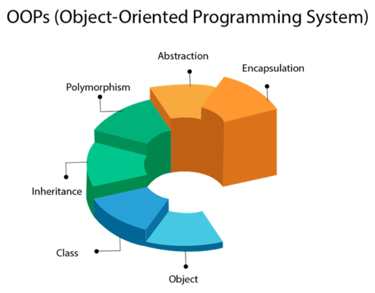
HOMEWORK1
FATMA BETÜL UYAR

1 – Why we need to use OOP ? Some major OOP languages ?



OOP programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming.

- We need to use OOP because it provides advantages such as **modularity**, **reusability through inheritance, flexibility through polymorphism**. What is the meaning of them?
- A module is more like an **independent partition of software** that is communicated through an interface.
- With modularity, the partitioning of the software environment in different modules makes the entire process **more optimized** and **reduces the coupling effect**.
- When working with OOP, you know exactly to look when something goes wrong. Objects are **self-contained**.
- Reusability can help developers save time **using the same code** while creating a newer version of the software or app.
- OOP makes **development** and **maintenance easier**, whereas, in other languages, it is not **easy to manage** if code grows as project size increases.
- It provides **data hiding** so that global data cannot be accessed from anywhere.

Some major OOP languages are **Java**, **Phyton**, **C++, C#**, **Ruby**

2 – Interface vs Abstract class ?

Interface	**Abstract Class**
only **abstract methods** Since Java 8, it can have **default and static methods** also.	**abstract** and **non-abstract methods**
can't provide the implementation of an abstract class.	can provide the implementation of the interface.
by default **final**	may contain **non-final**
only **static** and **final** variables	can have **final**, **non-final**,**static**,**non-static** variables
can be implemented using the keyword **'implements'**	can be extended using the keyword **'extends'**
an interface can **extend another java interface only**	an abstract class can **extend another java class** and **implement multiple java interfaces.**
members of a java interface are **public default**	can have class members like **private**, **protected**, etc

3 – Why we need equals and hashcode ? When to override ?

- In java **equals() method** is used to compare equality of two Objects.
- The **HashCode() method** returns the hashcode value as an Integer. Hashcode value is mostly used in hashing based collections like HashMap.
- This method must be overridden in every class which overrides equals() method. During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value, provided no information used in **equals(Object)** comparison on the Object is modified.
- If two Objects are equal, according to the **equals(Object)** method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the **equals(Object)** method, It is not necessary that the Integer value produced by hashCode() on each of the two Objects will be distinct.

We can override equals() method when it doesn't consider only object identity.

```java
class Money {
    int amount;
    String currencyCode;
}
```

```java
Money income = new Money(55, "USD");
Money expenses = new Money(55, "USD");
boolean balanced = income.equals(expenses)
```

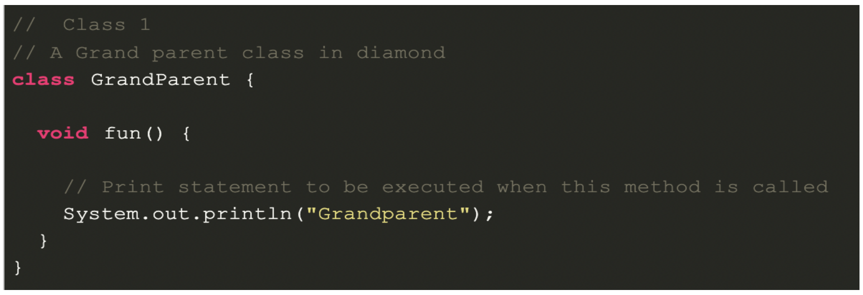
We would expect *income.equals(expenses)* to return *true*. But with the *Money* class in its current form, it won't.

