upGrad

#LifeKoKaroLift

Object-Oriented Programming - Encapsulation

upGrad

Course: Object-Oriented Analysis, Design and Programming

Lecture On: Object-Oriented Programming - Encapsulation

Instructor: Sayan Nayak

upGrad

Topics covered in the previous class...

1. The concept of Abstraction and how is it done 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 and Heap memory
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 concepts of constructors, along with their uses and types, and constructor overloading

Poll 1 (15 sec)

Which of the following is a characteristic of the Heap memory?
(Note: More than one option may be correct.)

1. It is separate for each method.
2. It is common for the entire application.
3. It is used to store primitive data types.
4. It is used to store objects.

Poll 1 (15 sec)

Which of the following is a characteristic of the Heap memory?
(Note: More than one option may be correct.)

1. It is separate for each method.
2. **It is common for the entire application.**
3. It is used to store primitive data types.
4. **It is used to store objects.**

upGrad

Homework Discussion

upGrad

Today's Agenda

- **Provide Class Relationship and Encapsulate the Classes**
  - Class Relationships: Association, Aggregation and Composition
  - Static Attributes, Methods and Constructors
  - Introduction to Enum
  - Encapsulation: The second fundamental principle of OOP
  - Getters and Setters for the Attributes
  - Nested Classes

upGrad

Class Relationships

Class Relationships

upGrad

- While drawing the UML Class Diagram for the Inventory Management System, you learnt about the following four class relationships:
  - ***Association***: A relationship that cannot be of the other three types
  - ***Generalization***: A parent-child relationship; in the next session, we will discuss this in detail.
  - ***Aggregation***: When one object uses another object
  - ***Composition***: When one object owns another object

Class Relationships

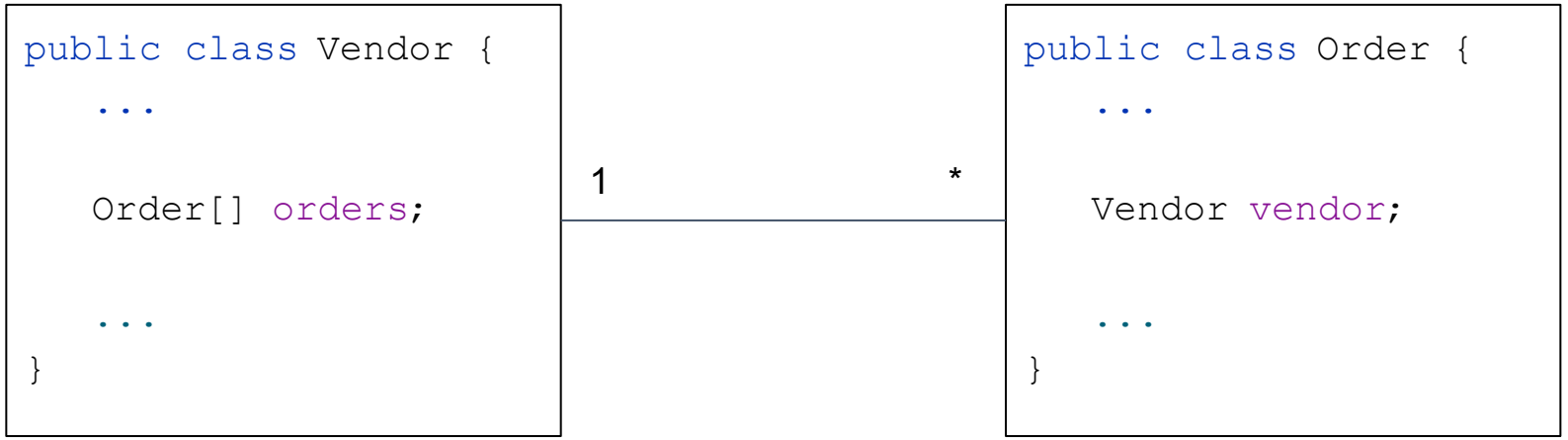
upGrad

- We also figured out how different classes will be related to each other.
  1. Order and Vendor: Association Relationship
  2. Product and Order: Aggregation Relationship
  3. Product and Vendor: Aggregation Relationship
  4. Customer and Address: Composition Relationship
  5. Customer and Contact: Composition Relationship
  6. Vendor and Address: Composition Relationship
  7. Vendor and Contact: Composition Relationship
- The last four relationships will change in the next session when we provide the BusinessPartner class. Instead of Customer and Vendor, Address and Contact will be in a relationship with the BusinessPartner class.

Association Relationship

upGrad

- You will learn how to provide an association relationship between the Vendor and Order classes.



Code Reference

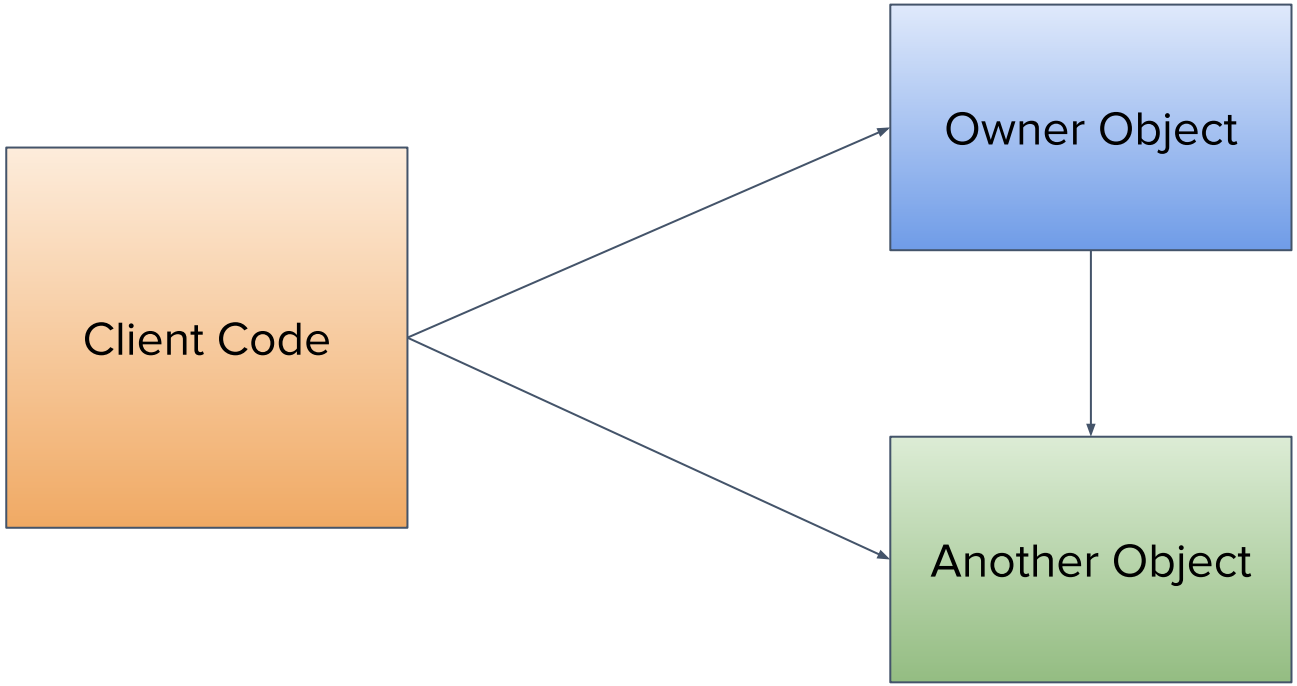
Aggregation and Composition Relationships

upGrad

- You know that when two objects are in an Aggregation relationship, they can exist independently.
- However, if the relationship is of the composition type, one object cannot exist if the owner of the relationship dies (does not exist).
- In an Aggregation relationship, the object will be created outside the owner object and will be provided to the owner object.
- However, in the Composition relationship, the object will be created inside the owner object.
- This is because if the object is created inside the owner object and the owner object is deleted, you will not be able to access the inner object either.

