**Introduce about yourself**

**Core Java**

**Java 8 Features**

**Spring Framework**

**Hibernate JPA**

**Spring Boot**

**Spring Cloud**

**Spring Junit Testing**

**Java Design Pattern**

**Java SOLID Principles**

**Micro Services Architecture**

**Micro Services Design Principles**

Introduce about yourself

Hi, I am Naveena, based out of Bangalore. I have around 10 years of experience in developing enterprise applications using Java using modern frameworks like Core Java 8, Spring, Spring Boot, and Hibernate. Following my engineering graduation, I started my career with Igate , working for five years. I then pursued a career in Wipro technology for five years, where I had the opportunity to work on an airline product end-to-end, from development to production and live support.

I was playing a lead role, responsible for the design, development, and code review of some critical deliverables.

Core Java

OOPS Concepts:

Java is a class-based object-oriented programming (OOP) language that is built around the concept of objects. OOP concepts (OOP) intend to improve code readability and reusability by defining how to structure a Java program efficiently. The main principles of object-oriented programming are:

**Abstraction**

**Encapsulation**

**Inheritance**

**Polymorphism**

**Association**

**Aggregation**

**Composition**

**Abstraction**

Abstraction aims to hide complexity from the users and show them only the relevant information. For example, if you want to drive a car, you don’t need to know about its internal workings. The same is true of Java classes. You can hide internal implementation details by using abstract classes or interfaces. On the abstract level, you only need to define the method signatures (name and parameter list) and let each class implement them in their own way.

**Abstraction in Java:**

* Hides the underlying complexity of data
* Helps avoid repetitive code
* Presents only the signature of internal functionality
* Gives flexibility to programmers to change the implementation of the abstract behaviour
* Partial abstraction (0-100%) can be achieved with abstract classes
* Total abstraction (100%) can be achieved with interfaces

**Encapsulation**

Encapsulation allows us to protect the data stored in a class from system-wide access. As its name suggests, it safeguards the internal contents of a class like a real-life capsule. You can implement encapsulation in Java by keeping the fields (class variables) private and providing public getter and setter methods to each of them. Java Beans are examples of fully encapsulated classes.

**Encapsulation in Java:**

* Restricts direct access to data members (fields) of a class.
* Fields are set to private
* Each field has a getter and setter method
* Getter methods return the field
* Setter methods let us change the value of the field

**Polymorphism**

[Polymorphism](https://en.wikipedia.org/wiki/Polymorphism)refers to the ability to perform a certain action in different ways. In Java, polymorphism can take two forms: method overloading and method overriding. Method overloading happens when various methods with the same name are present in a class. When they are called they are differentiated by the number, order, and types of their parameters. Method overriding occurs when the child class overrides a method of its parent.

**Polymorphism in Java:**

* The same method name is used several times.
* Different methods of the same name can be called from the object.
* All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class).
* Example of static polymorphism in Java is method overloading.
* Example of dynamic polymorphism in Java is method overriding.

**Inheritance**

[Inheritance](https://raygun.com/blog/oop-concepts-java/www.linkedin.com/pulse/types-relationships-object-oriented-programming-oop-sarah-el-dawody/) makes it possible to create a child class that inherits the fields and methods of the parent class. The child class can override the values and methods of the parent class, however it’s not necessary. It can also add new data and functionality to its parent. Parent classes are also called superclasses or base classes, while child classes are known as subclasses or derived classes as well. Java uses the extends keyword to implement the principle of inheritance in code.

**Inheritance in Java:**

* A class (child class) can extend another class (parent class) by inheriting its features.
* Implements the DRY (Don’t Repeat Yourself) programming principle.
* Improves code reusability.
* Multilevel inheritance is allowed in Java (a child class can have its own child class as well).
* Multiple inheritances are not allowed in Java (a class can’t extend more than one class).

**Association**

Besides the four main principles of OOP, Java also works with three further concepts (association, aggregation, composition) you can make use of when designing your programs. *Aggregation* is a special form of *association*, while *composition* is a special form of *aggregation*.

[Association](https://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/) simply means the act of establishing a relationship between two unrelated classes. For example, when you declare two fields of different types (e.g. Car and Bicycle) within the same class and make them interact with each other, you have performed association.

**Association in Java:**

* Two separate classes are associated through their objects.
* The two classes are unrelated, each can exist without the other one.
* Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship.

**Aggregation**

[Aggregation](https://beginnersbook.com/2013/05/aggregation/) is a narrower kind of association. It occurs when there’s a one-way (HAS-A) relationship between the two classes you associate through their objects. For example, every Passenger has a Car but a Car doesn’t necessarily have a Passenger. When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to. Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.

**Aggregation in Java:**

* One-directional association.
* Represents a HAS-A relationship between two classes.
* Only one class is dependent on the other.

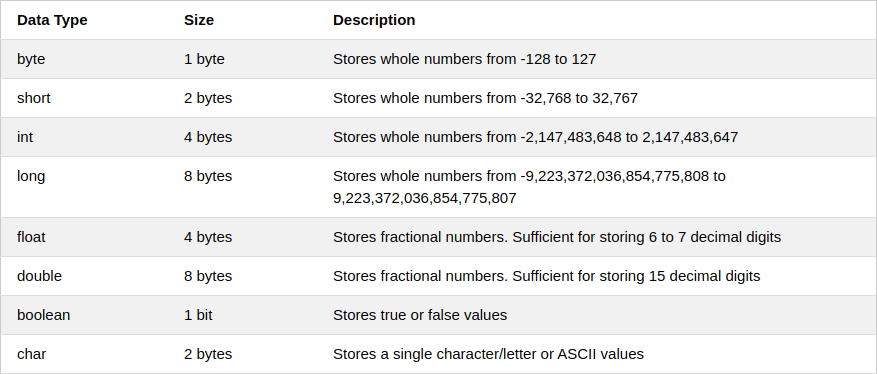
**Composition**

[Composition](https://www.infoworld.com/article/3029325/application-development/exploring-association-aggregation-and-composition-in-oop.html)is a stricter form of aggregation. It occurs when the two classes you associate are mutually dependent on each other and can’t exist without each other. For example, take a Car and an Engine class. A Car cannot run without an Engine, while an Engine also can’t function without being built into a Car. This kind of relationship between objects is also called a PART-OF relationship.

Composition in Java:

* A restricted form of aggregation
* Represents a PART-OF relationship between two classes
* Both classes are dependent on each other
* If one class ceases to exist, the other can’t survive alone

Data Types:



A Java both int and Integer are used to store integer type data the major difference between both is type of int is primitive while Integer is of class type.This difference become significant when concept of OOPs comes in picture during development as int follows the principle of primitive data type while Integer behave as a wrapper class.

Following are the important differences between int and Integer.

| **Sr. No.** | **Key** | **int** | **Integer** |
| --- | --- | --- | --- |
| 1 | Type | A int is a data type that stores 32 bit signed two's compliment integer. | On other hand Integer is a wrapper class which wraps a primitive type int into an object. |
| 2 | Purpose | int helps in storing integer value into memory. | Integer helps in converting int into object and to convert an object into int as per requirement. |
| 3 | Flexibility | int provides less flexibility as compare to Integer as it only allows binary value of an integer in it. | Integer on other hand is more flexible in storing and manipulating an int data.Since Wrapper classes inherit Object class, they can be used in collections with Object reference or generics. |
| 4 | Memory allocation | As already mentioned int is a primitive data type and takes 32 bits(4 bytes) to store. | On other hand Integer is an object which takes 128 bits (16 bytes) to store its int value. |
| 5 | Casting | In java one canâTMt assign a string value (containing an integer only) to an int variable directly or even by casting. | In case of Integer we can assign string to an object of Integer type using the Integer(String) constructor or by even use parseInt(String) to convert a String literal to an int value. |
| 6 | Direct Conversion to Other base. | In case of int we can't convert its integer value to other base. | However in Integer we can directly convert its integer value to other bases such as Binary, Octal or Hexadecimal format using toBinaryString(), toOctalString() or toHexString() respectively. |
| 7 | Allowed operations | int do not allowed any of inbuilt functions to change its value or syntax. | However in Integer we can reverse number or rotate it left or right using reverse(), rotateLeft() and rotateRight() respectively. |

## Example of int vs Integer

**JavaTester.java**

## Example

public class JavaTester {

   public static void main(String args[]){

      Integer a = new Integer("456");

      // Casting not possible

      // int a = (int)"456";

      // Casting not possible

      // int c="456";

      // Casting possible using methods

      // from Integer Wrapper class

      int b = Integer.parseInt("456");

      System.out.print(b);

   }

}

## Output

456

### ****Wrapper Classes****

#### **1. What are wrapper classes?**

A primitive wrapper class in the Java programming language is one of eight classes provided in the java.lang package to provide object methods for the eight primitive types. All of the primitive wrapper classes in Java are immutable.