The default implementation of *equals()* in the class *Object* says that equality is the same as object identity. And *income* and *expenses* are two distinct instances.

```java
@Override
public boolean equals(Object o) {
    if (o == this)
        return true;
    if (!(o instanceof Money))
        return false;
    Money other = (Money)o;
    boolean currencyCodeEquals = (this.currencyCode == null &&
other.currencyCode == null)
      || (this.currencyCode != null &&
this.currencyCode.equals(other.currencyCode));
    return this.amount == other.amount && currencyCodeEquals;
}
```

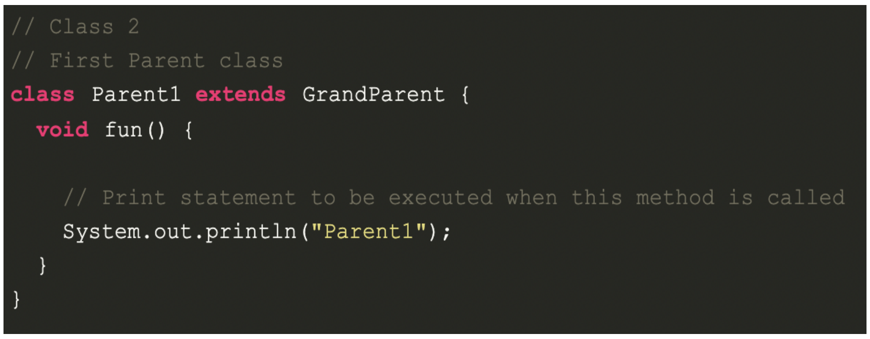
4 – Diamond problem in Java ? How to fix it?

- What Java doesn't allow is multiple inheritance where one class can inherit properties from more than one class. It is known as the Diamond Problem.



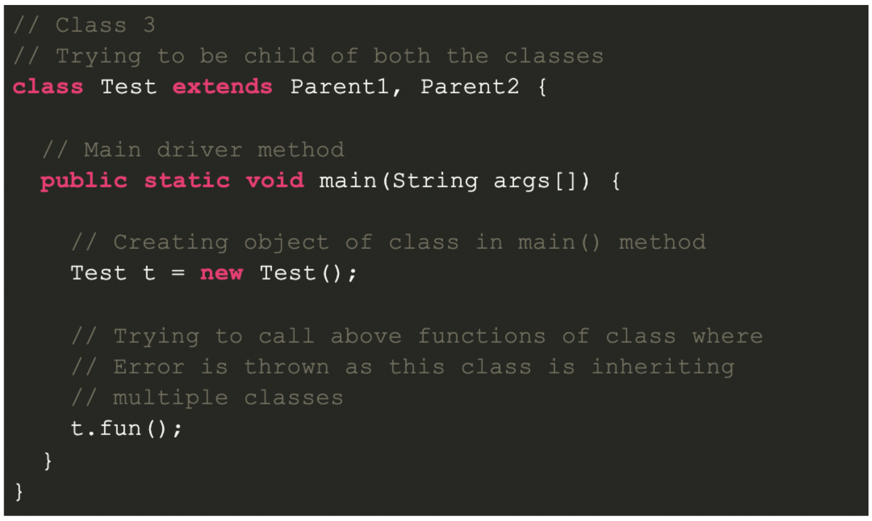
```
//  Class 1
// A Grand parent class in diamond
class GrandParent {

  void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Grandparent");
  }
}
```

Parent Class



```
// Class 2
// First Parent class
class Parent1 extends GrandParent {
  void fun() {

    // Print statement to be executed when this method is called
    System.out.println("Parent1");
  }
}
```