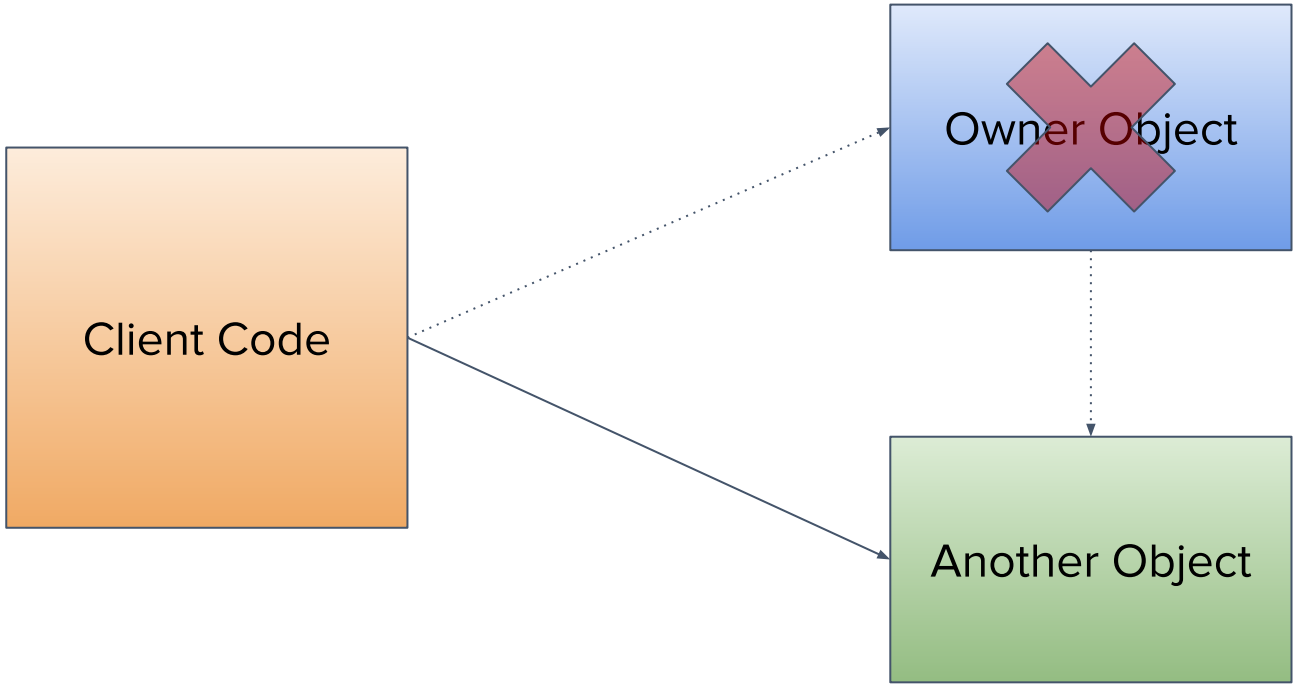
Aggregation Relationship

upGrad



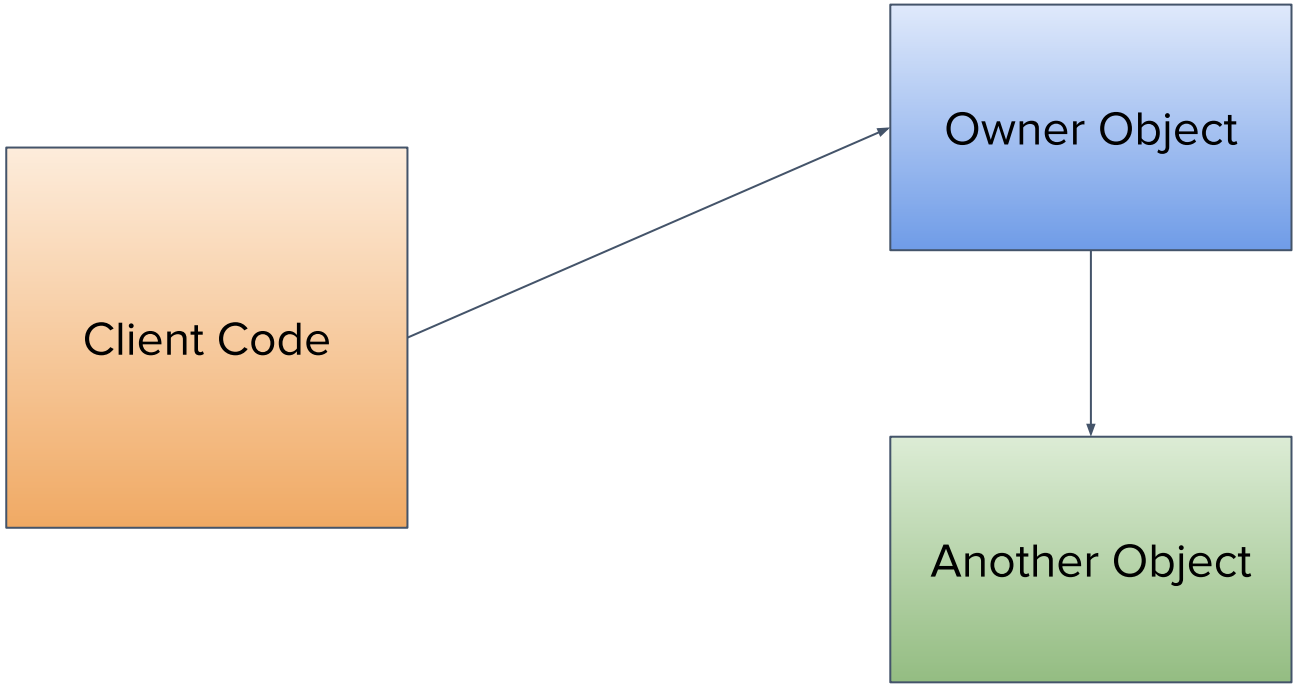
Aggregation Relationship

upGrad



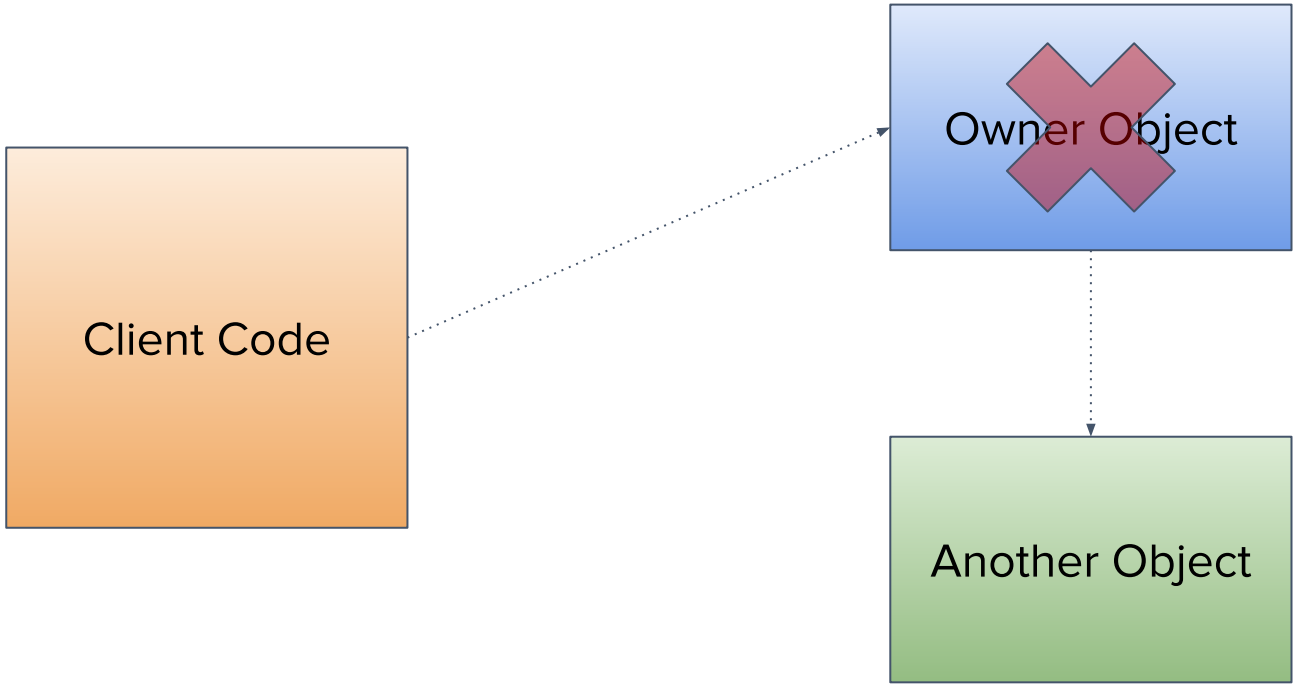
Composition Relationship

upGrad



Composition Relationship

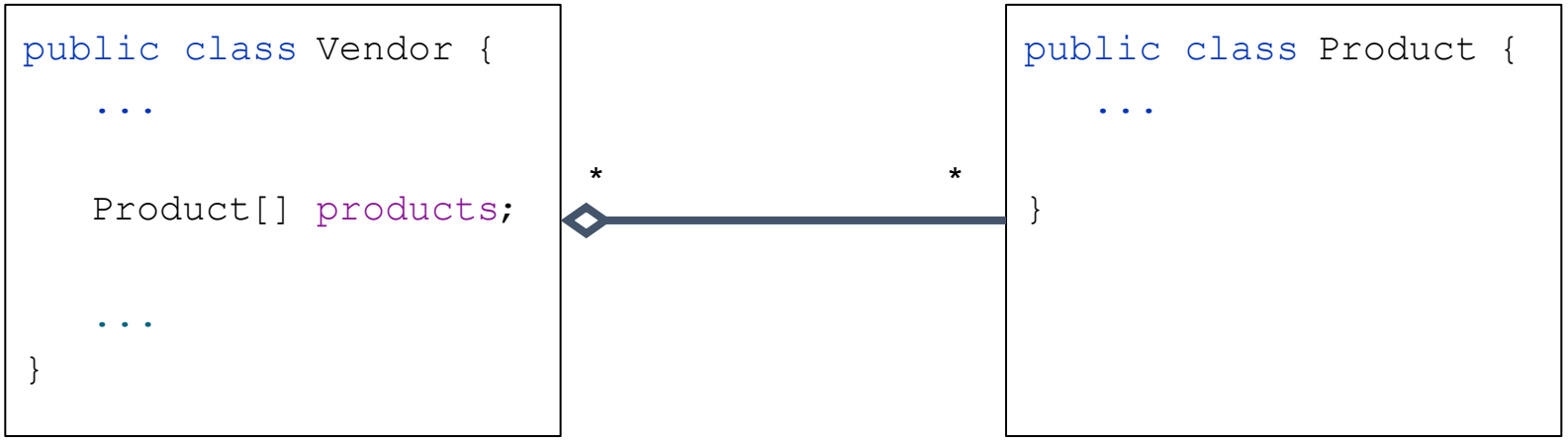
upGrad



Aggregation Relationship

upGrad

- Now, you will learn how to provide an aggregation relationship between the Vendor and Product classes.

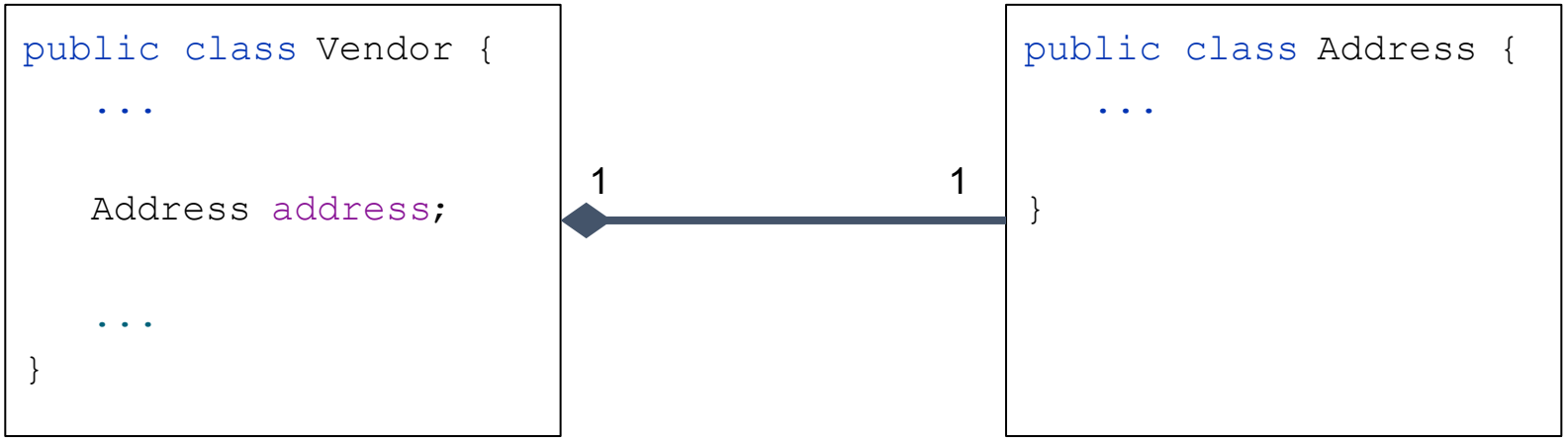


Code Reference

Composition Relationship

upGrad

- Let's provide a composition relationship between the Vendor and Address classes.



Code Reference

Poll 2 (15 sec)

Fill in the blank.

A Composition relationship is when _____.

1. One object uses another object
2. One object owns another object
3. One object is the child of another object
4. One object is related to another object

Poll 2 (15 sec)

Fill in the blank.

A Composition relationship is when _____.

1. One object uses another object
2. **One object owns another object**
3. One object is the child of another object
4. One object is related to another object

Poll 3 (15 sec)

How can you establish a Composition relationship between two objects?