**Wrapper:** Boolean, Byte, Character, Double, Float, Integer, Long, Short  
**Primitive:** boolean, byte, char, double, float, int, long, short

#### **2. Why do we need Wrapper Classes in Java?**

A wrapper class wraps (encloses) around a data type and gives it an object appearance.

Reasons why we need Wrapper Classes :-

• null is a possible value

• use it in a Collection

• Methods that support Object like creation from other types.. like String

◦ Integer number2 = new Integer(“55”); //String

#### **3. What are the different ways of creating Wrapper Class Instances?**

Two ways of creating Wrapper Class Instances are described below.

Using a Wrapper Class Constructor :

|  |
| --- |
| Integer number = new Integer(55); //int  Integer number2 = new Integer("55"); //String  Float number3 = new Float(55.0); //double argument  Float number4 = new Float(55.0 f); //float argument  Float number5 = new Float("55.0f"); //String  Character c1 = new Character('C'); //Only char constructor  //Character c2=new Character(124);//COMPILER ERROR  Boolean b = new Boolean(true);  //"true" "True" "tRUe" - all String Values give True  //Anything else gives false  Boolean b1 = new Boolean("true"); //value stored - true  Boolean b2 = new Boolean("True"); //value stored - true  Boolean b3 = new Boolean("False"); //value stored - false  Boolean b4 = new Boolean("SomeString"); //value stored - false |

valueOf Static Methods :

Provide another way of creating a Wrapper Object

|  |
| --- |
| Integer hundred = Integer.valueOf("100"); //100 is stored in variable  Integer seven = Integer.valueOf("111", 2); //binary 111 is converted to 7 |

#### **4. What are differences in the two ways of creating Wrapper Classes?**

The difference is that using the Constructor you will always create a new object, while using valueOf() static method, it may return you a cached value with-in a range.  
For example : The cached values for long are between [-128 to 127].  
We should prefer static valueOf method, because it may save you some memory. To understand it further, here is an implementation of valueOf method in the Long class

#### **5. What is Auto Boxing?**

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Example 1 :

|  |
| --- |
| Integer nineC = 9; |

Example 2 :

|  |
| --- |
| Integer ten = new Integer(10);  ten++; //allowed. Java does had work behind the screen for us |

#### **6. What are the advantages of Auto Boxing?**

Auto Boxing helps in saving memory by reusing already created Wrapper objects. Auto Boxing uses the static valueOf methods. However wrapper classes created using new are not reused.  
Two wrapper objects created using new are not same object.

|  |
| --- |
| Integer nineA = new Integer(9);  Integer nineB = new Integer(9);  System.out.println(nineA == nineB); //false  System.out.println(nineA.equals(nineB)); //true |

Two wrapper objects created using boxing are same object.

|  |
| --- |
| Integer nineC = 9;  Integer nineD = 9;  System.out.println(nineC == nineD); //true  System.out.println(nineC.equals(nineD)); //true |

#### **7. What is Casting?**

Casting is used when we want to convert on data type to another. There are two types of Casting :-

• Implicit Casting

• Explicit Casting

#### **8. What is Implicit Casting?**

Implicit Casting is done by the compiler. Good examples of implicit casting are all the automatic widening conversions i.e. storing smaller values in larger variable types.

|  |
| --- |
| int value = 100;  long number = value; //Implicit Casting  float f = 100; //Implicit   Casting |

#### **9. What is Explicit Casting?**

Explicit Casting is done through code. Good examples of explicit casting are the narrowing conversions. Storing larger values into smaller variable types

|  |
| --- |
| long number1 = 25678;  int number2 = (int) number1; //Explicit Casting  //int x = 35.35;//COMPILER ERROR  int x = (int) 35.35; //Explicit Casting |

Explicit casting would cause truncation of value if the value stored is greater than the size of the variable.

|  |
| --- |
| int bigValue = 280;  byte small = (byte) bigValue;  System.out.println(small); //output 24. Only 8 bits remain. |

Big Decimal

The BigDecimal class provides operations on double numbers for arithmetic, scale handling, rounding, comparison, format conversion and hashing. It can handle very large and very small floating point numbers with great precision but compensating with the time complexity a bit.

A BigDecimal consists of a random precision integer unscaled value and a 32-bit integer scale. If greater than or equal to zero, the scale is the number of digits to the right of the decimal point. If less than zero, the unscaled value of the number is multiplied by 10^(-scale).

double a = 0.02;

double b = 0.03;

double c = b - a;

System.out.println(c);

BigDecimal \_a = new BigDecimal("0.02");

BigDecimal \_b = new BigDecimal("0.03");

BigDecimal \_c = \_b.subtract(\_a);

System.out.println(\_c);

Program output:

0.009999999999999998

0.01

Q3. What is the advantage of BidDecimal over Double ? Core JavaAns. BigDecimal provides more precision as compared to double.

**Exception Handing**

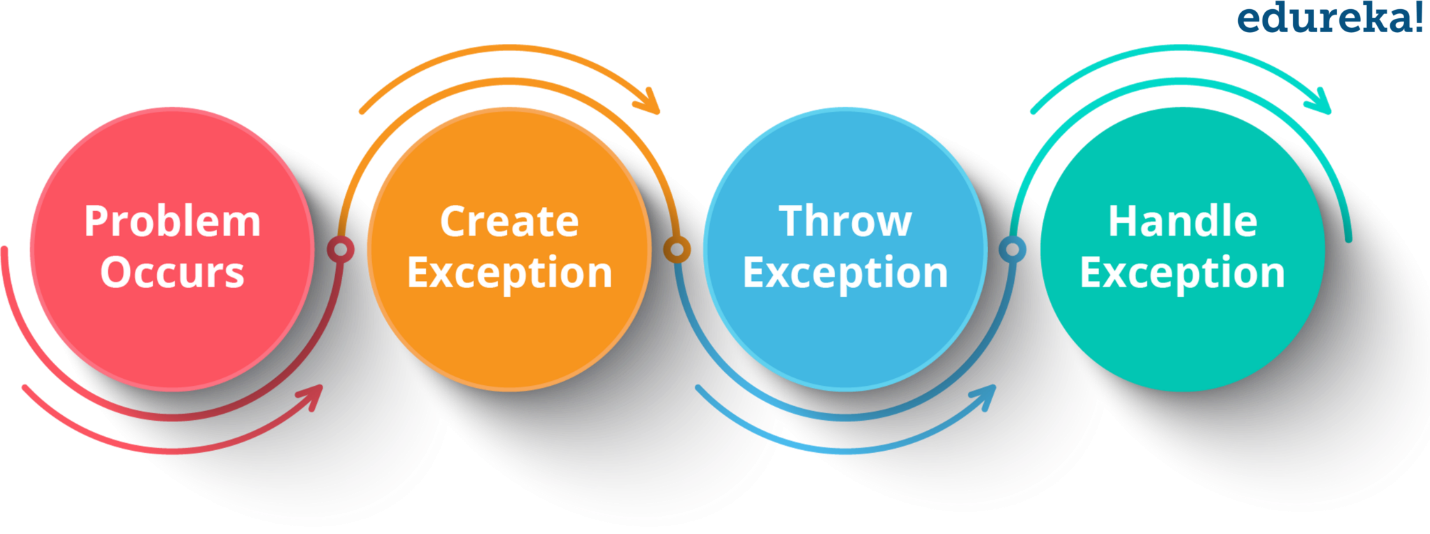
* 1. [Introduction to Exception Handling](https://www.edureka.co/blog/java-exception-handling#Introduction)
  2. [Exceptions Hierarchy](https://www.edureka.co/blog/java-exception-handling#Hierarchy)
  3. [Basic Exception Example](https://www.edureka.co/blog/java-exception-handling#BasicException)
  4. [Types of Exceptions](https://www.edureka.co/blog/java-exception-handling#ExceptionTypes)
  5. [Exception Handling Methods](https://www.edureka.co/blog/java-exception-handling#ExceptionMethods)
  6. [final vs finally vs finalize](https://www.edureka.co/blog/java-exception-handling#finalvsfinallyvsfinalize)
  7. [throw vs throws](https://www.edureka.co/blog/java-exception-handling#throwvsthrows)

## **Introduction to Exception Handling**

An exception is a problem that arises during the execution of a program. It can occur for various reasons say-

