Homework 1- Mustafa YAŞAR

1. Why do we need to use OOP? Some major OOP languages?

a) Modularity for easier troubleshooting

When working with object-oriented programming languages, you know exactly where to look when something goes wrong. “Oh, the car object broke down? The problem must be in the Car class!” You don’t have to go line-by-line through all your code.

That’s the beauty of encapsulation. Objects are self-contained, and each bit of functionality does its own thing while leaving the other bits alone. Also, this modularity allows an IT team to work on multiple objects simultaneously while minimizing the chance that one person might duplicate someone else’s functionality.

b) Reuse of code through inheritance

Suppose that in addition to your Car object, one colleague needs a RaceCar object, and another needs a Limousine object. Everyone builds their objects separately but discovers commonalities between them. In fact, each object is just a different kind of Car. This is where the inheritance technique saves time: Create one generic class (Car), and then define the subclasses (RaceCar and Limousine) that are to inherit the generic class’s traits.

Of course, Limousine and RaceCar still have their unique attributes and functions. If the RaceCar object needs a method to “fireAfterBurners” and the Limousine object requires a Chauffeur, each class could implement separate functions just for itself. However, because both classes inherit key aspects from the Car class, for example the “drive” or “fillUpGas” methods, your inheriting classes can simply reuse existing code instead of writing these functions all over again.

What if you want to make a change to all Car objects, regardless of type? This is another advantage of the OOP approach. Make a change to your Car class, and all car objects will simply inherit the new code.

c) Flexibility through polymorphism

Riffing on this example, you now need just a few drivers, or functions, like “driveCar,” driveRaceCar” and “DriveLimousine.” RaceCarDrivers share some traits with LimousineDrivers, but other things, like RaceHelmets and BeverageSponsorships, are unique.

This is where object-oriented programming’s polymorphism comes into play. Because a single function can shape-shift to adapt to whichever class it’s in, you could create

one function in the parent Car class called “drive” — not “driveCar” or “driveRaceCar,” but just “drive.” This one function would work with the RaceCarDriver, LimousineDriver and so on. In fact, you could even have “raceCar.drive(myRaceCarDriver)” or “limo.drive(myChauffeur).”

d) Effective problem solving

Many people avoid learning OOP because the learning curve seems steeper than that for top-down programming. But take the time to master OOP and you’ll find it’s the easier, more intuitive approach for developing big projects.

Object-oriented programming is ultimately about taking a huge problem and breaking it down to solvable chunks. For each mini-problem, you write a class that does what you require. And then — best of all — you can reuse those classes, which makes it even quicker to solve the next problem.

This isn’t to say that OOP is the only way to write software. But there’s a reason that languages like C++, C# and Java are the go-to options for serious software development.[1]

Java, Python, C++, Lisp, and Perl are all examples of popular object-oriented programming languages. They support programming using the classes and objects paradigm.

Five of the most popular object-oriented languages include:

- Java
- Python
- C++
- Ruby
- C#

Java – Java is everywhere, and it is one of the most used and in-demand languages of all time. Java’s motto is ‘write once, run anywhere,’ and that is reflected in the number of platforms it runs on and places it’s used.

Python – Python is general-purpose and used in many places. However, Python has a strong foothold in machine learning and data science. It’s one of the preferred languages for that new and ever-growing field.

C++ – C++has the speed of C with the functionality of classes and an object-oriented paradigm. It’s a compiled, reliable, and powerful language. In fact, it’s even used to build compilers and interpreters for other languages.

Ruby – Ruby is another general-purpose programming language. It was built for simplicity. With that said, Ruby is an incredibly powerful language. The creator of Ruby,

Yukihiro “Matz” Matsumoto, has said, “Ruby is very simple in appearance, but is very complex inside, just like our human body.”

C# – C# is a programming language designed by Microsoft. It was designed to improve upon existing concepts in C. C# powers the Microsoft .NET framework alongside many web apps, games, desktop apps, and mobile apps.

There are other object-oriented languages that we haven’t covered above. Perl, Objective-C, Dart, Lisp, JavaScript, and PHP are all object-oriented too or support object-oriented principles. [2]

2. Interface vs Abstract class?

An abstract class permits you to make functionality that subclasses can implement or override whereas an interface only permits you to state functionality but not to implement it. A class can extend only one abstract class while a class can implement multiple interfaces. [3]