1. By creating the object inside the owner object
2. By creating the object outside the owner object
3. By not creating the object at all
4. By creating the owner object inside the another object

Poll 3 (15 sec)

How can you establish a Composition relationship between two objects?

1. By creating the object inside the owner object
2. **By creating the object outside the owner object**
3. By not creating the object at all
4. By creating the owner object inside the another object

Hands-On Experience

upGrad

TODO:

- Use the following command to check out to the current state:
  git checkout a68fb57
- Use the IMS UML Class Diagram and provide the following relationship between the given classes:
  - Product and Order: Aggregation Relationship
  - Contact and Vendor: Composition Relationship
- Implement the updateVendorCredit() method of the Order class.
- Print the vendor credit after an order is placed.
- Print the vendor email id.

Code Reference

Static Members

Instance Members

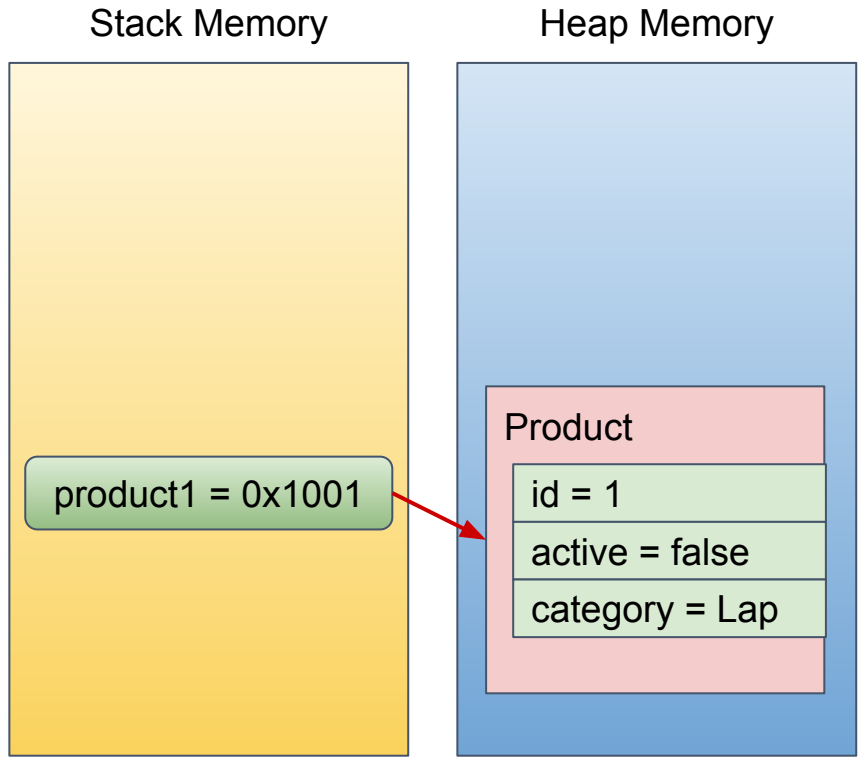
upGrad

- When we instantiate objects out of the classes, each instance or object has its own separate state.
- For each object, there is a separate copy of attributes. Changes made to the attributes of one object do not affect those of other objects.
- When you invoke methods on an object, that method can only access the attributes of that object for which it was invoked (the object before the dot).
- You will understand this through an example.

Instance Members

upGrad

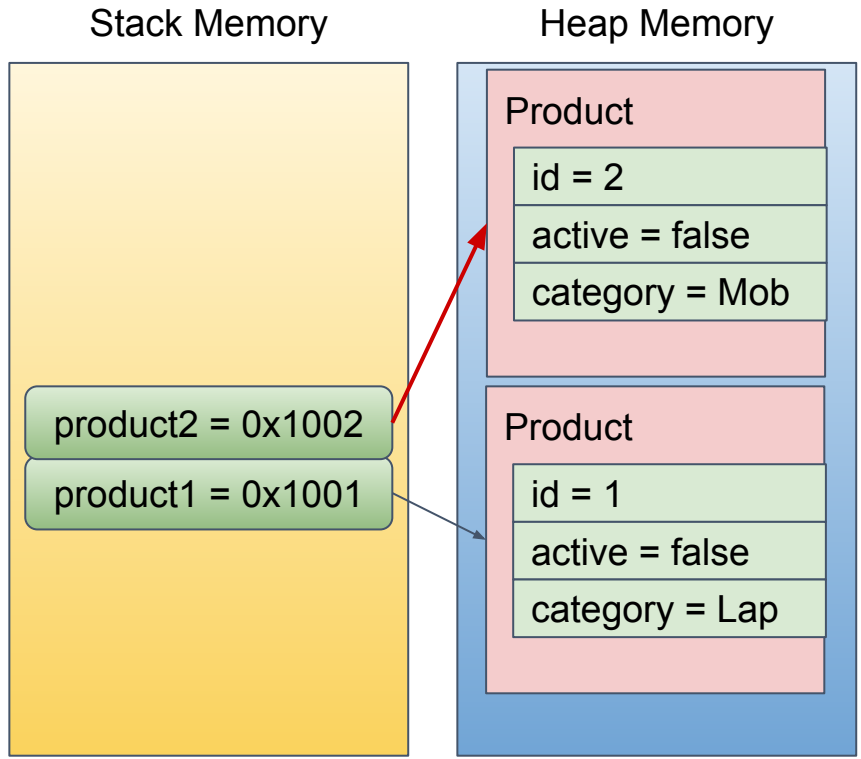
```
public static void main(String[] args) {
    Product product1 = new Product();
    Product product2 = new Product();
    product1.activate();
    System.out.println(product2.category);
}
```



Instance Members

upGrad

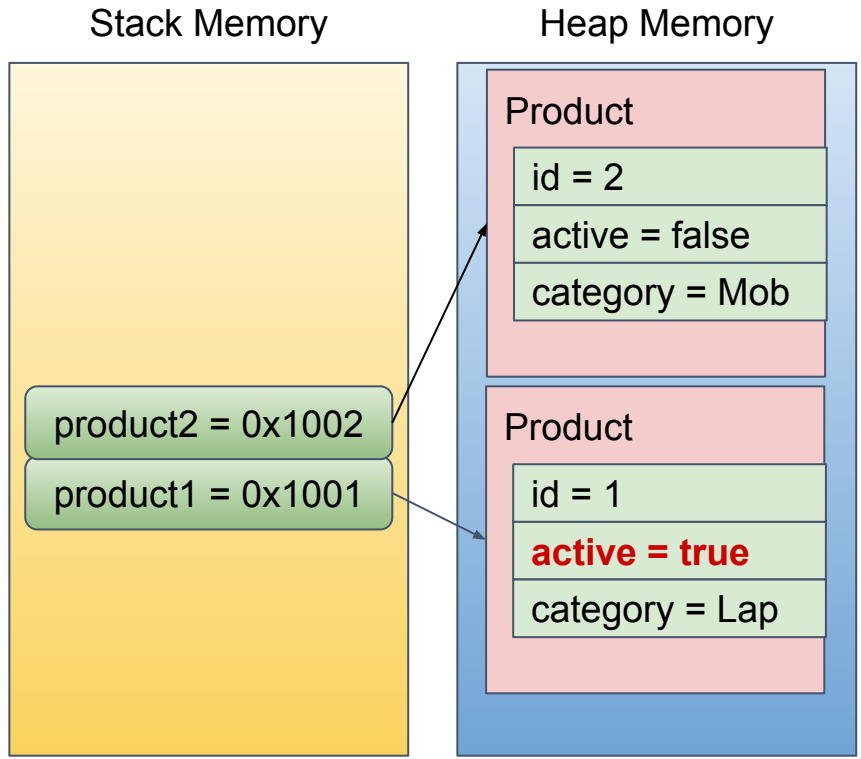
```
public static void main(String[] args) {
    Product product1 = new Product();
    Product product2 = new Product();
    product1.activate();
    System.out.println(product2.category);
}
```