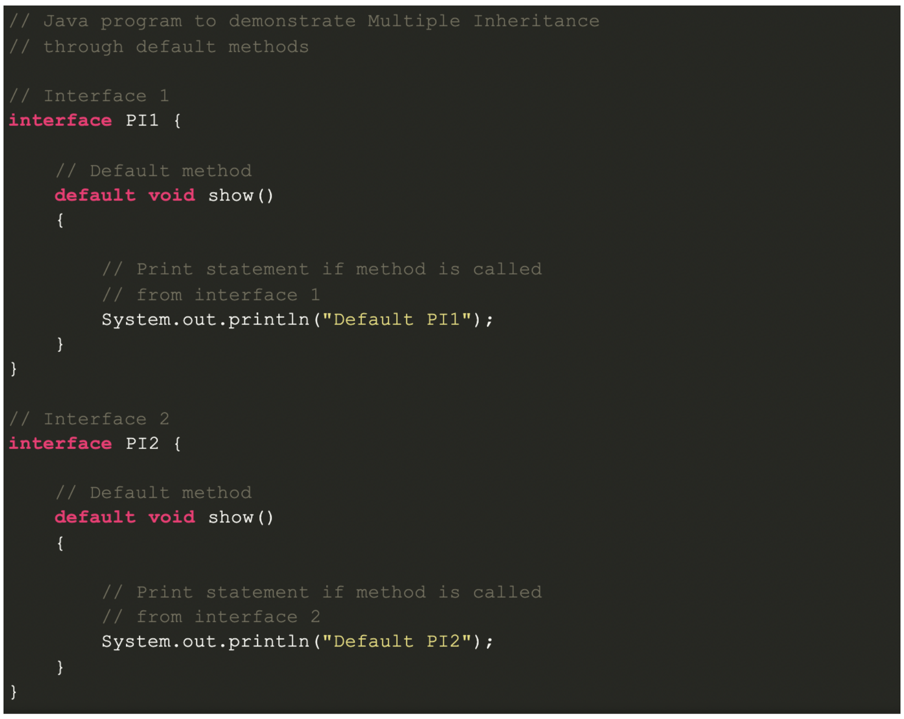
A class can inherit another class like this example.



```
// Class 3
// Trying to be child of both the classes
class Test extends Parent1, Parent2 {

  // Main driver method
  public static void main(String args[]) {

    // Creating object of class in main() method
    Test t = new Test();

    // Trying to call above functions of class where
    // Error is thrown as this class is inheriting
    // multiple classes
    t.fun();
  }
}
```

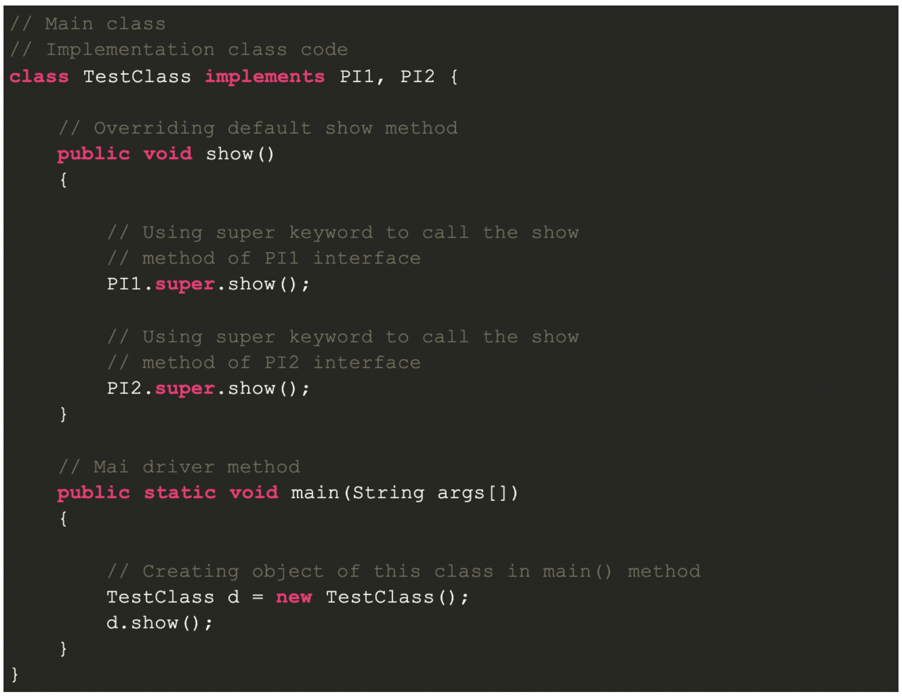
However, a class cannot inherit more than one class

- How to fix it?



```java
// Java program to demonstrate Multiple Inheritance
// through default methods

// Interface 1
interface PI1 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 1
        System.out.println("Default PI1");
    }
}

// Interface 2
interface PI2 {

    // Default method
    default void show()
    {

        // Print statement if method is called
        // from interface 2
        System.out.println("Default PI2");
    }
}
```

The solution to the diamond problem is **default methods** and **interfaces**.



```java
// Main class
// Implementation class code
class TestClass implements PI1, PI2 {

    // Overriding default show method
    public void show()
    {

        // Using super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // Using super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    // Mai driver method
    public static void main(String args[])
    {

        // Creating object of this class in main() method
        TestClass d = new TestClass();
        d.show();
    }
}
```

We can achieve multiple inheritance by using these two things.