Parameters	Interface	Abstract class
Speed	Slow	Fast
Multiple Inheritances	Implement several Interfaces	Only one abstract class
Structure	Abstract methods	Abstract & concrete methods
When to use	Future enhancement	To avoid independence
Inheritance/ Implementation	A Class can implemente multiple interfaces	The class can inherit only one Abstract Class
Default Implementation	While adding new stuff to the interface, it is a nightmare to find all the implementors and implemente newly defined stuff.	In case of Abstract Class, you can take advantage of the default implementation.
Access Modifiers	The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier.	Abstract Class can have an access modifier.
When to use	It is better to use interface when various implementations share only method signature. Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior.
Data fields	the interface cannot contain data fields.	the class can have data fields.
Multiple Inheritance Default	A class may implement numerous interfaces.	A class inherits only one abstract class.
Implementation	An interface is abstract so that it can’t provide any code.	An abstract class can give complete, default code which should be overridden.
Use of Access modifiers	You cannot use access modifiers for the method, properties, etc.	You can use an abstract class which contains access modifiers.
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.
Defined fields	No fields can be defined	An abstract class allows you to define both fields and constants
Inheritance	An interface can inherit multiple interfaces but cannot inherit a class.	An abstract class can inherit a class and multiple interfaces.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.

Limit of Extensions	It can extend any number of interfaces.	It can extend only one class or one abstract class at a time.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.
Class type	An interface can have only public abstract methods.	An abstract class has protected and public abstract methods.

Sample code for Interface and Abstract Class in Java

- ### Interface Syntax

```
interface name{
//methods
}
```

- ### Java Interface Example:

```
interface Pet {
    public void test();
}
class Dog implements Pet {
    public void test() {
        System.out.println("Interface Method Implemented");
    }
    public static void main(String args[]) {
        Pet p = new Dog();
        p.test();
    }
}
```

- ### Abstract Class Syntax

```
abstract class name{
    // code
}
```

- ### Abstract class example:

```
abstract class Shape {
    int b = 20;
    abstract public void calculateArea();
}

public class Rectangle extends Shape {
    public static void main(String args[]) {
        Rectangle obj = new Rectangle();
        obj.b = 200;
        obj.calculateArea();
    }
    public void calculateArea() {
        System.out.println("Area is " + (b * b));
    }
}
```

3. Why we need equals and hashcodes? 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. [4]

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

```
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.

```
@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?

In Java, the diamond problem is related to multiple inheritance. Sometimes it is also known as the deadly diamond problem or deadly diamond of death. In this section, we will learn what is the demand problem in Java and what is the solution to the diamond problem.

Before moving to the diamond problem let's have a look at inheritance in Java. [5]