Instance Members

upGrad

```
public static void main(String[] args) {
    Product product1 = new Product();
    Product product2 = new Product();
    product1.activate();
    System.out.println(product2.category);
}
```



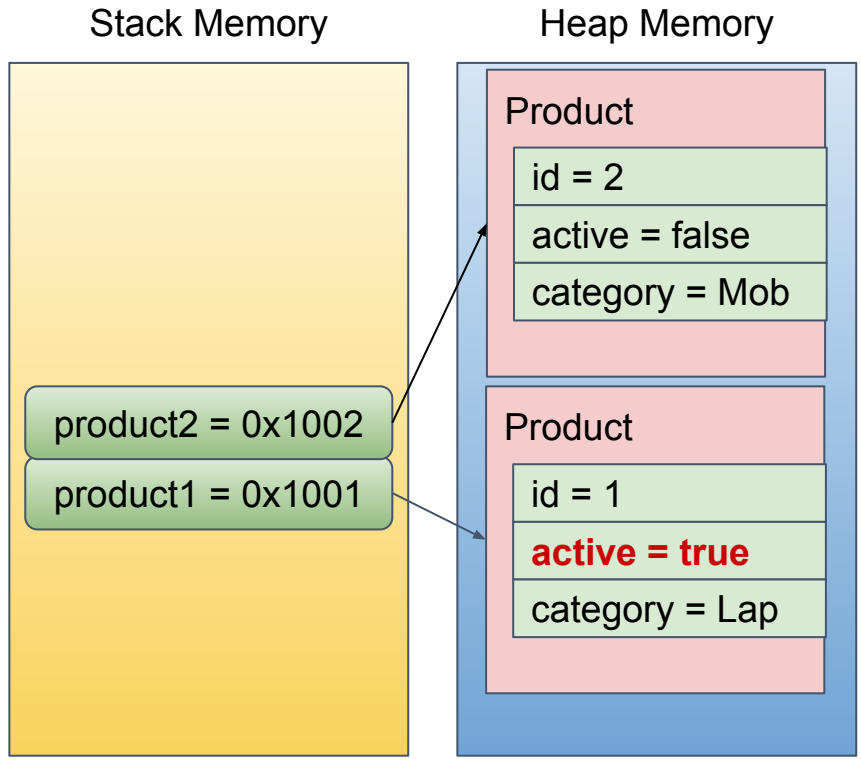
Instance Members

upGrad

```
public static void main(String[] args) {
    Product product1 = new Product();
    Product product2 = new Product();
    product1.activate();
    System.out.println(product2.category);
}
```

Console Output

Mob



Instance Members

upGrad

- As you can observe, each instance of the Product class has its own separate set of attributes.
- When you invoke a method on an instance, that method can only access attributes of that instance (object before dot).
- So, a separate copy of attributes is attached to each instance, and methods only work with that copy of attributes. Because of this nature, such attributes and methods are also called ***Instance Attributes*** and ***Instance Methods***.
- However, sometimes, you would also want attributes and methods that should be attached to a class rather than an instance. You want methods that need not access the instance attributes.

Static Members

upGrad

- You will understand this through an example.
- The String class provides a ***join()*** method, which joins the string array elements with the given delimiter.
- If the join() method was an instance method, you had to first create an instance of the String with a dummy value and then call the method.

```
String str = "dummy";
String[] names = {"Paul", "John", "David"};
String namesString = str.join(" => ", names);
System.out.println(namesString);
```

- **Output:** Paul => John => David

Static Members

upGrad

- Here, you will observe that the join() method of the String class does not need to access any instance attribute or method; thus, such a method can be attached to the Class rather that to the instances.
- Such methods are called ***Static Methods*** or ***Class Methods***. These methods can be accessed directly through the class, without the need to create an instance of that class.
- The join() method is also a static method of the String class.

Static Members

upGrad

```
String[] names = {"Paul", "John", "David"};
String namesString = String.join(" => ", names);
System.out.println(namesString);
```

- **Output:** Paul => John => David
- As you can observe, the join() method can be directly invoked using the class name, without the need to create an instance first as shown below.
  ClassName.methodName()

Static Members

upGrad

- You can also have static attributes. They will not be attached to each instance of that class but will be attached to the class itself and can be accessed using the class name.
- Static attributes are also called ***Class Attributes***.
- If you want to make an attribute or a method static, you only need to add ‘static’ keyword while declaring the attribute or the method.
- Once declared, you can access the static attributes and methods without first instantiating that class.

Static Members

upGrad

- Declaring static attributes and methods

```
static int attr = 10;

static void method() {

}
```

- Accessing static attributes and methods

```
int i = ClassName.attr;
ClassName.method();
```