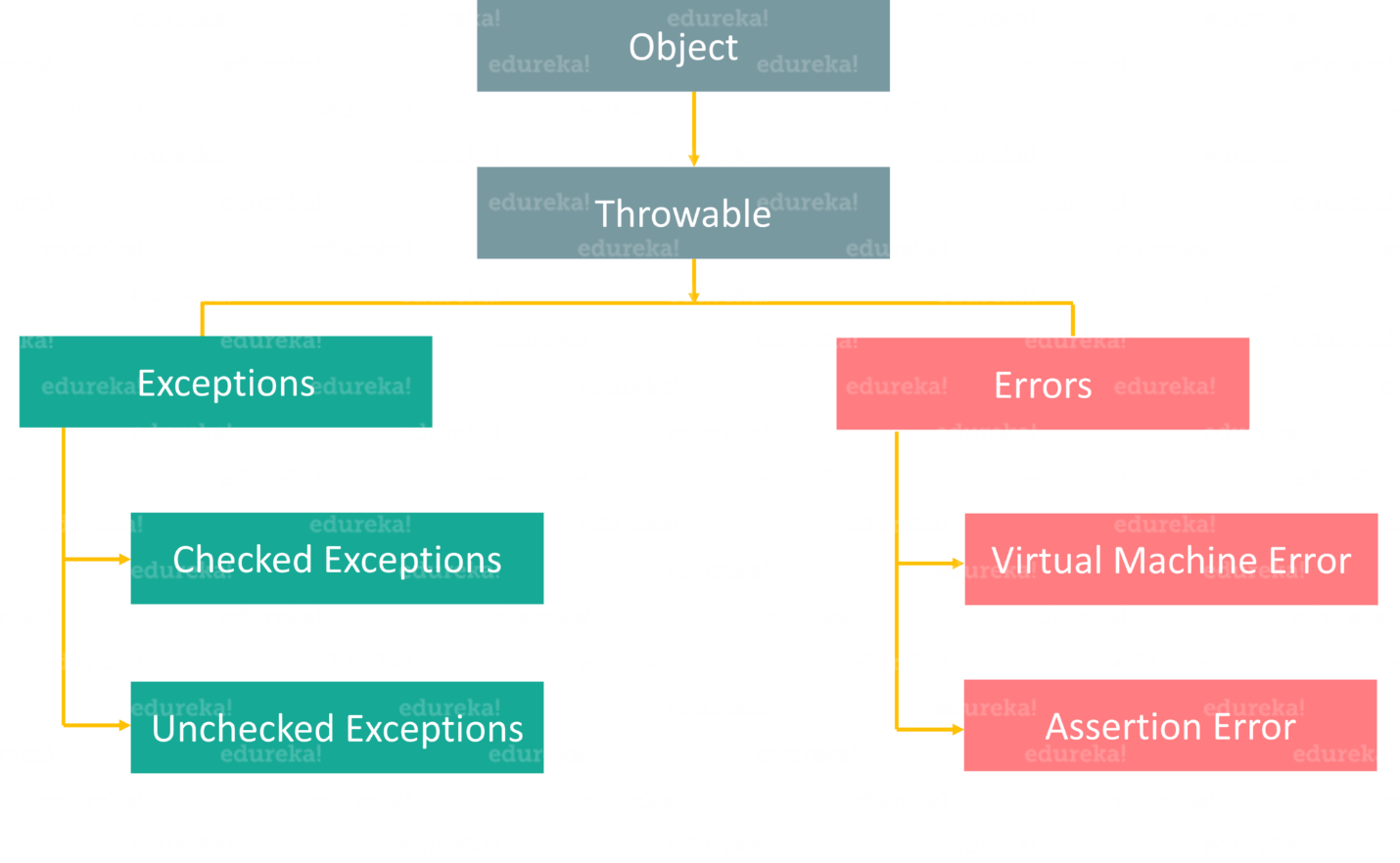
* A user has entered an invalid data
* File not found
* A network connection has been lost in the middle of communications
* The JVM has run out of a memory

Exception Handling mechanism follows a flow which is depicted in the below figure. But if an exception is not handled, it may lead to a system failure. That is why handling an exception is very important.



## **Exceptions Hierarchy**

All exception and error types are subclasses of class ***Throwable***, which is the base class of hierarchy. One branch is headed by ***Error*** which occurs at run-time and other by **Exception**that can happen either at compile time or run-time.



Basically,  an **Error**is used by the Java run-time system (JVM) to indicate errors that are associated with the run-time environment (JRE). StackOverflowError is an example of such an error. Whereas **Exception**is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.

Now that you know what errors and exceptions are, let’s find out the basic difference between them. Take a look at the below table which draws a clear line between both of them.

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| 1. Impossible to recover from an error | 1. Possible to recover from exceptions |
| 2. Errors are of type ‘unchecked’ | 2. Exceptions can be either ‘checked’ or ‘unchecked’ |
| 3. Occur at runtime | 3. Can occur at compile time or run time |
| 4. Caused by the application running environment | 4. Caused by the application itself |

Now, we will dive deeper into exceptions and see how they can be handled. First, let’s see the different types of exceptions.

* **Checked Exception**  
  It is an exception that occurs at compile time, also called compile time exceptions. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.
* **Unchecked Exception**  
  It is an exception that occurs at the time of execution. These are also called Runtime Exceptions. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to specify or catch the exceptions.

|  |  |
| --- | --- |
| **Built-in Exceptions** | **Description** |
| ***ArithmeticException*** | It is thrown when an exceptional condition has occurred in an arithmetic operation. |
| ***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. |
| ***ClassNotFoundException*** | This exception is raised when we try to access a class whose definition is not found. |
| ***FileNotFoundException*** | An exception that is raised when a file is not accessible or does not open. |
| ***IOException*** | It is thrown when an input-output operation is failed or interrupted. |
| ***InterruptedException*** | It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted. |
| ***NoSuchFieldException*** | It is thrown when a class does not contain the field (or variable) specified. |

### ****User-Defined Exceptions****

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, a user can also create exceptions which are called ‘User-Defined Exceptions’.  
**Key points to note:**

* 1. A user-defined exception must extend Exception class.
  2. The exception is thrown using throw keyword.

1. class MyException extends Exception{
2. String str1;
3. /\* Constructor of custom exception class
4. \* here I am copying the message that we are passing while
5. \* throwing the exception to a string and then displaying
6. \* that string along with the message.
7. \*/
8. MyException(String str2) {
9. str1=str2;
10. }
11. public String toString(){
12. return ("MyException Occurred: "+str1) ;
13. }
14. }
15. class Example1{
16. public static void main(String args[]){
17. try{
18. System.out.println("Starting of try block");
19. // I'm throwing the custom exception using throw
20. throw new MyException("This is My error Message");
21. }
22. catch(MyException exp){
23. System.out.println("Catch Block") ;
24. System.out.println(exp) ;
25. }
26. }
27. }

## **final vs finally vs finalize**

|  |  |  |
| --- | --- | --- |
| **final** | **finally** | **finalize** |
| It is a keyword. | It is a block. | It is a method. |
| Used to apply restrictions on class, methods & variables. | Used to place an important code. | Used to perform clean-up processing just before the object is garbage collected. |
| final class can’t be inherited, method can’t be overridden & the variable value can’t be changed. | It will be executed whether the exception is handled or not. | – |

Similarly, throw & throws sound alike, but they are different from each other. Let’s see how, with the help of the below table.

## **throw vs throws**

|  |  |
| --- | --- |
| **throw** | **throws** |
| 1. Used to explicitly throw an exception | 1. Used to declare an exception |
| 2. Checked exceptions cannot be propagated using throw only | 2. Checked exceptions can be propagated |
| 3. Followed by an instance | 3. Followed by a class |
| 4. Used within a method | 4. Used with a method signature |
| 5. Cannot throw multiple exceptions | 5. Can declare multiple exceptions |

## **Comparable v/s Comparator in Java**

|  |  |
| --- | --- |
| Comparable in Java | Comparator in Java |
| Comparable interface is used to sort the objects with natural ordering. | Comparator in Java is used to sort attributes of different objects. |
| Comparable interface compares “this” reference with the object specified. | Comparator in Java compares two different class objects provided. |
| Comparable is present in java.lang package. | A Comparator is present in the java.util package. |
| Comparable affects the original class, i.e., the actual class is modified. | Comparator doesn’t affect the original class |
| Comparable provides compareTo() method to sort elements. | Comparator provides compare() method, equals() method to sort elements. |

## **What is Comparable in Java?**

As the name itself suggests, **Comparable** is an [interface](https://www.edureka.co/blog/java-interface/) which defines a way to compare an object with other objects of the same type. It helps to sort the objects that have self-tendency to sort themselves, i.e., the objects must know how to order themselves. **Eg:** Roll number, age, salary. This interface is found in java.lang package and it contains only one method, i.e., compareTo(). Comparable is not capable of sorting the [objects](https://www.edureka.co/blog/java-tutorial/#obj) on its own, but the interface defines a method int compareTo() which is responsible for sorting.