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 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**.

```
// 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?

Java applications obtain objects in memory as needed. It is the task of garbage collection (GC) in the Java virtual machine (JVM) to automatically determine what memory is no longer being used by a Java application and to recycle this memory for other uses. Because memory is automatically reclaimed in the JVM, Java application developers are not burdened with having to explicitly free memory objects that are not being used.

The GC operation is based on the premise that most objects used in the Java code are short-lived and can be reclaimed shortly after their creation. Because unreferenced objects are automatically removed from the heap memory, GC makes Java memory-efficient.

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of application program bugs are eliminated or substantially reduced by GC:

- Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use with unpredictable results.
- Double free bugs, which occur when the program tries to free a region of memory that has already been freed and perhaps already been allocated again.
- Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion.

There are two types of garbage collection activity that usually happens in Java:

- A minor or incremental garbage collection is said to have occurred when unreachable objects in the young generation heap memory are removed.
- A major or full garbage collection is said to have occurred when the objects that survived the minor garbage collection and copied into the old generation or permanent generation heap memory are removed. When compared to young generation, garbage collection happens less frequently in old generation.

To free up memory, the JVM must stop the application from running for at least a short time and execute GC. This process is called "stop-the-world." This means all the threads, except for the GC threads, will stop executing until the GC threads are executed and objects are freed up by the garbage collector.

Modern GC implementations try to minimize blocking "stop-the-world" stalls by doing as much work as possible on the background (i.e. using a separate thread), for example marking unreachable garbage instances while the application process continues to run.

Garbage collection consumes CPU resources for deciding which memory to free. Various garbage collectors have been developed over time to reduce the application pauses that occur during garbage collection and at the same time to improve on the performance hit associated with garbage collection.

The traditional Oracle HotSpot JVM has four ways of performing the GC activity:

- Serial where just one thread executed the GC
- Parallel where multiple minor threads are executed simultaneously each executing a part of GC
- Concurrent Mark Sweep (CMS), which is similar to parallel, also allows the execution of some application threads and reduces the frequency of stop-the-world GC
- G1 which is also run in parallel and concurrently but functions differently than CMS

Many JVMs, such as Oracle HotSpot, JRockit, OpenJDK, IBM J9, and SAP JVM, use stop-the-world GC techniques. Modern JVMs like Azul Zing use Continuously Concurrent Compacting Collector (C4), which eliminates the stop-the-world GC pauses that limit scalability in the case of conventional JVMs.

Java applications obtain objects in memory as needed. It is the task of garbage collection (GC) in the Java virtual machine (JVM) to automatically determine what memory is no longer being used by a Java application and to recycle this memory for other uses. Because memory is automatically reclaimed in the JVM, Java application developers are not burdened with having to explicitly free memory objects that are not being used.