5 – Why we need Garbage Collector ? How does it run ?

- Garbage collection in Java is the process by which Java programs perform **automatic memory management**.
- When java programs run on the JVM,objects are created on the heap, which is a portion of memory dedicated to the program.Eventually, some objects will no longer be needed. The Garbage collector **finds these unused objects** and **deletes** them to free up memory.
- We need a Garbage Collector because It makes java **memory efficient.**
- It is **automatically done** by the garbage collector so we don't need to make extra efforts.
-We can never predict when the garbage collector will run.
- We can try to explicitly call the garbage collector by **System.gc()** and **Runtime.gc()**

6 – Java 'static' keyword usage ?

The **static keyword** in Java is mainly used for memory management.

- We can access *static* fields without object initialization.
- **Static variables** are stored in the **heap.**
- *static* methods in Java are resolved at compile time. Since method overriding is part of Runtime Polymorphism, **static methods can't be overridden.**
- Abstract methods can't be static.
- *static* methods can't use *this* or *super* keywords.
- A class can have multiple *static* blocks.
- *static* fields and *static* blocks are resolved and run in the same order as they are present in the class.
- ***static* nested classes do not have access to any instance members of the enclosing outer class.** It can only access them through an object's reference.

7 – Immutability means ? Where, How and Why to use it ?

In Java, that means that the contents or state of an immutable object simply **can't be changed**.We can use it if we don't want to change something.
Immutable objects don't change their internal state in time, they are thread-safe and side-effects free. Because of those properties, immutable objects are also especially useful when dealing with multi-thread environments.

We can use **in Classes**

```
public final class Dog {
  // class contents here
}
```

The **final** keyword in the class declaration makes it such that the class can't change.

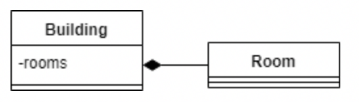
We can use **in Fields**

```
public final class Dog {
    private final String name;
}
```

final makes it such that once the variable's value is declared, it can't change.

8 – Composition and Aggregation means and differences ?

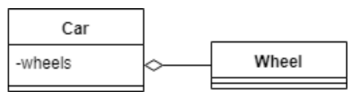
- Composition is a “**belongs-to**” type of relationship. It means that one of the objects is a logically larger structure, which contains the other object.We often call it a “**has-a**” relationship.



- For example, a room belongs to a building, or in other words a building has a room.

The lifecycles **are tied**. It means that **if we destroy the owner object**, its **members also will be destroyed** with it.
- Note that doesn’t mean that the containing object can’t exist without any of its part.For example, we can tear down all the walls inside a building, hence destroy the rooms.But the building will still exist.
- All of the parts need to have exactly one container.

- **Aggregation** is also a “**has-a**” relationship. What distinguishes it from composition, that it doesn’t involve owning.

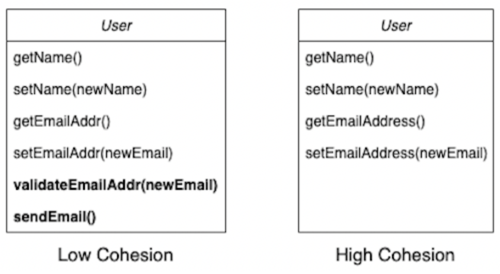


- For example, a car and its wheels. We can **take off** the wheels, and they will **still exist**.

As a result, the lifecycles of the objects **aren’t tied**: every one of them can exist **independently** of each other.
- Since aggregation doesn’t involve owning, a member **doesn’t need to be tied to only one container.**

9 – Cohesion and Coupling means and differences ?