Further, you must be thinking what is the compareTo method? Well, let me explain that to you!

## **What is the compareTo method and how it is used?**

This method is used to compare the given object with the current object. The **compareTo()** method returns an int value. The value can be either positive, negative, or zero. So now we are well acquainted with the theoretical knowledge of Comparable [interface in Java](https://www.edureka.co/blog/java-interface/) and **compareTo** method.

Let’s hop into understanding the implementation process. First, let’s see how to implement Comparable.

## **What is Comparator in Java?**

A Comparator interface is used to order the objects of a specific class. This interface is found in java.util package. It contains two methods;

* compare(Object obj1,Object obj2)
* equals(Object element).

The first method, compare(Object obj1,Object obj2)  compares its two input arguments and showcase the output. It returns a negative integer, zero, or a positive integer to state whether the first argument is less than, equal to, or greater than the second.

The second method, equals(Object element), requires an Object as a parameter and shows if the input object is equal to the comparator. The method will return true, only if the mentioned object is also a Comparator. The order remains the same as that of the Comparator.

After attaining brief learning about Comparator in Java, it’s time to move a step ahead. Let me show you an example depicting Comparator in Java.

ArrayList

What is the use of transient keyword?

The transient keyword in Java is used **to avoid serialization**. If any object of a data structure is defined as a transient , then it will not be serialized. Serialization is the ​process of converting an object into a byte stream.

private transient String password;

**@Transient**annotation is used to mark a field to be transient for the mapping framework, which means the field marked with [**@Transient**](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/annotation/Transient.html) is ignored by mapping framework and the field not mapped to any database column (in RDBMS) or Document property (in NOSQL). Thus the property will not be persisted to data store. @**Transient** exists in **org.springframework.data.annotation** package. The mapping framework will be different for each [Spring Data](https://spring.io/projects/spring-data) module (Spring Data Jdbc, Spring Data MongoDB, Spring Data JPA .. etc).

[ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) is a resizable array implementation in java. ArrayList grows dynamically and ensures that there is always a space to add elements. The backing data structure of ArrayList is an array of Object class. ArrayList class in Java has 3 constructors. It has its own version of readObject and writeObject methods. Object Array in ArrayList is [transient](https://www.geeksforgeeks.org/transient-keyword-java/). It implements RandomAccess, Cloneable, java.io.Serializable ( which are [Marker Interface in Java](https://www.geeksforgeeks.org/marker-interface-java/))

nternally an ArrayList uses an Object[] Array which is an array of objects. All operation like deleting, adding and updating the elements happens in this Object[] array.

/\*\*

\* The array buffer into which the elements of the ArrayList are stored.

\* The capacity of the ArrayList is the length of this array buffer. Any

\* empty ArrayList with elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA

\* will be expanded to DEFAULT\_CAPACITY when the first element is added.

\*/

transient Object[] elementData; // non-private to simplify nested class access

### ****Constructors****

To create an ArrayList, First need to create Object of ArrayList class.. ArrayList contains 3 types of constructors in Java 8

1. **ArrayList()**: This constructor is to initialize an empty List.
2. **ArrayList(int capacity):** This constructor we can pass capacity as a parameter, used to initialize the capacity by the user.
3. **ArrayList(Collection c):** In this constructor, we can pass a Collection c as a parameter, In which an Array list will contain the elements of Collection c.

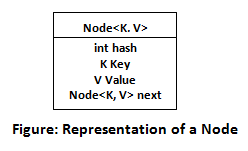
How hash map works

## **What is Hashing**

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

## **What is HashMap**

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.



**How to avoid ConcurrentModificationException?**

To avoid this exception,

* Simply we can do the modifications once the iteration is done, or
* Implement the concept of the [synchronized block or method](https://www.geeksforgeeks.org/method-block-synchronization-java/)

for (String str : myArrayList) {

if (someCondition) {

myArrayList.remove(str);

}

}

Iterator<String> iter = myArrayList.iterator();

while (iter.hasNext()) {

String str = iter.next();

if (someCondition)

iter.remove();}

What is singleton class

All implementations of the Singleton have these two steps in common:

* Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
* Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.



**public** **class** Singleton {

**private** **static** Singleton *singleton* = **new** Singleton( );

/\* A private Constructor prevents any other

\* class from instantiating.

\*/

**private** Singleton() { }

/\* Static 'instance' method \*/

**public** **static** Singleton getInstance( ) {

**return** *singleton*;

}

/\* Other methods protected by singleton-ness \*/

**protected** **static** **void** demoMethod( ) {

System.***out***.println("demoMethod for singleton");

}

}

// File Name: SingletonDemo.java

public class SingletonDemo {

public static void main(String[] args) {

Singleton tmp = Singleton.getInstance( );

tmp.demoMethod( );

}

}

## Practices in Java 8

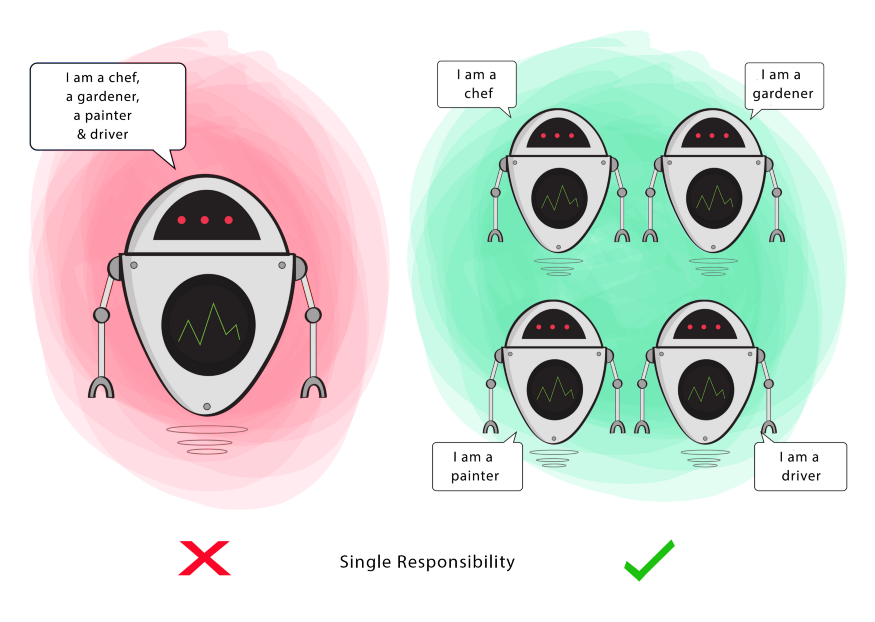
* **For Default methods -** use 1 default method per interface, and don't enhance functional interfaces. Instead, you'll focus on conservative implementations for those enhancements.
* **For Lambdas -** use expressions instead of statements, refactoring to use method references and chaining Lambdas.
* **For java.util.Optional -** use plain objects within fields and method parameters and optional for return values. Then, instead of get(), you'll want to use orElse()

The main rule of thumb for using default methods is not to abuse them and not to make the code messier than it would be without it. For example, if you want to add some sort of functionality to Java classes without polluting their hierarchy with a common superclass, consider creating a separate interface just for this one utility method. Here’s an example of an interface called **Debuggable**that uses the reflection API to get the access to the object’s fields and provides a decent t*oString()* implementation for the object that prints the fields values.