The GC operation is based on the premise that most objects used in the Java code are short-lived and can be reclaimed shortly after their creation. Because unreferenced objects are automatically removed from the heap memory, GC makes Java memory-efficient.

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of application program bugs are eliminated or substantially reduced by GC:

- Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use with unpredictable results.
- Double free bugs, which occur when the program tries to free a region of memory that has already been freed and perhaps already been allocated again.
- Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion.

There are two types of garbage collection activity that usually happens in Java:

- A minor or incremental garbage collection is said to have occurred when unreachable objects in the young generation heap memory are removed.
- A major or full garbage collection is said to have occurred when the objects that survived the minor garbage collection and copied into the old generation or permanent generation heap memory are removed. When compared to young generation, garbage collection happens less frequently in old generation.

To free up memory, the JVM must stop the application from running for at least a short time and execute GC. This process is called "stop-the-world." This means all the threads, except for the GC threads, will stop executing until the GC threads are executed and objects are freed up by the garbage collector.

Modern GC implementations try to minimize blocking "stop-the-world" stalls by doing as much work as possible on the background (i.e. using a separate thread), for example marking unreachable garbage instances while the application process continues to run.

Garbage collection consumes CPU resources for deciding which memory to free. Various garbage collectors have been developed over time to reduce the application pauses that occur during garbage collection and at the same time to improve on the performance hit associated with garbage collection.

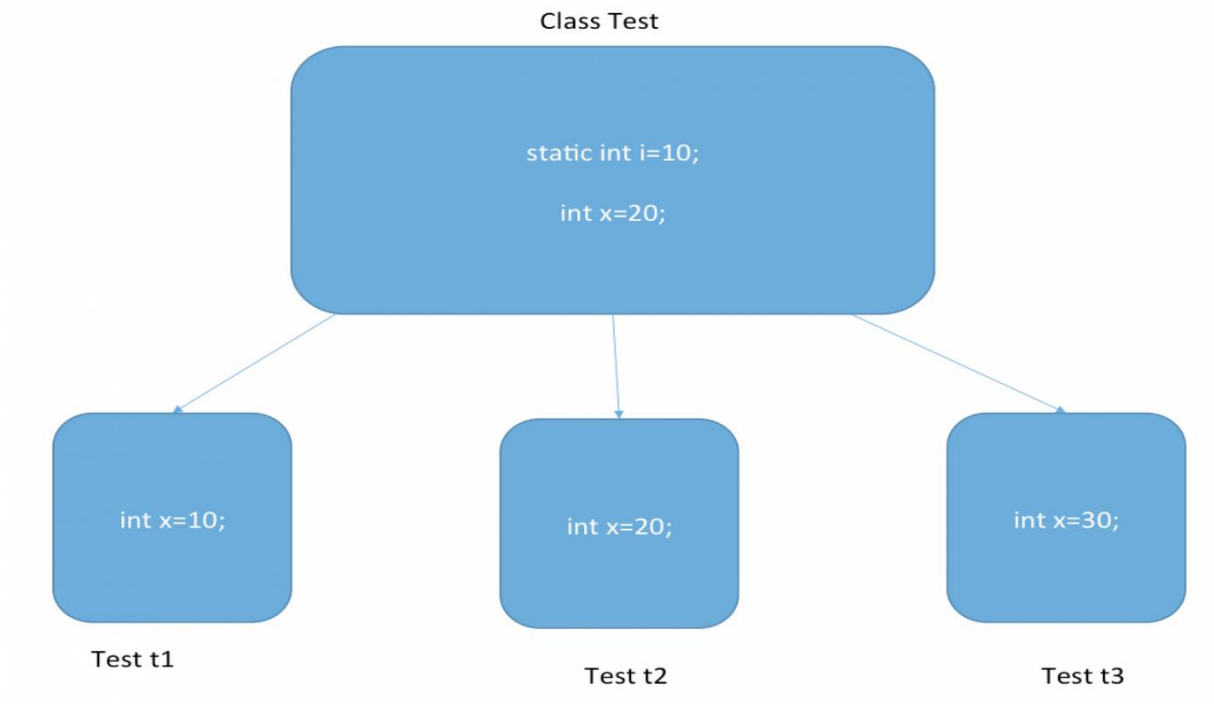
The traditional Oracle HotSpot JVM has four ways of performing the GC activity:

- Serial where just one thread executed the GC
- Parallel where multiple minor threads are executed simultaneously each executing a part of GC
- Concurrent Mark Sweep (CMS), which is similar to parallel, also allows the execution of some application threads and reduces the frequency of stop-the-world GC
- G1 which is also run in parallel and concurrently but functions differently than CMS

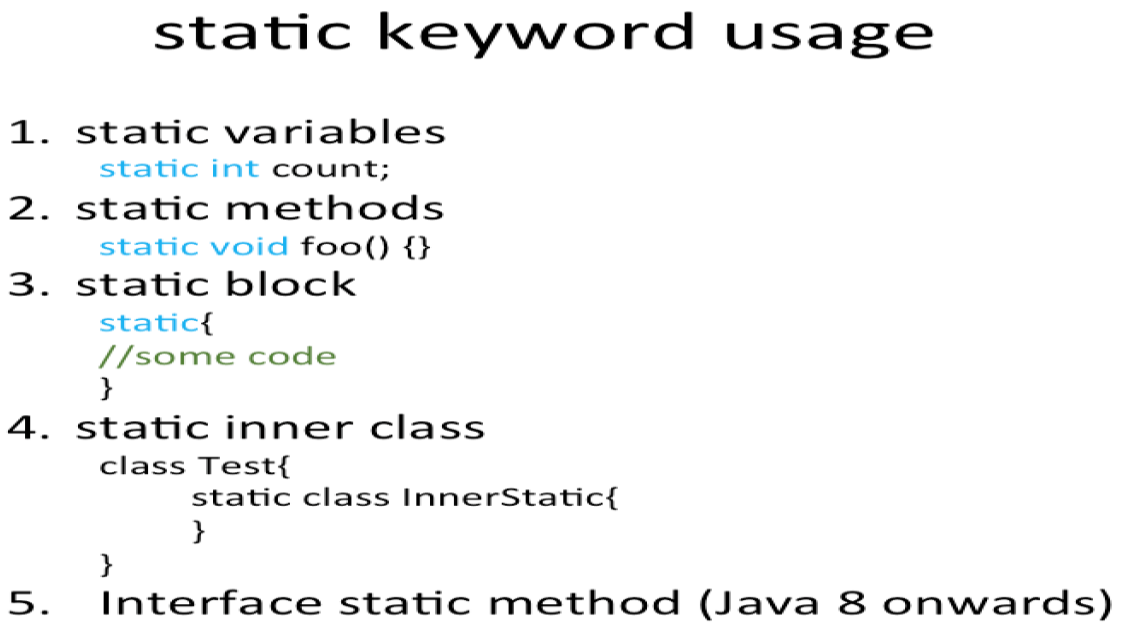
Many JVMs, such as Oracle HotSpot, JRockit, OpenJDK, IBM J9, and SAP JVM, use stop-the-world GC techniques. Modern JVMs like Azul Zing use Continuously Concurrent Compacting Collector (C4), which eliminates the stop-the-world GC pauses that limit scalability in the case of conventional JVMs. [6]

6. Immutability means? Where, How and Why to use it?

static keyword in Java is used a lot in java programming. Java static keyword is used to create a Class level variable in java. static variables and methods are part of the class, not the instances of the class.



Java static keyword can be used in five cases as shown in below image.



We will discuss four of them here, the fifth one was introduced in Java 8 and that has been discussed at Java 8 interface changes.

a) Java static variable

We can use **static** keyword with a class level variable. A static variable is a class variable and doesn't belong to Object/instance of the class.

Since static variables are shared across all the instances of Object, they are not thread safe.

Usually, static variables are used with the final keyword for common resources or constants that can be used by all the objects. If the static variable is not private, we can access it with **ClassName.variableName**