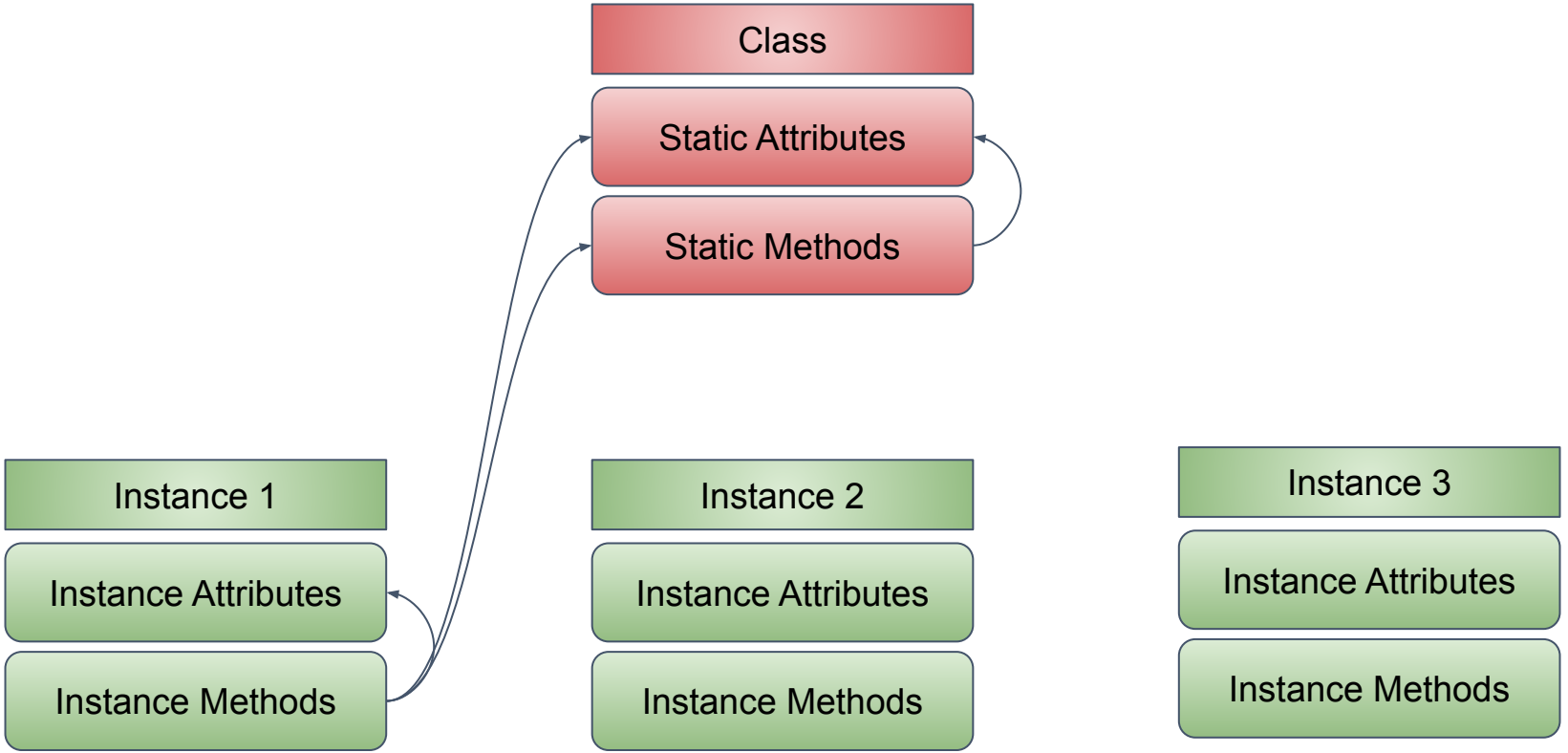
Static Members

upGrad

- Static Methods ***CANNOT*** access the instance attributes and methods but instance methods ***CAN***.
- What is the reason for this? This is because there can be multiple instances of the same class (multiple copies of the attributes), and static methods will not know the copy to which it has access, as there is no object before the dot, but there is a class name before the dot.
- However, there will be only one set of static attributes; hence, instance methods will know the attributes to which they have access.

Static Members

upGrad



Static Members

upGrad

	Instance Attributes and Methods	Static Attributes and Methods
Instance Methods can access	YES	YES
Static Methods can access	NO	YES

Static Members

upGrad

- Let’s provide an id counter attribute inside the Product class that will count the number of products that have been instantiated and help you assign a unique id to each product instance based on the number of products.

Code Reference

Static Constructor

upGrad

- How can you initialise static attributes? You cannot initialise them inside the constructor because otherwise, the static variable will be reset every time you instantiate the class.
- This is why Java provides a special ***static initializer*** or ***static constructor*** to initialise static attributes.
- A static initializer looks like the following:

```
static {
    //code to initialize static attributes
}
```

Static Constructor

upGrad

- There can be any number of static initializer blocks, and Java will call them in the same order in which they were provided in the class.
- Static blocks are executed when the class is loaded. So, the code written inside the static block will get executed prior to any other code in that class.
- Let’s provide a static initializer block for the product class to initialise the idCounter attribute.

Code Reference

Initializer Block

upGrad

- Similar to the static initializer block, there is also the initializer block. It is the same as the initializer block but without the static keyword.

```
{
    System.out.println("Inside initializer block");
}
```

- Initializer blocks are rarely used. The code that you write inside the initializer block gets copied to each constructor for that class.
- Also, similar to static initializer blocks, you can have any number of initializer blocks. Java will copy them in each constructor in the same order in which they were provided.

Initializer Block

upGrad

```
public class User {
    {
        System.out.println("Inside initializer block: 1");
    }

    {
        System.out.println("Inside initializer block: 2");
    }

    User() {
        System.out.println("Inside Constructor");
    }
}
```

Output: (When you instantiate the User class)

Inside initializer block: 1
Inside initializer block: 2
Inside Constructor

Poll 4 (15 sec)

Suppose a class contains a static initializer block and the main() method. Which one will be executed first?

1. main() method
2. Static initializer block
3. They are called randomly.
4. It depends on other factors.

Poll 4 (15 sec)

Suppose a class contains a static initializer block and the main() method. Which one will be executed first?

1. main() method
2. **Static initializer block**
3. They are called randomly.
4. It depends on other factors.

Poll 5 (15 sec)

Fill in the blank.

Static methods can access the _____ of the same class? (Note: More than one option may be correct.)

1. Instance Attributes
2. Instance Methods
3. Static Attributes
4. Static Methods

Poll 5 (15 sec)

Fill in the blank.

Static methods can access the _____ of the same class? (Note: More than one option may be correct.)

1. Instance Attributes
2. Instance Methods
3. **Static Attributes**
4. **Static Methods**

Hands-On Experience

upGrad

TODO:

- Use the following command to check out to the current state:
  git checkout e2d133e
- Provide the idCounter attribute inside the Vendor class. Make it of the types int and static.
- Provide a static initializer block inside the Vendor class to initialise the idCounter attribute to 0.
- Use this idCounter attribute to assign unique increasing ids to vendor objects.
- Provide the countVendors() method inside the Vendor class. This method should return the number of the currently available vendors and should be static.

Code Reference

Enum

Enum

upGrad

- Sometimes, you want that the value for a variable or an attribute should be from a predefined set of constants.
- For example, suppose you have a gender attribute in the class. You know that the value for this attribute can be either male, female or other.
- Now, this attribute can be declared in many ways; for example, you can make it int and assign different values such as 0, 1 or 2 based on whether the gender is male, female or other, respectively.
- Alternatively, you can declare this attribute as String and assign values such as ‘male’, ‘female’ or ‘other’.

Enum

upGrad

- However, in these approaches, there are high chances of making typographical errors.
- This is where enum helps you.
- ***Enum is a special data type. A variable of the enum type must be equal to one of the values that have been predefined for it.***
- Let's provide the Gender enum.

```
public enum Gender {
    MALE, FEMALE, OTHER
}
```

- As all the members inside the enum are constants, they are in uppercase letters. For multiple words, we separate them using underscore.

Enum

upGrad

- You will learn how to use Enum.

```
public class User {
    String name;
    Gender gender;

    User(String name, Gender gender) {
        this.name = name;
        this.gender = gender;
    }

    String getDetails() {
        return "Name: " + this.name + ", Gender: " + this.gender.name();
    }
}
```

```
public static void main(String[] args) {

    User user = new User("John", Gender.MALE);
    System.out.println(user.getDetails());
}
```

Output:

Name: John, Gender: MALE

Poll 6 (15 sec)

The following Direction enum has been provided. Which of the following is the correct way to create a variable of the Direction type?

enum Direction {NORTH, SOUTH, EAST, WEST}

1. Direction direction = 0;
2. Direction direction = “NORTH”;
3. Direction direction = Direction.NORTH;
4. Direction direction = NORTH;

Poll 6 (15 sec)

The following Direction enum has been provided. Which of the following is the correct way to create a variable of the Direction type?

enum Direction {NORTH, SOUTH, EAST, WEST}

1. Direction direction = 0;
2. Direction direction = “NORTH”;
3. **Direction direction = Direction.NORTH;**
4. Direction direction = NORTH;

upGrad

Encapsulation: The Second Fundamental Principle of OOP

Encapsulation

upGrad

- ***Encapsulation refers to hiding the internal complex details and providing a simple interface to work with objects.***
- For example, suppose you are writing a class for the stack data structure for integer elements.
- How will you interact with the stack? You can interact with the stack by the following two methods:
  - push(ele: int): void
  - pop(): int
- The push() method inserts an element into the stack, whereas the pop() method gets the elements in the LIFO manner.