# **The SOLID Principles**

## **S — Single Responsibility**

A class should have a single responsibility



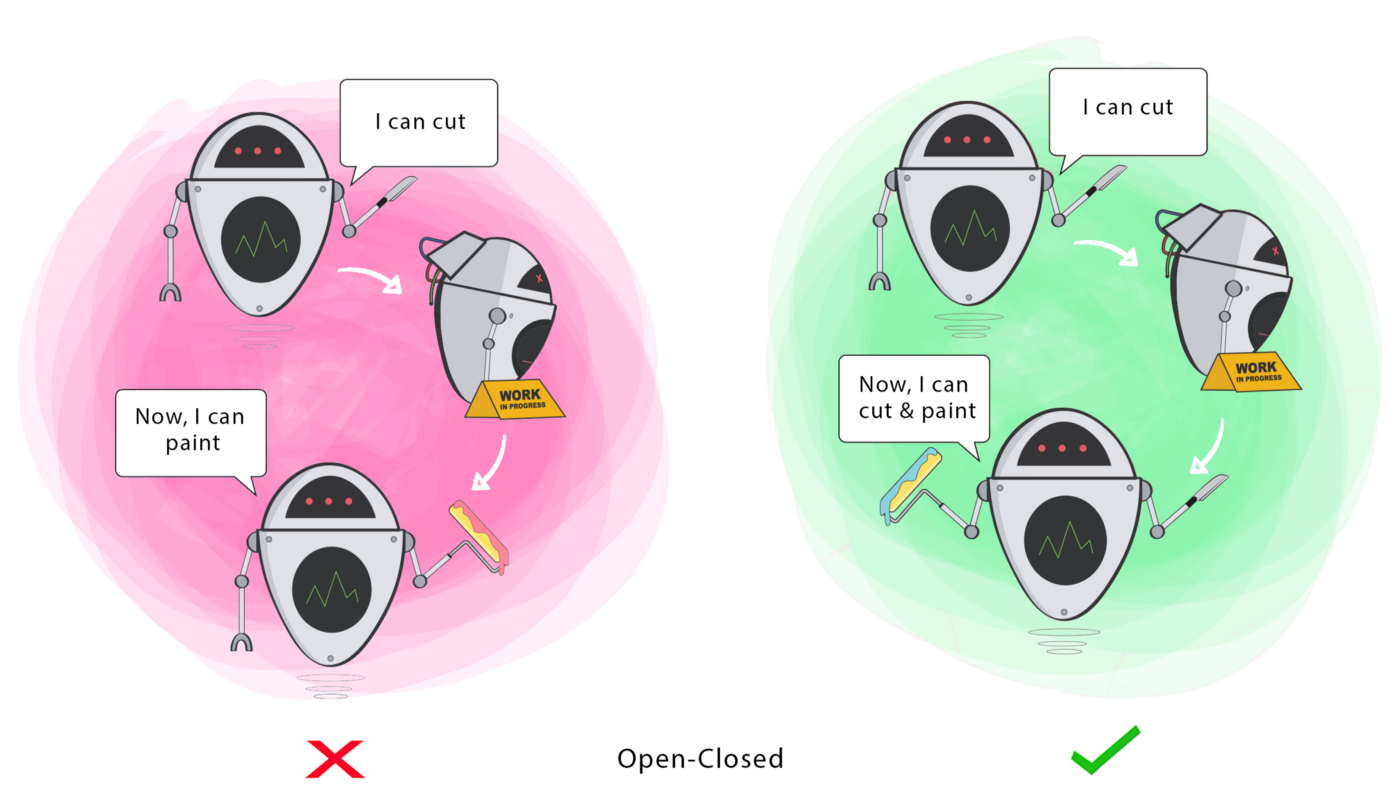
If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities, could affect the other ones without you knowing.

**Goal**

This principle aims to separate behaviours so that if bugs arise as a result of your change, it won’t affect other unrelated behaviours.

## **O — Open-Closed**

Classes should be open for extension, but closed for modification



Changing the current behaviour of a Class will affect all the systems using that Class.

If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.

**Goal**

This principle aims to extend a Class’s behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.

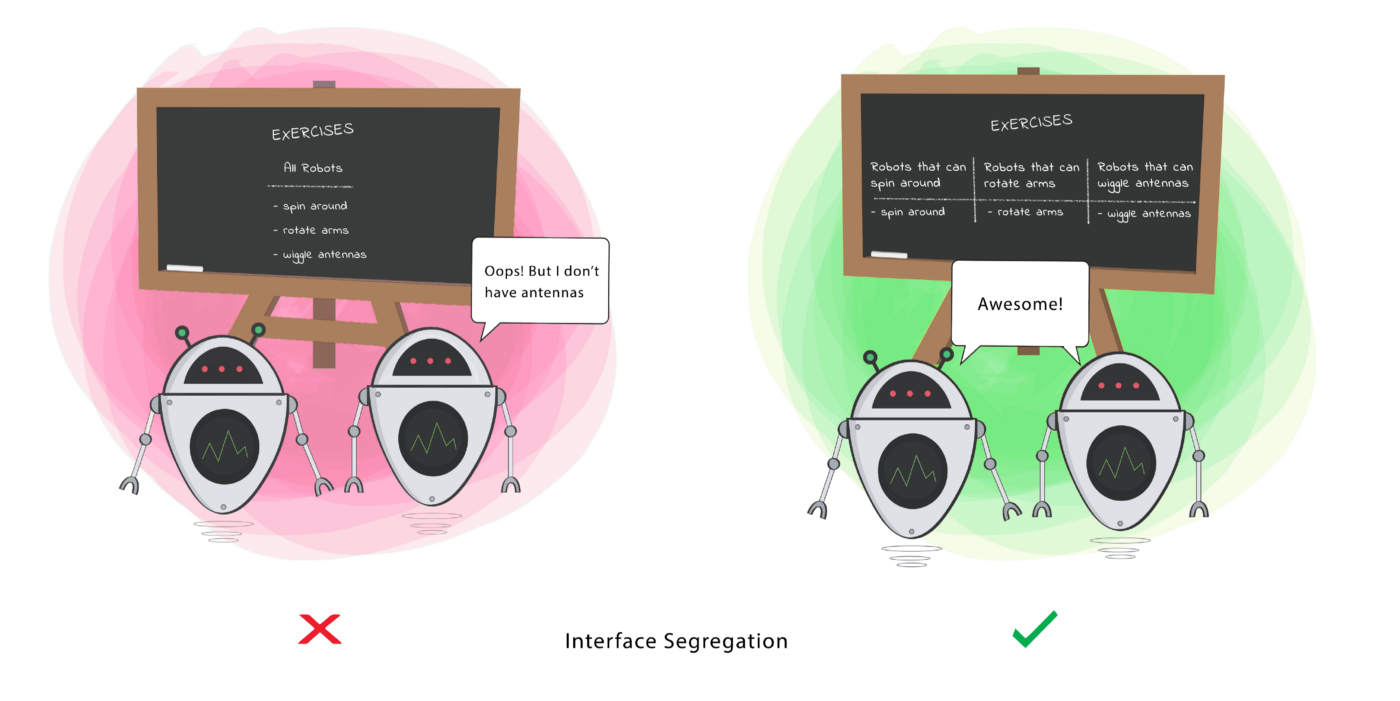
**L**— **Liskov Substitution**

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

**I**— **Interface Segregation**

Clients should not be forced to depend on methods that they do not use.



When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

A Class should perform only actions that are needed to fulfil its role. Any other action should be removed completely or moved somewhere else if it might be used by another Class in the future.