```
//static variable example
private static int count;
public static String str;
public static final String DB_USER = "myuser";
```

b) Java static method

Same as static variable, static method belongs to class and not to class instances.

A static method can access only static variables of class and invoke only static methods of the class.

Usually, static methods are utility methods that we want to expose to be used by other classes without the need of creating an instance. For example, Collections class.

Java Wrapper classes and utility classes contains a lot of static methods. The main () method that is the entry point of a java program itself is a static method.

```
//static method example
public static void setCount(int count) {
    if(count > 0)
    StaticExample.count = count;
}

//static util method
public static int addInts(int i, int...js){
    int sum=i;
    for(int x : js) sum+=x;
    return sum;
}
```

c) Java static block

Java static block is the group of statements that gets executed when the class is loaded into memory by Java ClassLoader.

Static block is used to initialize the static variables of the class. Mostly it's used to create static resources when the class is loaded.

We can't access non-static variables in the static block. We can have multiple static blocks in a class, although it doesn't make much sense. Static block code is executed only once when the class is loaded into memory.

```
static{
    //can be used to initialize resources when class is loaded
    System.out.println("StaticExample static block");
    //can access only static variables and methods
    str="Test";
    setCount(2);
}
```

We can use static keyword with nested classes. static keyword can't be used with top-level classes.

A static nested class is same as any other top-level class and is nested for only packaging convenience.

Let's see all the static keyword in java usage in a sample program.

StaticExample.java

```
package com.journaldev.misc;

public class StaticExample {

    //static block
    static{
        //can be used to initialize resources when class is loaded
        System.out.println("StaticExample static block");
        //can access only static variables and methods
        str="Test";
        setCount(2);
    }

    //multiple static blocks in same class
    static{
        System.out.println("StaticExample static block2");
    }

    //static variable example
    private static int count; //kept private to control its value through setter
    public static String str;

    public int getCount() {
        return count;
    }

    //static method example
    public static void setCount(int count) {
        if(count > 0)
        StaticExample.count = count;
    }

    //static util method
    public static int addInts(int i, int...js){
        int sum=i;
        for(int x : js) sum+=x;
        return sum;
    }

    //static class example - used for packaging convenience only
    public static class MyStaticClass{
        public int count;

    }

}

```

Let's see how to use static variable, method and static class in a test program.

TestStatic.java

```
package com.journaldev.misc;

public class TestStatic {

    public static void main(String[] args) {
        StaticExample.setCount(5);

        //non-private static variables can be accessed with class name
        StaticExample.str = "abc";
        StaticExample se = new StaticExample();
        System.out.println(se.getCount());
        //class and instance static variables are same
        System.out.println(StaticExample.str +" is same as "+se.str);
        System.out.println(StaticExample.str == se.str);

        //static nested classes are like normal top-level classes
        StaticExample.MyStaticClass myStaticClass = new
StaticExample.MyStaticClass();
        myStaticClass.count=10;

        StaticExample.MyStaticClass myStaticClass1 = new
StaticExample.MyStaticClass();
        myStaticClass1.count=20;

        System.out.println(myStaticClass.count);
        System.out.println(myStaticClass1.count);
    }

}
```

The output of the above static keyword in java example program is:

```
StaticExample static block
StaticExample static block2
5
abc is same as abc
true
10
20
```

Notice that static block code is executed first and only once as soon as class is loaded into memory. Other outputs are self-explanatory.

Java static import

Normally we access static members using Class reference, from Java 1.5 we can use java static import to avoid class reference. Below is a simple example of Java static import.

```
package com.journaldev.test;

public class A {

        public static int MAX = 1000;

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

```
package com.journaldev.test;

import static com.journaldev.test.A.MAX;
import static com.journaldev.test.A.foo;

public class B {

        public static void main(String args[]){
                System.out.println(MAX); //normally A.MAX
                foo(); // normally A.foo()
        }
}
```

Notice the import statements, for static import we have to use import static followed by the fully classified static member of a class. For importing all the static members of a class, we can use * as in import static com.journaldev.test.A.*; We should use it only when we are using the static variable of a class multiple times, it's not good for readability.

Update: I have recently created a video to explain static keyword in java, you should watch it below. [7]

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

An immutable object is an object whose internal state remains constant after it has been entirely created.

This means that the public API of an immutable object guarantees us that it will behave in the same way during its whole lifetime.

If we take a look at the class String, we can see that even when its API seems to provide us a mutable behavior with its replace method, the original String doesn't change:

```
String name = "baeldung";
String newName = name.replace("dung", "----");

assertEquals("baeldung", name);
assertEquals("bael----", newName);
```

The API gives us read-only methods, it should never include methods that change the internal state of the object.

a) The final Keyword in Java

Before trying to achieve immutability in Java, we should talk about the final keyword.

In Java, variables are mutable by default, meaning we can change the value they hold.

By using the final keyword when declaring a variable, the Java compiler won't let us change the value of that variable. Instead, it will report a compile-time error:

```
final String name = "baeldung";
name = "bael...";
```

Note that final only forbids us from changing the reference the variable holds, it doesn't protect us from changing the internal state of the object it refers to by using its public API:

```
final List<String> strings = new ArrayList<>();
assertEquals(0, strings.size());
strings.add("baeldung");
assertEquals(0, strings.size());
```

The second assertEquals will fail because adding an element to the list changes its size, therefore, it isn't an immutable object.

b) Immutability in Java

Now that we know how to avoid changes to the content of a variable, we can use it to build the API of immutable objects.

Building the API of an immutable object requires us to guarantee that its internal state won't change no matter how we use its API.

A step forward in the right direction is to use final when declaring its attributes:

```
class Money {
    private final double amount;
    private final Currency currency;

    // ...
}
```

Note that Java guarantees us that the value of amount won't change, that's the case with all primitive type variables.