Encapsulation

upGrad

- Will there be only two methods inside the Stack class?
- No, there can be many other methods or attributes to make the Stack work.
- For example, if you are implementing the stack using a linked list, then the following are some of the attributes and methods that can also be part of the Stack class:
  - list: internal data structure to store elements
  - head: pointer to the head of the stack
  - createNewNode(): to create a new node while pushing elements
  - append(): to insert a new node and update the head pointer
  - isEmpty(): to check whether the stack is empty

Encapsulation

upGrad

- So, a class can contain many attributes and methods.
- However, to a client code, only those attributes and methods should be visible that are required to interact with the object. All other attributes and methods that are required for the internal working of an object should be hidden from outside.
- Why hide the internal details? This makes the code clean and maintainable. Also, the internal details can be prone to changes, but the interface should never change.

Encapsulation

upGrad

- For example, you can the stack implementation from the linked list to an array.
- Many or all the internal attributes and methods may change, but you will not change the two interface methods–push() and pop().
- How will this help you? If only the interface methods are visible to the client code, no other class will be referring to the internal attributes and methods.
- So, any changes made inside the Stack class will not force you to make changes at other places. You can easily change the internal structure of the class without worrying about your code breaking at any other place. Hence, the code will be clean and maintainable.

Abstraction vs Encapsulation

upGrad

Abstraction	Encapsulation
Ignoring the unnecessary details	Hiding the unnecessary (internal or implementation) details
Focuses on the observable behaviour of an object	Focuses on the implementation that gives rise to this behaviour

Abstraction and Encapsulation are complementary because for abstraction (classes and objects) to work, implementation must be encapsulated.

Access Modifiers

upGrad

- You know that you should hide the internal details of an object. Internal details refer to those attributes and methods that are not part of the interface for that object.
- How can you hide attributes and methods? You can hide them using ***Access Modifiers***.
- Java provides three access modifiers: ***public, private*** and ***protected***.
- If you do not specify an access modifier for a class member, then that member is given default access or, in Java terms, the ***package-private*** access.
- Now, you will learn how class members with different access modifiers can be accessed.

Access Modifiers

upGrad

	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
package-private	Yes	Yes	No	No
private	Yes	No	No	No

Access Modifiers

upGrad

- Currently, we have made our classes public, but their attributes, methods and constructors has no access modifier specified for it.
- Thus, our classes are public, but the class members are *package-private*.
- Create a new package, provide a class in that package and try the following:
  - Create an attribute or variable of the Product class. You will be able to do this because the Product class is public.
  - Now, try to create an instance of the Product class. The compiler will throw an error because none of the Product constructors are public.
  - Make a constructor public, and you will be able to instantiate the Product class but will not be able to access its member because attributes and methods of Product class are not public.

Access Modifiers with Attributes

upGrad

- The syntax to declare attributes with an access modifier is as follows:
  [accessModifier] [static] type attributeName = value;

- Example:

```
public static int idCounter;
private int id;
```

Access Modifiers with Methods

upGrad

- The syntax to declare methods with the access modifier is as follows:

[accessModifier] [static] returnType methodName (parameterList) {
//code
}

- Example:

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

```
public static int countProducts() {
    return idCounter;
}
```

Poll 7 (15 sec)

What is the main use of Encapsulation?

1. It makes your code more efficient.
2. It makes your code more maintainable.
3. You need to write less code.
4. It makes your code easy to compile.

Poll 7 (15 sec)

What is the main use of Encapsulation?

1. It makes your code more efficient.
2. **It makes your code more maintainable.**
3. You need to write less code.
4. It makes your code easy to compile.

Poll 8 (15 sec)

To which of the following are private attributes of a class accessible? (Note: More than one option may be correct.)

1. Private methods of the same class
2. Public methods of the same class
3. Private methods of another class
4. Public methods of another class

Poll 8 (15 sec)

To which of the following are private attributes of a class accessible? (Note: More than one option may be correct.)

1. **Private methods of the same class**
2. **Public methods of the same class**
3. Private methods of another class
4. Public methods of another class

Getters and Setters

Getters and Setters

upGrad

- So far, you have learnt that encapsulation is used to hide the implementation details so that your code remains more maintainable than the non-encapsulated code.
- One of the coding principles that is related to encapsulation is data hiding.
- Data hiding means that you should make your attributes private and then provide an interface (public methods) to work with those attributes.
- You should do this to prevent the client code from assigning (setting) an invalid value to an attribute, for example, making id or idCounter variables negative.

Getters and Setters

upGrad

- Also, data hiding makes your code more maintainable, as it is easier to track whether a method was called or not (by putting a log statement) than to track whether an attribute was accessed.
- The public methods that are used to get the value of the private attributes are called ***getters***.
- On the other hand, the public methods that are used to set the value of the private attributes are called ***setters***.

Getters and Setters Naming Convention

upGrad

- The conventions to be followed while declaring getters and setters for an attribute are as follows:
  - They are named after the attribute in lowerCamelCase.
  - The getter methods start with '***get***', and the setter methods start with '***set***', except the getter method for the Boolean attribute that starts with '***is***'.
  - The return type of the getter is the same as the attribute type, and its parameter list is empty.
  - The return type of setter is void, and it accepts only the parameter, which is of the same type as that of the attribute.
- Let's encapsulate the Product class and provide getters and setters for the attributes.

Getters and Setters

upGrad

```
public class Product {
    ...
    private float salesPrice;

    public float getSalesPrice() {
        return salesPrice;
    }

    public void setSalesPrice(float salesPrice) {
        this.salesPrice = Math.max(salesPrice, 0.0f);
    }
    ...
}
```

Getters and Setters

upGrad

```
public class Product {
    ...
    private boolean active;

    public boolean isActive() {
        return active;
    }

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

    public void deactivate() {
        this.active = false;
    }
    ...
}
```

Code Reference

Private Constructor

upGrad

- Earlier, you learnt that private methods (both instance and static) are used to hide internal implementation details that are prone to change.
- If these methods are hidden, then other pieces of code will not be dependent on these methods. Thus, you can make changes to these methods without breaking your code.
- This improves the maintainability of your code.
- You can also make constructors private, not only methods. If a constructor is private, then that class can only be instantiated inside the same class.
- What is the use of such classes?

Private Constructor

upGrad

- Such classes are used to implement the ***Singleton pattern***. In the Singleton pattern, there will exist only one instance of a class–the ***Singleton class***.
- A private constructor is also used for utility classes that consist of only static methods. If there are only static methods in a class, you do not need an instance to be created for that class. Thus, you should make the constructor private.
- Now, you will learn how the Singleton pattern is implemented.