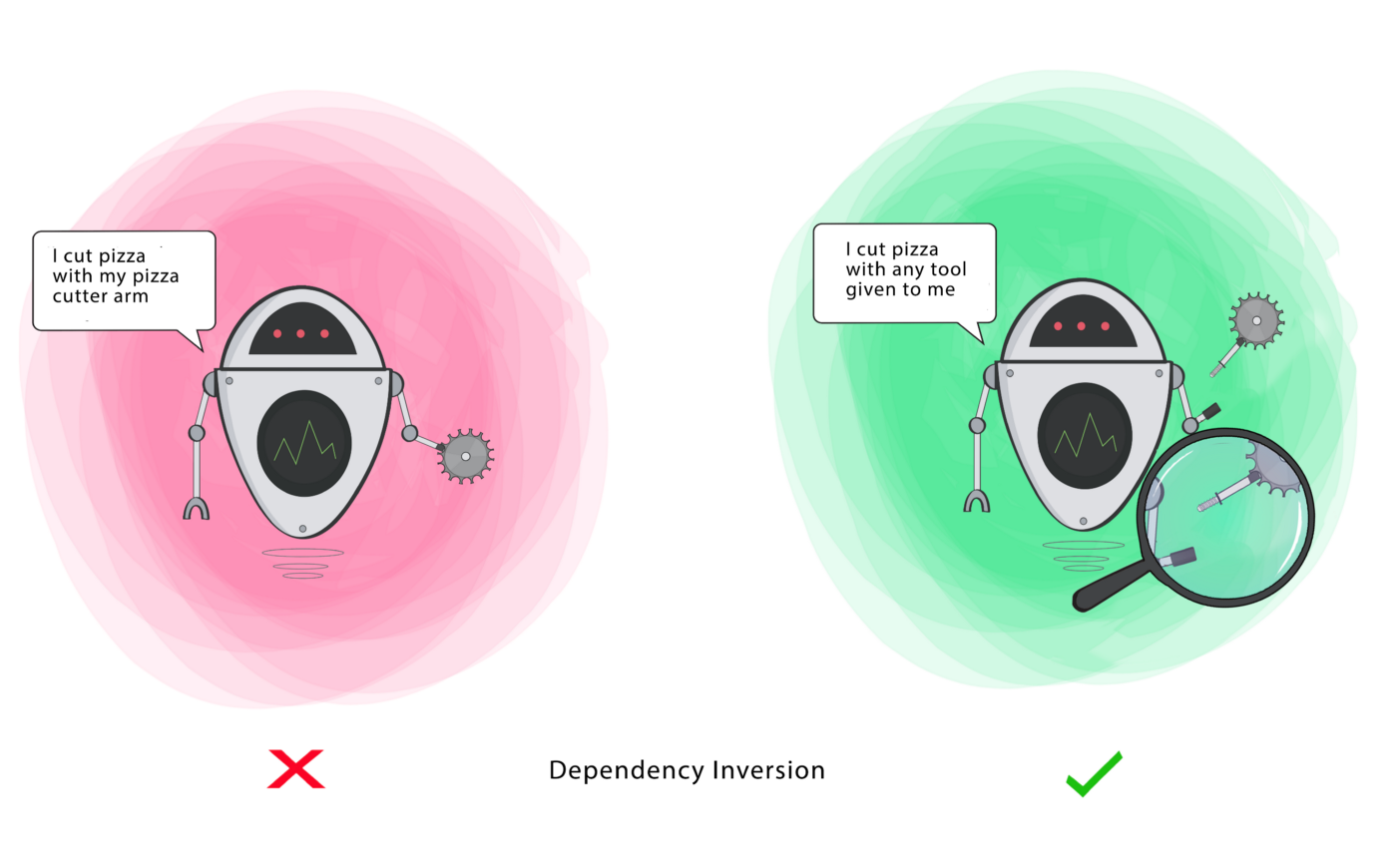
**Goal**

This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.

**D**— **Dependency Inversion**

- High-level modules should not depend on low-level modules. Both should depend on the abstraction.

- Abstractions should not depend on details. Details should depend on abstractions.



Firstly, let’s define the terms used here more simply

**High-level Module(or Class)**: Class that executes an action with a tool.

**Low-level Module (or Class)**: The tool that is needed to execute the action

**Abstraction**: Represents an interface that connects the two Classes.

**Details**: How the tool works

This principle says a Class should not be fused with the tool it uses to execute an action. Rather, it should be fused to the interface that will allow the tool to connect to the Class.

It also says that both the Class and the interface should not know how the tool works. However, the tool needs to meet the specification of the interface.

**Goal**

This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.

Spring

## @ControllerAdvice

Spring 3.2 brings support for**a global @ExceptionHandler with the @ControllerAdvice annotation.**

This enables a mechanism that breaks away from the older MVC model and makes use of ResponseEntity along with the type safety and flexibility of @ExceptionHandler:

@ControllerAdvice

**public** **class** **RestResponseEntityExceptionHandler**

**extends** **ResponseEntityExceptionHandler** {

@ExceptionHandler(value

= { IllegalArgumentException.class, IllegalStateException.class })

**protected** ResponseEntity<Object> **handleConflict**(

RuntimeException ex, WebRequest request) {

**String** bodyOfResponse = "This should be application specific";

**return** handleExceptionInternal(ex, bodyOfResponse,

**new** **HttpHeaders**(), HttpStatus.CONFLICT, request);

}

}

The@ControllerAdvice annotation allows us to **consolidate our multiple, scattered @ExceptionHandlers from before into a single, global error handling component.**

The actual mechanism is extremely simple but also very flexible:

* It gives us full control over the body of the response as well as the status code.
* It provides mapping of several exceptions to the same method, to be handled together.
* It makes good use of the newer RESTful ResposeEntity response.

One thing to keep in mind here is to **match the exceptions declared with @ExceptionHandler to the exception used as the argument of the method.**

If these don't match, the compiler will not complain — no reason it should — and Spring will not complain either.

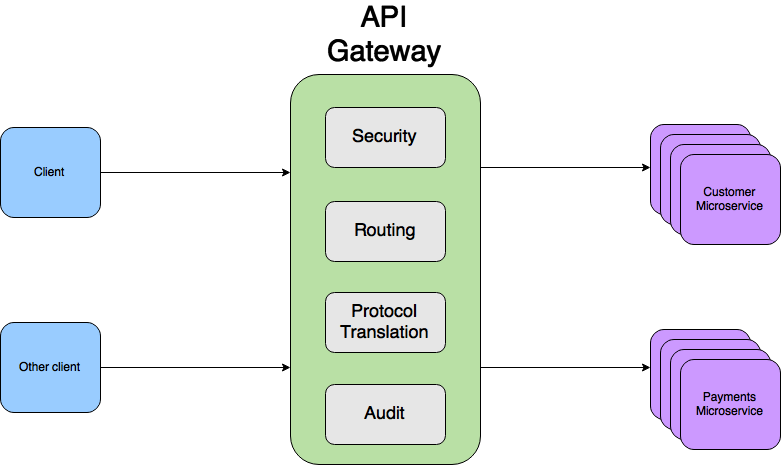
However, when the exception is actually thrown at runtime, **the exception resolving mechanism will fail with**:

Oauth 2.0 for MS to MS secure communication

## Spring boot OAuth2. OAuth 2 is an authorization method to… | by Bushra Saifi | Medium **OAuth 2**

OAuth 2.0 is **an authorization protocol** and NOT an authentication protocol. As such, it is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user's data. OAuth 2.0 uses Access Tokens.

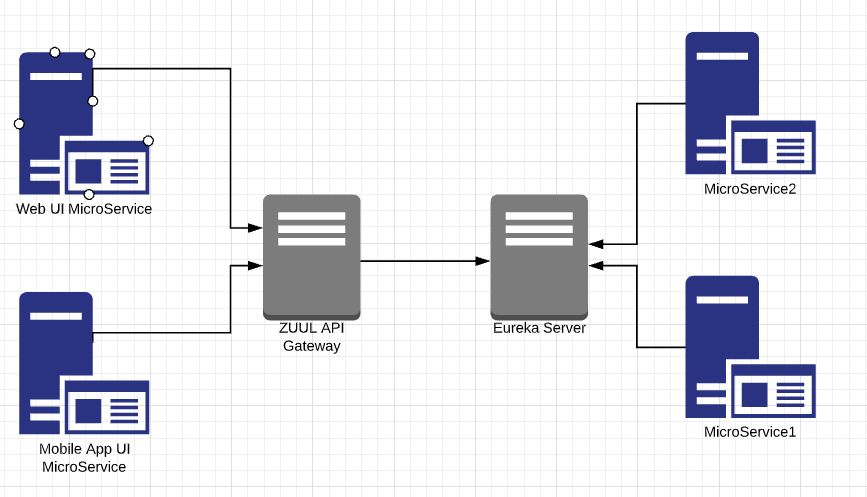
# **Zuul API Gateway**



## **Zuul API Gateway**

**Zuul** Server is an **API Gateway** application. It handles all the requests and performs the dynamic routing of microservice applications. It is also known as Edge Server. **Zuul** is built to enable dynamic **routing, monitoring, resiliency, and security**. It can also route the requests to multiple Amazon Auto Scaling Groups.

For Example, **/api/products** are mapped to the **product** service and **/api/user** is mapped to the **user** service. The Zuul Server dynamically routes the requests to the respective back-end application.



## **Zuul Components**

Zuul has mainly four types of filters that enable us to intercept the traffic in different timelines of the request processing for any particular transaction. We can add any number of filters for a particular**URL pattern**.

* **pre-filters** — are invoked before the request is routed
* **post-filers** — are invoked after the request has been routed
* **route-filters** — are used to route the request
* **error-filters** — are invoked when an error occurs while handling the request.

## **What is Eureka Server?**

**Eureka** Server is an application that holds information about all client-service applications. Every Microservice will register into the **Eureka** server and the **Eureka** server knows all the client applications running on each port and IP address. **Eureka** Server is also known as Discovery Server. With **Netflix Eureka** each client can simultaneously act as a server, to replicate its status to a connected peer. In other words, a client retrieves a list of all connected peers of a service registry and makes all further requests to any other services through a load-balancing algorithm.