However, in our example we are only guaranteed that the currency won't change, so we must rely on the Currency API to protect itself from changes.

Most of the time, we need the attributes of an object to hold custom values, and the place to initialize the internal state of an immutable object is its constructor:

```
class Money {
    // ...
    public Money(double amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public Currency getCurrency() {
        return currency;
    }

    public double getAmount() {
        return amount;
    }
}
```

As we've said before, to meet the requirements of an immutable API, our Money class only has read-only methods.

Using the reflection API, we can break immutability and change immutable objects. However, reflection violates immutable object's public API, and usually, we should avoid doing this.[8]

8. Composition and Aggregation means and differences?

The composition is a design technique in java to implement a has-a relationship. The composition is achieved by using an instance variable that refers to other objects. If an object contains the other object and the contained object cannot exist without the existence of that object, then it is called composition.

Aggregation in Java is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship. It is a more specialized version of the association relationship. The aggregate class contains a reference to another class and is said to have ownership of that class. Each class referenced is considered to be part-of the aggregate class.

In composition, the dependent object cannot exist without the parent. Whereas, in aggregation, the dependent objects can exists without a parent. The composition is implemented in java by having non-static inner class but aggregation by having a static inner class or object references.

Aggregation vs Composition

1. Dependency: Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don't exist separate to a Human

2. Type of Relationship: Aggregation relation is "has-a" and composition is "part-of" relation.

3. Type of association: Composition is a strong Association whereas Aggregation is a weak Association. [9]

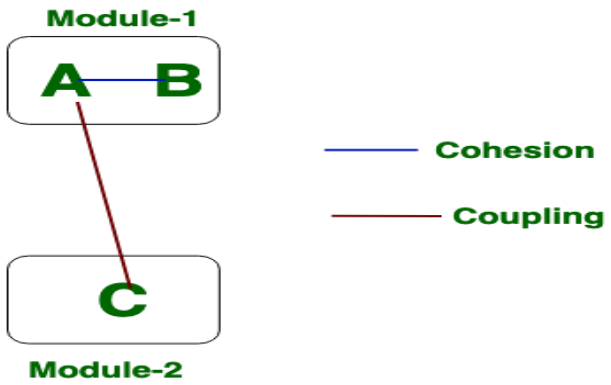
9. Cohesion and Coupling means and differences?

Cohesion:

Cohesion is the indication of the relationship within module. It is concept of intra-module. Cohesion has many types but usually highly cohesion is good for software.

Coupling:

Coupling is also the indication of the relationships between modules. It is concept of Inter-module. Coupling has also many types but usually low coupling is good for software.



Now we will see the difference between Cohesion and Coupling. the differences between cohesion and coupling are given below: [10]

Cohesion	Coupling
Cohesion is the concept of intra module.	Coupling is the concept of inter module.
Cohesion represents the relationship within module.	Coupling represents the relationships between modules.
Increasing in cohesion is good for software.	Increasing in coupling is avoided for software.
Cohesion represents the functional strength of modules.	Coupling represents the independence among modules.
Highly cohesive gives the best software.	Where as loosely coupling gives the best software.
In cohesion, module focuses on the single thing.	In coupling, modules are connected to the other modules.

10.Heap and Stack means and differences?

Stack Memory

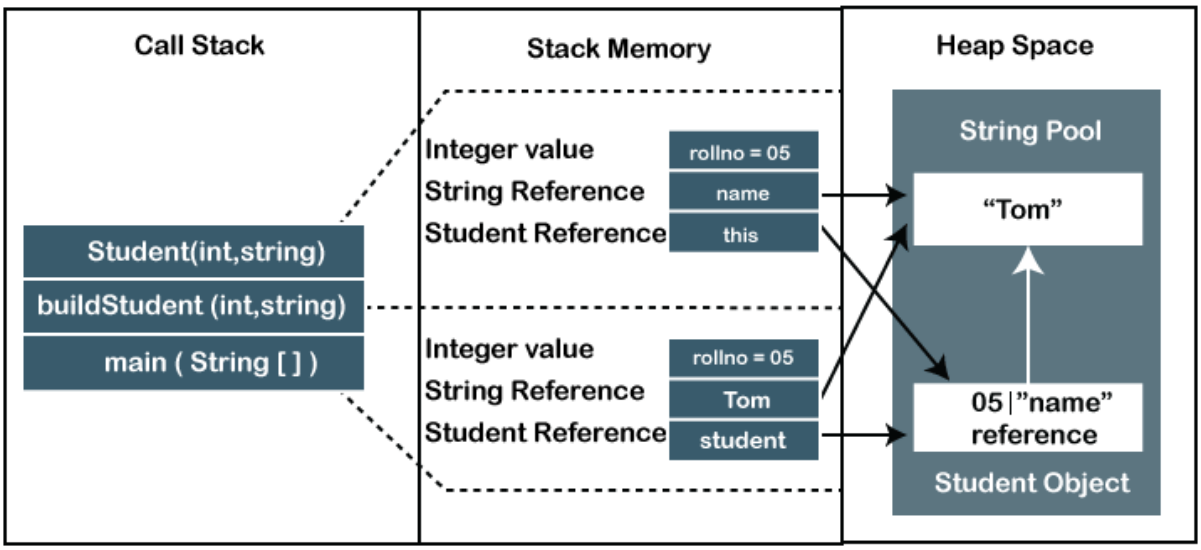
The stack memory is a physical space (in RAM) allocated to each thread at run time. It is created when a thread creates. Memory management in the stack follows LIFO (Last-In-First-Out) order because it is accessible globally. It stores the variables, references to objects, and partial results. Memory allocated to stack lives until the function returns. If there is no space for creating the new objects, it throws the java.lang.StackOverFlowError. The scope of the elements is limited to their threads. The JVM creates a separate stack for each thread.