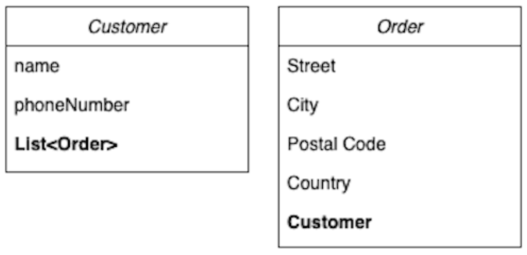
- Cohesion is the degree to which the elements inside a module belong together.
- A module with **high cohesion** contains elements that are **tightly related** to each other and **united in their purpose**.



For example, all the methods within a User class should represent the user behavior.

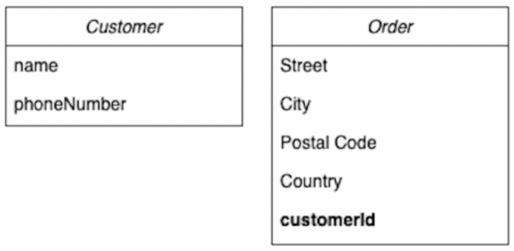
- A module is said to have **low cohesion** if it **contains unrelated elements**.For example, A User can be responsible for storing the email address of the user but not for validating it or sending an email.

- **Coupling** is the degree of interdependence between software modules.
- Two modules have **high coupling** (or tight coupling) if they are closely connected.



Tight Coupling

For example, two concrete classes storing references to each other and calling each other's methods. As shown in the diagram below, *Customer* and *Order* are tightly coupled to each other.
The *Customer* is storing the list of all the orders placed by a customer, whereas the *Order* is storing the reference to the *Customer* object.

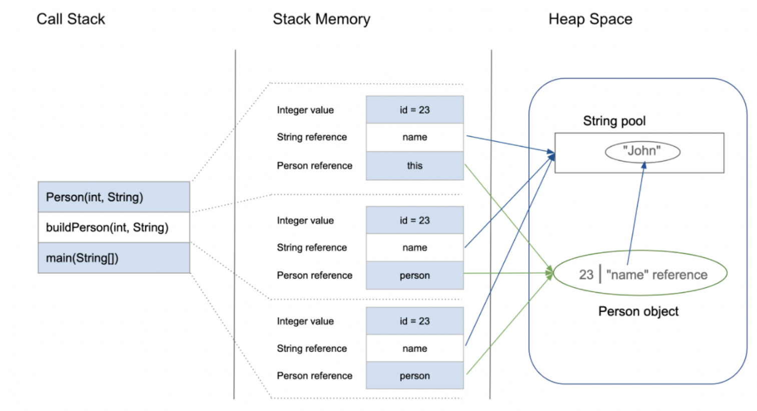


Loose Coupling

Every time the customer places a new order, we need to add it to the order list present inside the *Customer*. This seems an unnecessary dependency. Also, *Order* only needs to know the customer identifier and does need a reference to the *Customer* object.

10 - Heap and Stack means and differences ?

- Heap space is used for the dynamic memory allocation of Java objects and JRE classes at runtime. New objects are always created in heap space, and the references to these objects are stored in stack memory.



- Stack Memory in Java is used for static memory allocation and the execution of a thread. It contains primitive values that are specific to a method and references to objects referred from the method that are in a heap.
Access to this memory is in Last-In-First-Out (LIFO) order

Parameter	Stack Memory	Heap Space
Application	Stack is used in parts, one at a time during execution of a thread	The entire application uses Heap space during runtime
Size	has size limits depending upon OS	There is no size limit
Storage	Stores only primitive variables and references to objects that are created in Heap Space	All the newly created objects are stored here
Order	It's accessed using Last-in First-out (LIFO) memory allocation system	This memory is accessed via complex memory management techniques.
Life	Stack memory only exists as long as the current method is running	Heap space exists as long as the application runs
Allocation/Deallocation	This Memory is automatically allocated and deallocated when a method is called and returned, respectively	Heap space is allocated when new objects are created and deallocated by Garbage Collector when they're no longer referenced

11 – Exception means ? Type of Exceptions ?

- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

There are three types of exceptions:

1. **Checked Exception:** It is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions.
2. **Unchecked Exception:** An unchecked exception is an exception that occurs at the time of execution. These are also called Runtime Exceptions.
3. **Error:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

12 – How to summarize 'clean code' as short as possible ?

If I had to summarize what clean code means in one sentence, I would say that for me;
"Clean code is code that is easy to understand and easy to change."
Easy to understand means the code is **easy to read**, whether that reader is the original author of the code or somebody else.
Easy to change means the code is **easy to extend** and **refactor**, and it's **easy to fix bugs** in the codebase.

13 - What is the method of hiding in Java ?

When a child class defines a static method with the same signature as a static method in the parent class, then the child's method *hides* the one in the parent class.

```
public class BaseMethodClass {

    public static void printMessage() {
        System.out.println("base static method");
    }
}
```

BaseMethodClass has a single *printMessage() static* method.

Next, let's create a child class with the same signature as in the base class:

```
public class ChildMethodClass extends BaseMethodClass {

    public static void printMessage() {
        System.out.println("child static method");
    }
}
```

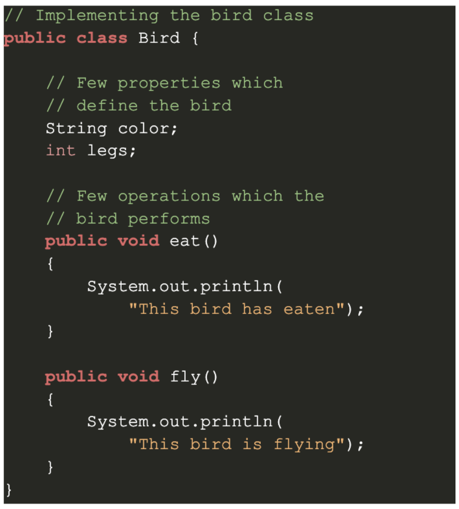
```
ChildMethodClass.printMessage();
```

The output after calling the *printMessage()* method:

```
child static method
```

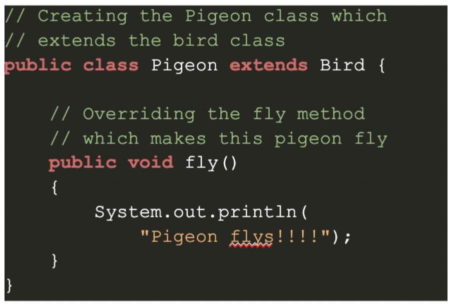
The *ChildMethodClass.printMessage()* hides the method in *BaseMethodClass*.

14 - What is the difference between abstraction and polymorphism in Java ?



```
// Implementing the bird class
public class Bird {

    // Few properties which
    // define the bird
    String color;
    int legs;

    // Few operations which the
    // bird performs
    public void eat()
    {
        System.out.println(
            "This bird has eaten");
    }

    public void fly()
    {
        System.out.println(
            "This bird is flying");
    }
}
```

Polymorphism is the ability of the **same object** to **take multiple forms**.



```
// Creating the Pigeon class which
// extends the bird class
public class Pigeon extends Bird {

    // Overriding the fly method
    // which makes this pigeon fly
    public void fly()
    {
        System.out.println(
            "Pigeon flys!!!!");
    }
}
```

In this example, a pigeon is inherently a bird.
Bird bird = new Pigeon();

Abstraction in general means hiding.The user is simply interested in seeing the pigeon fly but not interested in how the bird is actually flying. Therefore, in the above scenario where the user wishes to make it fly, he will simply call the fly method by using **pigeon.fly()** where the pigeon is the object of the bird pigeon.

References:

https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/
https://www.upgrad.com/blog/modularity-in-java/
https://www.javatpoint.com/java-oops-concepts#oops
https://www.roberthalf.com/blog/salaries-and-skills/4-advantages-of-object-oriented-programming
https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/
https://www.geeksforgeeks.org/java-and-multiple-inheritance/
https://www.baeldung.com/java-composition-aggregation-association
https://www.baeldung.com/java-variable-method-hiding