Open shift

Red Hat OpenShift is an open source container application platform that runs on Red Hat Enterprise Linux CoreOS (RHCOS) and is built on top of Kubernetes. ... OpenShift includes everything you need for hybrid cloud, like **a container runtime, networking, monitoring, container registry, authentication, and authorization**.

**Image** − Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, Kubernetes only supports Docker images. Each container in a pod has its Docker image running inside it. When configuring a pod, the image property in the configuration file has the same syntax as the Docker command.

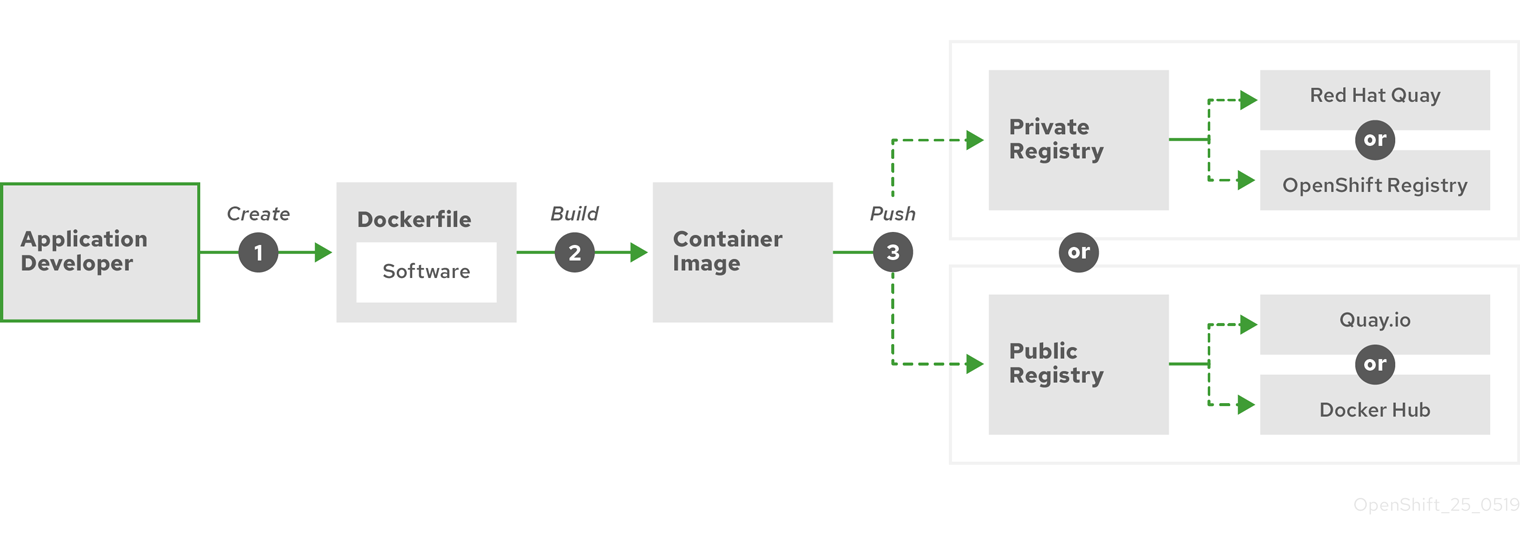
**Project** − They can be defined as the renamed version of the domain which was present in the earlier version of OpenShift V2.

**Container** − They are the ones which are created after the image is deployed on a Kubernetes cluster node.

**Node** − A node is a working machine in Kubernetes cluster, which is also known as minion for master. They are working units which can a physical, VM, or a cloud instance.

**Pod** − A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it. For example, keeping the database container and web server container inside the pod.

For deploying and managing Micro services



Spring Transactions

##### **What is database transaction?**

##### **What is an Application Transaction?**

##### **What is Spring Transaction?**

##### **What are types of Spring transaction management?**

##### **What is use of @transactional annotation in Spring?**

##### **How does transaction management work in Spring?**

##### **What is Transaction Propagation?**

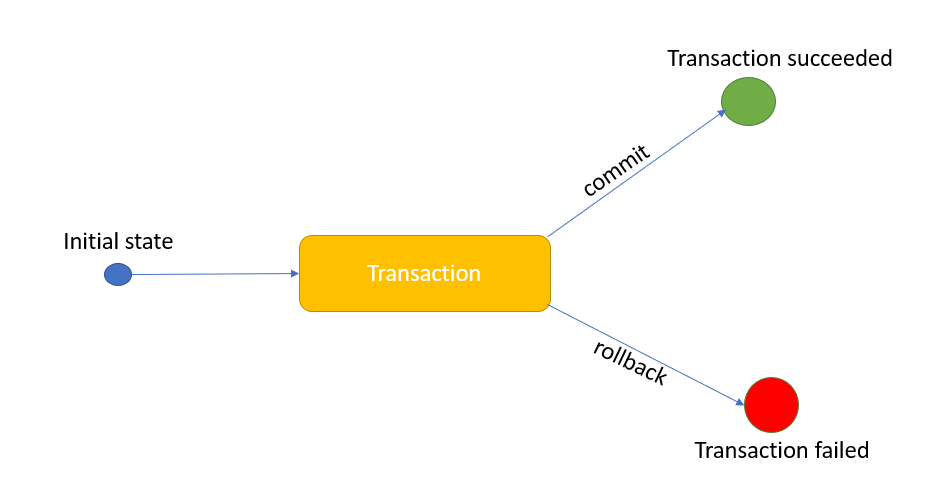
##### **What are different types of Transaction Propagation?**

*A database transaction is a sequence of actions that are treated as a single unit of work. Transaction management is an essential aspect of RDBMS application to ensure data integrity and consistency. The transaction can be defined with ACID properties.*

1. **Atomicity** - All success or none.
2. **Consistency** - Integrity constraints must be maintained in such a way that the database is consistent before and after the transaction.
3. **Isolation** - One transaction should not impact on another transaction.
4. **Durability** - After completion of the transaction, the changes and modifications to the database will be stored in and written to the disk and will continue even if a system failure occurs.

#### **What is Spring Transaction? Ans:**

*A database transaction is a sequence of actions that are considered as a single unit of work. These actions will either be completed in full or take no effect at all. Transaction management is a crucial part of the RDBMS based enterprise application to maintain data integrity and consistency.*



#### **What are different types of Transaction Propagation? Ans:**

There are six types of Transaction Propagation. ***REQUIRED*** is Default Transaction Propagation.  
To understand more, you can go through [Transaction Propagation example](https://www.techgeeknext.com/spring-boot/spring-boot-transaction-propagation)

* REQUIRED
* SUPPORTS
* NOT\_SUPPORTED
* REQUIRES\_NEW
* NEVER
* MANDATOR

@RestTemaplate

For MS to MS calling

// HttpHeaders

HttpHeaders headers = **new** HttpHeaders();

headers.setAccept(Arrays.asList(**new** MediaType[] { MediaType.APPLICATION\_JSON }));

// Request to return JSON format

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.set("my\_other\_key", "my\_other\_value");

// HttpEntity<String>: To get result as String.

HttpEntity<String> entity = **new** HttpEntity<String>(headers);

// RestTemplate

RestTemplate restTemplate = **new** RestTemplate();

// Send request with GET method, and Headers.

ResponseEntity<String> response = restTemplate.exchange(URL\_EMPLOYEES, //

HttpMethod.GET, entity, String.**class**);

String result = response.getBody();

System.out.println(result);

ResponseEntity is meant to represent the entire HTTP response. You can control anything that goes into it: status code, headers, and body.

Using Spring ResponseEntity to Manipulate the HTTP Response

@GetMapping("/hello") ResponseEntity<String> **hello**() { **return** ResponseEntity.ok("Hello World!"); }

A profile is a set of configuration settings. Spring Boot allows to define profile specific property files in the form of application-{profile}.properties. It automatically loads the properties in an application.properties file for all profiles, and the ones in profile-specific property files only for the specified profile. The keys in the profile-specific property override the ones in the master property file.

pom.xml

src

├── main

│   ├── java

│   │   └── com

│   │   └── zetcode

│   │   └── Application.java

│   └── resources

│   ├── application-dev.properties

│   ├── application-prod.properties

│   └── application.properties

└── test

└── java

**resources/application.properties**

spring.profiles.active=local

**Using the maven command:**

$mvn spring-boot:run -Dspring-boot.run.arguments="--spring.profiles.active=production"

**JVM command:**

$java -jar target/app.jar – spring.profiles.active=production

It is also possible to pass multiple arguments at the same time. Using the example above we will pass one more property, server port as shown below.

