1. **New Features in java 8:**

- forEach() method in Iterable interface:

forEach Method in java.lang.Iterable Interface.

Whenever we need to traverse over a collection we have to create an Iterator to iterate over the collection and then we can have our business logic inside a loop for each of the elements inside the collection. We may greeted with ConcurrentModificationException if it is not implemented properly.

The implementation of forEach method in Iterable interface is:

default void forEach(Consumer action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

Parameter: This method takes a parameter action of type java.util.function.Consumer which represents the action to be performed for each element.

Returns: The return type of forEach is void. Hence it do not returns anything.

Exception: Throws NullPointerException if the input action is null.

Program 1: Program to iterate a list of String using the Iterator.

// Java program to demonstrate

// forEach() method of Iterable interface

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.function.Consumer;

public class ForEachExample {

public static void main(String[] args)

{

List<String> data = new ArrayList<>();

data.add("New Delhi");

data.add("New York");

data.add("Mumbai");

data.add("London");

Iterator<String> itr = data.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

// data.remove(itr.next());

// this line can introduce you to

// java.util.ConcurrentModificationException.

}

}

}

Output:

New Delhi

New York

Mumbai

London

Program 2: Program to demonstrate forEach() method on a List which contains list of cities.

// Java program to demonstrate

// forEach() method of Iterable interface

import java.util.ArrayList;

import java.util.List;

import java.util.function.Consumer;

public class ForEachExample {

public static void main(String[] args)

{

List<String> data = new ArrayList<>();

data.add("New Delhi");

data.add("New York");

data.add("Mumbai");

data.add("London");

data.forEach(new Consumer<String>() {

@Override

public void accept(String t)

{

System.out.println(t);

}

});

}

}

Output:

New Delhi

New York

Mumbai

London

In the above program, we have created a List of String with 4 elements and then we have iterated over the list using the forEach method. As described earlier forEach method take Consumer object as input, we have created an anonymous inner class implementation of Consumer interface and overrides the accept method. In this example, we have kept the business logic inside the anonymous inner class and we can not reuse it.

Program 3: In this program we will demonstrate the implementation of Consumer interface separately so that we can reuse it. Let’s create a class CityConsumer which implements Consumer interface and overrides its accept method.

// Java program to demonstrate

// forEach() method of Iterable interface

import java.util.\*;

import java.util.function.Consumer;

class CityConsumer implements Consumer<String> {

@Override

public void accept(String t)

{

System.out.println(t);

}

}

// Now we can use the CityConsumer

// with forEach method by just creating

// an object of CityConsumer class as below.

public class ForEachExample {

public static void main(String[] args)

{

List<String> data = new ArrayList<>();

data.add("New Delhi");

data.add("New York");

data.add("Mumbai");

data.add("London");

// create an object of CityConsumer

// and pass it to forEach method

CityConsumer cityConsumer = new CityConsumer();

data.forEach(cityConsumer);

}

}

Output:

New Delhi

New York

Mumbai

London

- default and static methods in Interfaces:

Like regular interface methods, default methods are implicitly public — there's no need to specify the public modifier.

Unlike regular interface methods, they are declared with the default keyword at the beginning of the method signature, and they provide an implementation.

Let's see a simple example:

public interface MyInterface {

// regular interface methods

default void defaultMethod() {

// default method implementation

}

}

The reason why default methods were included in the Java 8 release is pretty obvious.

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.

Default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Thus, there's no need to modify the implementing classes.

In this way, backward compatibility is neatly preserved without having to refactor the implementers.

1. Default Interface Methods in Action

To better understand the functionality of default interface methods, let's create a simple example.