Heap Memory

It is created when the JVM starts up and used by the application as long as the application runs. It stores objects and JRE classes. Whenever we create objects it occupies space in the heap memory while the reference of that object creates in the stack. It does not follow any order like the stack. It dynamically handles the memory blocks. It means, we need not to handle the memory manually. For managing the memory automatically, Java provides the garbage collector that deletes the objects which are no longer being used. Memory allocated to heap lives until any one event, either program terminated or memory free does not occur. The elements are globally accessible in the application. It is a common memory space shared with all the threads. If the heap space is full, it throws the java.lang.OutOfMemoryError. The heap memory is further divided into the following memory areas:

- Young generation
- Survivor space
- Old generation
- Permanent generation
- Code Cache

The following image shows the allocation of stack memory and heap space.



Difference Between Stack and Heap Memory

The following table summarizes all the major differences between stack memory and heap space. [11]

Parameter	Stack Memory	Heap Space
Application	It stores items that have a very short life such as methods, variables, and reference variables of the objects.	It stores objects and Java Runtime Environment (JRE) classes.
Ordering	It follows the LIFO order.	It does not follow any order because it is a dynamic memory allocation and does not have any fixed pattern for allocation and deallocation of memory blocks.
Flexibility	It is not flexible because we cannot alter the allocated memory.	It is flexible because we can alter the allocated memory.
Efficiency	It has faster access, allocation, and deallocation.	It has slower access, allocation, and deallocation.
Memory Size	It is smaller in size.	It is larger in size.
Java Options Used	We can increase the stack size by using the JVM option -Xss.	We can increase or decrease the heap memory size by using the -Xmx and -Xms JVM options.
Visibility or Scope	The variables are visible only to the owner thread.	It is visible to all threads.
Generation of Space	When a thread is created, the operating system automatically allocates the stack.	To create the heap space for the application, the language first calls the operating system at run time.
Distribution	Separate stack is created for each object.	It is shared among all the threads.

Exception Throws	JVM throws the java.lang.StackOverFlowError if the stack size is greater than the limit. To avoid this error, increase the stack size.	JVM throws the java.lang.OutOfMemoryError if the JVM is unable to create a new native method.
Allocation/ Deallocation	It is done automatically by the compiler.	It is done manually by the programmer.
Cost	Its cost is less.	Its cost is more in comparison to stack.
Implementation	Its implementation is hard.	Its implementation is easy.
Order of allocation	Memory allocation is continuous.	Memory allocated in random order.
Thread-Safety	It is thread-safe because each thread has its own stack.	It is not thread-safe, so properly synchronization of code is required.

11.Exception means? Type of Exceptions?

The term exception is shorthand for the phrase "exceptional event."

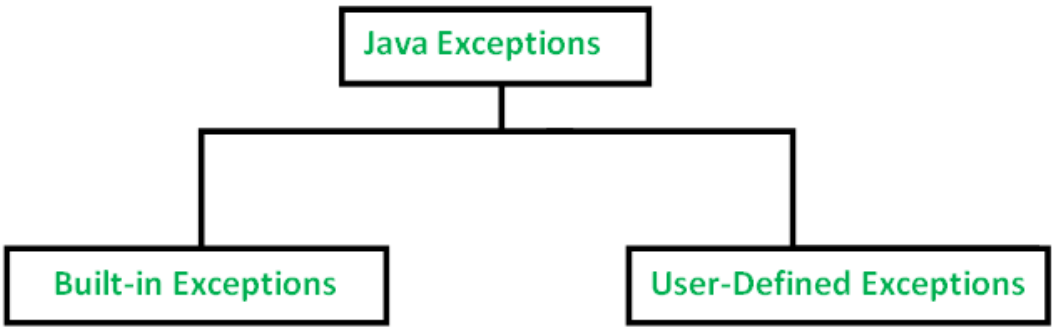
Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the next figure).[12]

Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



1. *User-Defined Exceptions*

a) ArithmeticException

It is thrown when an exceptional condition has occurred in an arithmetic operation.

b) ArrayIndexOutOfBoundsException

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

c) ClassNotFoundException

This Exception is raised when we try to access a class whose definition is not found

d) FileNotFoundException

This Exception is raised when a file is not accessible or does not open.

e) IOException

It is thrown when an input-output operation failed or interrupted

f) InterruptedException

It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

g) NoSuchFieldException

It is thrown when a class does not contain the field (or variable) specified

h) NoSuchMethodException

It is thrown when accessing a method which is not found.

i) NullPointerException

This exception is raised when referring to the members of a null object. Null represents nothing

j) NumberFormatException

This exception is raised when a method could not convert a string into a numeric format.

k) RuntimeException

This represents any exception which occurs during runtime.

l) StringIndexOutOfBoundsException

It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

2. *User-Defined Exceptions*

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’. [13]

12.How to summarize ‘clean code’ as short as possible?

Code is clean if it can be understood easily – by everyone on the team. Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability.

General rules

1. Follow standard conventions.
2. Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Boy scout rule. Leave the campground cleaner than you found it.
4. Always find root cause. Always look for the root cause of a problem.

Design rules

1. Keep configurable data at high levels.
2. Prefer polymorphism to if/else or switch/case.
3. Separate multi-threading code.
4. Prevent over-configurability.
5. Use dependency injection.
6. Follow Law of Demeter. A class should know only its direct dependencies.

Understandability tips

1. Be consistent. If you do something a certain way, do all similar things in the same way.
2. Use explanatory variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
4. Prefer dedicated value objects to primitive type.
5. Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
6. Avoid negative conditionals.

Names rules

1. Choose descriptive and unambiguous names.
2. Make meaningful distinction.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.
6. Avoid encodings. Don't append prefixes or type information.

Functions rules

1. Small.
2. Do one thing.
3. Use descriptive names.
4. Prefer fewer arguments.
5. Have no side effects.
6. Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

Comments rules

1. Always try to explain yourself in code.
2. Don't be redundant.
3. Don't add obvious noise.
4. Don't use closing brace comments.
5. Don't comment out code. Just remove.
6. Use as explanation of intent.
7. Use as clarification of code.
8. Use as warning of consequences.

Source code structure

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don't use horizontal alignment.
9. Use white space to associate related things and disassociate weakly related.
10. Don't break indentation.

Objects and data structures

1. Hide internal structure.
2. Prefer data structures.
3. Avoid hybrids structures (half object and half data).
4. Should be small.
5. Do one thing.
6. Small number of instance variables.

7. Base class should know nothing about their derivatives.
8. Better to have many functions than to pass some code into a function to select a behavior.
9. Prefer non-static methods to static methods.

Tests

1. One assert per test.
2. Readable.
3. Fast.
4. Independent.
5. Repeatable.

Code smells

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand. [14]

13. What is the method of hiding in Java?

Method hiding may happen in any hierarchy structure 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. To learn more about the static keyword, this write-up is a good place to start.

The same behavior involving the instance methods is called method overriding. To learn more about method overriding checkout our guide here.

Now, let's have a look at this practical example:

```
public class BaseMethodClass {

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

BaseMethodClass has a single printMessage() static method. [15]

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");
    }
}
```

Here's how it works:

```
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?

Encapsulation in Java

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Abstraction in Java

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects. [16]

Difference between Abstraction and Encapsulation:

Abstraction	Encapsulation
Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.

Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public.
In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

15.References

[1] https://careerkarma.com/blog/object-oriented-languages/

[2] https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

[3] https://www.guru99.com/interface-vs-abstract-class-java.html

[4] https://www.baeldung.com/java-equals-hashcode-contracts

[5] https://www.javatpoint.com/what-is-diamond-problem-in-java

[6] https://www.eginnovations.com/blog/what-is-garbage-collection-java/

[7] https://www.journaldev.com/1365/static-keyword-in-java

[8] https://www.baeldung.com/java-immutable-object

[9] https://www.geeksforgeeks.org/association-composition-aggregation-java/

[10] https://www.javatpoint.com/stack-vs-heap-java

[11] https://www.javatpoint.com/stack-vs-heap-java

[12] https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

[13] https://www.geeksforgeeks.org/types-of-exception-in-java-with-examples/

[14] https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

[15] https://www.baeldung.com/java-variable-method-hiding

[16] https://www.geeksforgeeks.org/difference-between-abstraction-and-encapsulation-in-java-with-examples/