**Maven command (space separated):**

$mvn spring-boot:run -Dspring-boot.run.arguments="--spring.profiles.active=production – server.port=8089"

**Using JVM command:**

$java -jar target/app.jar – spring.profiles.active=production – server.port=8089

|  |
| --- |
| What are the different types of dependency injections in spring? |
| |  | | --- | | **Answer:** | | Spring supports 2 types of dependency injection, they are:  1) Constructor-based dependency injection: It is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.  2) Setter-based dependency injection: It is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. | |

|  |
| --- |
|  |
| Can Enum extend any class in Java? |
| |  | | --- | | **Answer:** | | Enum can not extend any class in java, the reason is by default, Enum extends abstract base class java.lang.Enum. Since java does not support multiple inheritance for classes, Enum can not extend another class. | |

|  |
| --- |
| What is Dependency Injection? |
| |  | | --- | | **Answer:** | | Any application is composed of many objects that collaborate with each other to perform some useful stuff. Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.  For example, consider a Car object. A Car depends on Wheels, Engine, Fuel, Battery, etc to run. Traditionally we define the brand of such dependent objects along with the definition of the Car object.   |  | | --- | | [?](https://www.java2novice.com/java_interview_questions/dependency-injection/) |   Here, the Car object is responsible for creating the dependent objects.  What if we want to change the type of its dependent object - say Wheel - after the initial NepaliRubberWheel() punctures? We need to recreate the Car object with its new dependency say ChineseRubberWheel(), but only the Car manufacturer can do that.  Then what the Dependency Injection do us for ...  When using Dependency Injection, objects are given their dependencies at run time rather than compile time (car manufacturing time). So that we can now change the Wheel whenever we want. Here, the Dependency (Wheel) can be injected into Car at run time.  Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control. | |

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Car{    private Wheel wh= new NepaliRubberWheel();    private Battery bt= new ExcideBattery();    //rest  } |

|  |
| --- |
| Difference between ConcurrentHashMap and Collections.synchronizedMap(Map)? |
| |  | | --- | | **Answer:** | | The "scalability issues" for Hashtable are present in exactly the same way in Collections.synchronizedMap(Map) - they use very simple synchronization, which means that only one thread can access the map at the same time.  This is not much of an issue when you have simple inserts and lookups (unless you do it extremely intensively), but becomes a big problem when you need to iterate over the entire Map, which can take a long time for a large Map - while one thread does that, all others have to wait if they want to insert or lookup anything.  The ConcurrentHashMap uses very sophisticated techniques to reduce the need for synchronization and allow parallel read access by multiple threads without synchronization and, more importantly, provides an Iterator that requires no synchronization and even allows the Map to be modified during interation (though it makes no guarantees whether or not elements that were inserted during iteration will be returned).  Here are the differences:   |  |  |  |  | | --- | --- | --- | --- | | **Property** | **HashMap** | **Hashtable** | **ConcurrentHashMap** | | Null as a key | Allowed | Not Allowed | Not Allowed | | Is thread-safe | No | Yes | Yes | | Lock mechanism | N/A | Locks the whole map | Locks the portion | | Iterator | Fail-fast | Fail-fast | Fail-safe | | |
|  |
| What is Spring IoC container? |
| |  | | --- | | **Answer:** | | Inversion of Control (IoC) is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.  The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:   * To instantiate the application class. * To configure the object. * To assemble the dependencies between the objects.   There are two types of IoC containers. They are:   1. BeanFactory 2. ApplicationContext | |

|  |
| --- |
| Difference between map and flatMap methods in Java 8 |
| |  | | --- | | **Answer:** | | Both map and flatMap can be applied to a Stream<T> and they both return a Stream<R>. The difference is that the map operation produces one output value for each input value, whereas the flatMap operation produces an arbitrary number (zero or more) values for each input value. This is reflected in the arguments to each operation.  The map operation takes a Function, which is called for each value in the input stream and produces one result value, which is sent to the output stream.  The flatMap operation takes a function that conceptually wants to consume one value and produce an arbitrary number of values. However, in Java, it's cumbersome for a method to return an arbitrary number of values, since methods can return only zero or one value. One could imagine an API where the mapper function for flatMap takes a value and returns an array or a List of values, which are then sent to the output. Given that this is the streams library, a particularly apt way to represent an arbitrary number of return values is for the mapper function itself to return a stream! The values from the stream returned by the mapper are drained from the stream and are passed to the output stream. The "clumps" of values returned by each call to the mapper function are not distinguished at all in the output stream, thus the output is said to have been "flattened." | |

|  |
| --- |
| Difference between constructor injection and setter injection in Spring. |
| |  | | --- | | **Answer:** | | We need the assurance from the Inversion of control (IoC) container that, before using any bean, the injection of necessary beans must be done.  In ***setter injection*** strategy, we trust the Inversion of control (IoC) container that it will first create the bean first but will do the injection right before using the bean using the setter methods. And the injection is done according to your configuration. If you somehow misses to specify any beans to inject in the configuration, the injection will not be done for those beans and your dependent bean will not function accordingly when it will be in use!  But in ***constructor injection*** strategy, container imposes (or must impose) to provide the dependencies properly while constructing the bean. This was addressed as "container-agnostic manner", as we are required to provide dependencies while creating the bean, thus making the visibility of dependency, independent of any IoC container. | |

|  |
| --- |
| How Java-8 Streams differ from collections? |
| |  | | --- | | **Answer:** | | First of all, please note that *"Streams are not collections"*. java.util.stream is introduced to process elements in sequence. Streams are wrappers for collections and arrays. They wrap an existing collection to support operations expressed with lambdas, so you specify what you want to do, not how to do it. Also don't get confused with InputStream, java.util.stream does not have any relationship with InputStream or OutputStreams. java.util.stream are part of functional programming.  Streams differ from collection framework in several ways:   * ***No storage.*** A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations. * ***Functional in nature.*** An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection. * ***Laziness-seeking.*** Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy. * ***Possibly unbounded.*** While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time. * ***Consumable.*** The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source. | |

|  |
| --- |
|  |
| What are the different types of bean scope in Spring framework? |
| |  | | --- | | **Answer:** | | In the spring bean configurations, bean attribute called 'scope' defines what kind of object has to created and returned. There are 5 types of bean scopes available, they are:  **1) singleton:** Returns a single bean instance per Spring IoC container.  **2) prototype:** Returns a new bean instance each time when requested.  **3) request:** Returns a single instance for every HTTP request call.  **4) session:** Returns a single instance for every HTTP session.  **5) global session:** global session scope is equal as session scope on portlet-based web applications.  If no bean scope is specified in bean configuration file, then it will be by default 'singleton'. | |

|  |
| --- |
| What are the key components of Spring Boot framework? |
| |  | | --- | | **Answer:** | | Spring Boot Framework has mainly four major components.  **Spring Boot Starters:** The main responsibility of Spring Boot Starter is to combine a group of common or related dependencies into single dependencies. Spring Boot starters can help to reduce the number of manually added dependencies just by adding one dependency. So instead of manually specifying the dependencies just add one starter. Examples are spring-boot-starter-web, spring-boot-starter-test, spring-boot-starter-data-jpa, etc.  **Spring Boot AutoConfigurator:** One of the common complaint with Spring is, we need to make lot of XML based configurations. Spring Boot AutoConfigurator will simplify all these XML based configurations. It also reduces the number of annotations.  **Spring Boot CLI:** Spring Boot CLI(Command Line Interface) is a Spring Boot software to run and test Spring Boot applications from command prompt. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application. | |

|  |
| --- |
| What are the advantages and disadvantages of Spring Boot? |
| |  | | --- | | **Answer:** | | This answer is based on my personal opinion, it may vary from person to person: Spring Boot Advantages  1. Simplified & version conflict free dependency management through the starter POMs. 2. We can quickly setup and run standalone, web applications and micro services at very less time. 3. You can just assemble the jar artifact which comes with an embedded Tomact, Jetty or Undertow application server and you are ready to go. 4. Spring Boot provides HTTP endpoints to access application internals like detailed metrics, application inner working, health status, etc. 5. No XML based configurations at all. Very much simplified properties. The beans are initialized, configured and wired automatically. 6. The Spring Initializer provides a project generator to make you productive with the certain technology stack from the beginning. You can create a skeleton project with web, data access (relational and NoSQL datastores), cloud, or messaging support.  Spring Boot Disadvantages  1. Spring boot may unnecessarily increase the deployment binary size with unused dependencies. 2. If you are a control freak, I doubt Spring Boot would fit your needs. 3. Spring Boot sticks good with micro services. The Spring Boot artifacts can be deployed directly into Docker containers. In a large and monolithic based applications, I would not encourage you to use Spring Boot. | |