Singleton Pattern

upGrad

```
public class Admin {
    private String username;
    private String password;

    private static Admin instance;

    public static Admin getInstance() {
        if (instance == null) {
            instance = new Admin();
        }
        return instance;
    }

    private Admin() {
        //get username and password from db
        this.username = "admin";
        this.password = "password";
    }
}
```

Utility Class

upGrad

```
public class UtilClass {
    public static void method1() {
        //code
    }

    public static void method2() {
        //code
    }

    private UtilClass(){}
}
```

Immutable Objects

upGrad

- If you make attributes private and then do not provide any setter method for that class, then, once initialised, the attributes cannot be changed. Such objects are called immutable objects.
- ***An object is considered to be immutable if its state cannot change after it is constructed.***
- The most famous example for immutable objects is the String class that keeps all its attributes private and does not provide any setter method.
- However, only making attributes private and not providing any setter method is not enough for immutability; the attributes also need to be made constant. You will gain a deeper understanding of it while learning about the ***final*** keyword.

Poll 9 (15 sec)

Which of the following is the correct way to provide a setter for the given attribute?

```
private String previousCompany;
```

1. public String setPreviousCompany() {...}
2. public void setPreviousCompany(String previousCompany) {...}
3. public void setPreviousCompany() {...}
4. public String setPreviousCompany(String previousCompany) {...}

Poll 9 (15 sec)

Which of the following is the correct way to provide a setter for the given attribute?

```
private String previousCompany;
```

1. public String setPreviousCompany() {...}
2. **public void setPreviousCompany(String previousCompany) {...}**
3. public void setPreviousCompany() {...}
4. public String setPreviousCompany(String previousCompany) {...}

Hands-On Experience

upGrad

TODO:

- Use the following command to check out to the current state:
  git checkout 617dc53
- Encapsulate the Order class with data hiding (make attributes private and provide public getters and setters)
- Provide getters and setters for the following attributes:
  - id (only getter)
  - vendor, date, orderedProduct, orderedQuantity, amountPaid
- Make updateVendorCredit() a private method.
- The updateVendorCredit() method should be called whenever you change the vendor, orderedProduct, orderQuantity and amountPaid.

Code Reference

upGrad

Nested Classes

Nested Classes

upGrad

- So far, we have provided only one class per Java file, and the name of the class was the same as the file name (without .java suffix).
- However, we can provide multiple classes in the same Java file, with only the following two restrictions:
  - There can be maximum one public class in a Java file.
  - If there is a public class in a Java file, then the name of the public class should be the same as the file name (without .java suffix).
- You will understand this through an example.

Nested Classes

upGrad

- For a Java file named MyClass.java, the following will not throw an error.

```
public class MyClass {
}

class MyClass1 {
}

class MyClass2 {
}

class MyClass3 {
}
```

Nested Classes

upGrad

- For a Java file named MyClass.java, the following will also not throw an error.

```
class MyClass1 {
}

class MyClass2 {
}

class MyClass3 {
}
```

Nested Classes

upGrad

- For a Java file named MyClass.java, the following will throw an error.

```
public class MyClass1 {
}

class MyClass2 {
}

class MyClass3 {
}
```

Nested Classes

upGrad

- We can also declare classes inside another class, and such classes are called ***Nested Classes***.
- Nested classes are usually used to declare those classes that are only used in one place to make code more maintainable code and increase encapsulation.
- For example, when you provide the BusinessPartner class, then the Address and Contact classes will only be used with the BusinessPartner class. Thus, you can make both these classes nested classes to the BusinessPartner class.

Nested Classes

upGrad

- In Java, nested classes are of the following four types:
  - Static nested classes
  - Non-static nested classes or inner classes
  - Local classes
  - Anonymous classes (this will be covered when you learn about inheritance)

Nested Classes

upGrad

```
public class OuterClass {
    static class StaticNestedClass {
        //code here
    }

    class InnerClass {
        //code here
    }

    void method () {
        class LocalClass {
            //code here
        }
    }
}
```

Nested Classes

upGrad

Static Nested Class	Inner Class	Local Class
Similar to static methods; attached to the enclosing class and not to an instance	Similar to instance methods; attached to an instance of the enclosing class	Similar to local variables; they are declared inside a method
Similar to static methods; can be made public, private, protected or package-private	Similar to instance methods; can be made public, private, protected or package-private	Similar to local variables; they cannot have an access modifier in their declaration
Similar to static methods; can only access static members of the enclosing class	Similar to instance methods; can access both static and instance members of the enclosing class	Can access both static and instance members of the enclosing class
Can provide both static and non-static members	Can only provide non-static members	Can only provide non-static members

Nested Classes

upGrad

Static Nested Class	Inner Class	Local Class

Can be instantiated outside the class as shown below
 	
`OuterClass.StaticNestedClass obj = new OuterClass.StaticNestedClass();`
	Can be instantiated outside the class as shown below
	
	`OuterClass outerObj = new OuterClass();`
	
	`OuterClass.InnerClass obj = outerObj.new InnerClass();`
		Similar to local variables; cannot be accessed outside the method

Nested Classes

upGrad

- Let’s make an Address class that is nested to the Vendor class.
- As you do not want to access the Address class outside the vendor class, you can make it private.

Code Reference

upGrad

Poll 10 (15 sec)

Which of the following classes cannot be made private?

1. Outer class
2. Static Nested Class
3. Inner Class
4. Local Class

Poll 10 (15 sec)

Which of the following classes cannot be made private?

1. Outer class
2. Static Nested Class
3. Inner Class
4. **Local Class**

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

https://github.com/ishwar-soni/fsd-ooadp-ims/tree/session4-demo

upGrad

Important Concepts and Questions

Important Questions

upGrad

1. Can we instantiate static nested classes? Which syntax is used to do so?
2. What is use of the static attributes and variables? Can you make local variables static?
3. What are enum data types? Can you instantiate enum types?
4. Out of a static initializer block or main() method, which will get executed first?
5. What is a static constructor? What would happen when you remove static from the static constructor?
6. How are Abstraction and Encapsulation complementary?
7. Suppose a class is declared without any access modifiers. Where may the class be accessed?

Important Questions

upGrad

8. What are some modifiers that may be used with nested classes?
9. What you understand by Encapsulation?
10. What is the use of a private constructor?
11. What are immutable objects? Can you give real-life examples of immutable objects?

upGrad

Doubt Clearance Window

upGrad

Today, you learnt about the following:

1. Different types of class relationships and how to represent them using code
2. Static members of class and how to declare and access them
3. The concepts of static constructors (static initializer block) and initializer block
4. Enum data types and how to use them
5. The concepts of encapsulation and data hiding
6. The concepts of access modifiers or access specifiers
7. How to provide getters and setters
8. The use of private constructor and how to instantiate such classes
9. Different types of nested classes and how to use them

TODO:

- Create a Player class with the following attributes and methods:
- Attributes:
  - ***- maxHealth: int*** (make it static)
  - ***- name: String***
  - ***- currentHealth: int***
- Here, '-' is the UML symbol for the private access modifier.
- Provide a static initializer block to initialise the maxHealth attribute to 100.
- Provide a public constructor for the Player class with one argument of the type String. This argument should be used to initialise the name attribute of the Player. Initialise the currentHealth attribute to maxHealth.
- Provide the getter and setter methods for the name and currentHealth attributes.

TODO:

- In the Game class, declare two attributes of the type Player and name them player1 and player2.
- These attributes will be initialised in the Game constructor and will print the respective message when the game starts.

upGrad

Tasks to Complete After Today's Session

MCQs
Homework
Coding Questions
Project Checkpoint 3

upGrad

#RahoAmbitious

Thank You!