f, 0f, 0, false);  
}
```

- This is called **constructor chaining**. Also, this() should be the first statement in the constructor if you are performing constructor chaining.

## Poll 8 (15 sec)

Which of the following statements is true regarding the 'this' keyword?

(Note: More than one option may be correct.)

1. 'this' refers to the current object.
2. 'this' refers to the class of the current object.
3. 'this' can be used for constructor chaining.
4. 'this' cannot be used inside methods.

## Poll 8 (15 sec)

Which of the following statements is true regarding the 'this' keyword?

(Note: More than one option may be correct.)

1. **'this' refers to the current object.**
2. 'this' refers to the class of the current object.
3. **'this' can be used for constructor chaining.**
4. 'this' cannot be used inside methods.

All the codes that were used in today's session  
can be found at the link provided below:

[https://github.com/ishwar-soni/fsd-ooadp-ims/tree/session3  
-demo](https://github.com/ishwar-soni/fsd-ooadp-ims/tree/session3-demo)

# Important Concepts and Questions

1. What are the differences and similarities between methods and constructors?
2. How does JVM optimise String object creation?
3. What is 'null' and how is memory allocation performed for null objects?
4. What is the difference between object and instance?
5. What is the difference between pass by value and pass by reference?  
Can you give an example wherein pass by value throws an error and pass by reference rectifies the statement?
6. Can you pass objects as method arguments?
7. What are shallow cloning and deep cloning? How will you create the clone of a class that exists in a jar file and does not have source code access?

8. Can you give real-life examples in which deep cloning is required?
9. What is a memory leak in heap?
10. What is constructor overloading?
11. What is the difference between null and void?



# Doubt Clearance Window

# Today, you learnt about the following:

1. The concept of Abstraction and how is it performed in OOP
2. How to create classes and provide attributes and methods
3. How to create objects out of the classes
4. Difference between stack memory (for local variables) and heap memory (for objects)
5. The concepts of pass by value and pass by reference
6. The concept of String Pool and how JVM uses it to optimise String object creation
7. The concept of constructors along with their uses and types
8. The concept of constructor overloading
9. The 'this' keyword and its uses

**TODO:**

- Create a Player class with the following attributes and methods:
- Attributes:
  - ***playerName: String*** (to store the player name)
  - ***hitPoint: int*** (to store the player hit points)
  - ***alive: boolean*** (to store the current state of the player)
- Methods:
  - ***getHit(damage: int) : void*** (It will reduce the damage from the player hit point. If the player hit point goes below 0, you need to reset the hit point to 0 and make the player dead by setting alive attribute to false.)
  - ***isAlive() : boolean*** (It returns the value of alive.)

## TODO:

- Provide the following constructors for the Player class:
- ***Player(playerName : String, hitPoint: int)*** (Set the playerName and hitPoint attributes for the player that are equal to the values that are provided through the constructor arguments. Set the alive attribute to be true.)
- ***Player(playerName : String)*** (Use the constructor given above using the this() keyword to initialise the player with the given playerName and hitPoint as 100. Set the alive attributes to true.)

## Note:

1. Complete all the TODOs given in the stub code.
2. You should have the same name for the attributes, methods and arguments, as specified in the TODOs.

# Tasks to Complete After Today's Session

MCQs
Homework
Coding Questions
Project Checkpoint 2



Thank You!