Say that we have a naive Vehicle interface and just one implementation. There could be more, but let's keep it that simple:

public interface Vehicle {

String getBrand();

String speedUp();

String slowDown();

default String turnAlarmOn() {

return "Turning the vehicle alarm on.";

}

default String turnAlarmOff() {

return "Turning the vehicle alarm off.";

}

}

And let's write the implementing class:

public class Car implements Vehicle {

private String brand;

// constructors/getters

@Override

public String getBrand() {

return brand;

}

@Override

public String speedUp() {

return "The car is speeding up.";

}

@Override

public String slowDown() {

return "The car is slowing down.";

}

}

Lastly, let's define a typical main class, which creates an instance of Car and calls its methods:

public static void main(String[] args) {

Vehicle car = new Car("BMW");

System.out.println(car.getBrand());

System.out.println(car.speedUp());

System.out.println(car.slowDown());

System.out.println(car.turnAlarmOn());

System.out.println(car.turnAlarmOff());

}

Please notice how the default methods turnAlarmOn() and turnAlarmOff() from our Vehicle interface are automatically available in the Car class.

Furthermore, if at some point we decide to add more default methods to the Vehicle interface, the application will still continue working, and we won't have to force the class to provide implementations for the new methods.

The most typical use of default methods in interfaces is to incrementally provide additional functionality to a given type without breaking down the implementing classes.

In addition, they can be used to provide additional functionality around an existing abstract method:

public interface Vehicle {

// additional interface methods

double getSpeed();

default double getSpeedInKMH(double speed) {

// conversion

}

}

1. Multiple Interface Inheritance Rules

Default interface methods are a pretty nice feature indeed, but with some caveats worth mentioning. Since Java allows classes to implement multiple interfaces, it's important to know what happens when a class implements several interfaces that define the same default methods.

To better understand this scenario, let's define a new Alarm interface and refactor the Car class:

public interface Alarm {

default String turnAlarmOn() {

return "Turning the alarm on.";

}

default String turnAlarmOff() {

return "Turning the alarm off.";

}

}

With this new interface defining its own set of default methods, the Car class would implement both Vehicle and Alarm:

public class Car implements Vehicle, Alarm {

// ...

}

In this case, the code simply won't compile, as there's a conflict caused by multiple interface inheritance (a.k.a the Diamond Problem). The Car class would inherit both sets of default methods. Which ones should be called then?

To solve this ambiguity, we must explicitly provide an implementation for the methods:

@Override

public String turnAlarmOn() {

// custom implementation

}

@Override

public String turnAlarmOff() {

// custom implementation

}

We can also have our class use the default methods of one of the interfaces.

Let's see an example that uses the default methods from the Vehicle interface:

@Override

public String turnAlarmOn() {

return Vehicle.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

return Vehicle.super.turnAlarmOff();

}

Similarly, we can have the class use the default methods defined within the Alarm interface:

@Override

public String turnAlarmOn() {

return Alarm.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

return Alarm.super.turnAlarmOff();

}

Furthermore, it's even possible to make the Car class use both sets of default methods:

@Override

public String turnAlarmOn() {

return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();

}

@Override

public String turnAlarmOff() {

return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff();

}

1. static Interface Methods

Aside from being able to declare default methods in interfaces, Java 8 allows us to define and implement static methods in interfaces.

Since static methods don't belong to a particular object, they are not part of the API of the classes implementing the interface, and they have to be called by using the interface name preceding the method name.

To understand how static methods work in interfaces, let's refactor the Vehicle interface and add to it a static utility method:

public interface Vehicle {

// regular / default interface methods

static int getHorsePower(int rpm, int torque) {

return (rpm \* torque) / 5252;

}

}

Defining a static method within an interface is identical to defining one in a class. Moreover, a static method can be invoked within other static and default methods.

Now, say that we want to calculate the horsepower of a given vehicle's engine. We just call the getHorsePower() method:

Vehicle.getHorsePower(2500, 480));

The idea behind static interface methods is to provide a simple mechanism that allows us to increase the degree of cohesion of a design by putting together related methods in one single place without having to create an object.

Pretty much the same can be done with abstract classes. The main difference lies in the fact that abstract classes can have constructors, state, and behavior.

Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

- Functional Interfaces and Lambda Expressions:

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

// Java program to demonstrate functional interface

class Test

{

public static void main(String args[])

{

// create anonymous inner class object

new Thread(new Runnable()

{

@Override

public void run()

{

System.out.println("New thread created");

}

}).start();

}

}

Output:

New thread created

Java 8 onwards, we can assign lambda expression to its functional interface object like this:

// Java program to demonstrate Implementation of

// functional interface using lambda expressions

class Test

{

public static void main(String args[])

{

// lambda expression to create the object

new Thread(()->

{System.out.println("New thread created");}).start();

}

}

New thread created

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.

// Java program to demonstrate lamda expressions to implement

// a user defined functional interface.

@FunctionalInterface

interface Square

{

int calculate(int x);

}

class Test

{

public static void main(String args[])

{

int a = 5;

// lambda expression to define the calculate method

Square s = (int x)->x\*x;

// parameter passed and return type must be

// same as defined in the prototype

int ans = s.calculate(a);

System.out.println(ans);

}

}

Output:

25

java.util.function Package:

The java.util.function package in Java 8 contains many builtin functional interfaces like-

Predicate: The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is

public interface Predicate

{

public boolean test(T t);

}

BinaryOperator: The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is

public interface BinaryOperator

{

public T apply(T x, T y);

}

Function: The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is

public interface Function

{

public R apply(T t);

}

// A simple program to demonstrate the use

// of predicate interface

import java.util.\*;

import java.util.function.Predicate;

class Test

{

public static void main(String args[])

{

// create a list of strings

List<String> names =

Arrays.asList("Geek","GeeksQuiz","g1","QA","Geek2");

// declare the predicate type as string and use

// lambda expression to create object

Predicate<String> p = (s)->s.startsWith("G");

// Iterate through the list

for (String st:names)

{

// call the test method

if (p.test(st))

System.out.println(st);

}

}

}

Output:

Geek

GeeksQuiz

Geek2

Important Points/Observations:

A functional interface has only one abstract method but it can have multiple default methods.

@FunctionalInterface annotation is used to ensure an interface can’t have more than one abstract method. The use of this annotation is optional.

The java.util.function package contains many builtin functional interfaces in Java 8.

- Streams In Java:

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-

Intermediate Operations:

map: The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

List number = Arrays.asList(2,3,4,5);

List square = number.stream().map(x->x\*x).collect(Collectors.toList());

filter: The filter method is used to select elements as per the Predicate passed as argument.

List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

sorted: The sorted method is used to sort the stream.

List names = Arrays.asList("Reflection","Collection","Stream");

List result = names.stream().sorted().collect(Collectors.toList());

Terminal Operations:

collect: The collect method is used to return the result of the intermediate operations performed on the stream.

List number = Arrays.asList(2,3,4,5,3);

Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());

forEach: The forEach method is used to iterate through every element of the stream.

List number = Arrays.asList(2,3,4,5);

number.stream().map(x->x\*x).forEach(y->System.out.println(y));

reduce: The reduce method is used to reduce the elements of a stream to a single value.

The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);

int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

Program to demonstrate the use of Stream

//a simple program to demonstrate the use of stream in java

import java.util.\*;

import java.util.stream.\*;

class Demo

{

public static void main(String args[])

{

// create a list of integers

List<Integer> number = Arrays.asList(2,3,4,5);

// demonstration of map method

List<Integer> square = number.stream().map(x -> x\*x).

collect(Collectors.toList());

System.out.println(square);

// create a list of String

List<String> names =

Arrays.asList("Reflection","Collection","Stream");

// demonstration of filter method

List<String> result = names.stream().filter(s->s.startsWith("S")).

collect(Collectors.toList());

System.out.println(result);

// demonstration of sorted method

List<String> show =

names.stream().sorted().collect(Collectors.toList());

System.out.println(show);

// create a list of integers

List<Integer> numbers = Arrays.asList(2,3,4,5,2);

// collect method returns a set

Set<Integer> squareSet =

numbers.stream().map(x->x\*x).collect(Collectors.toSet());

System.out.println(squareSet);

// demonstration of forEach method

number.stream().map(x->x\*x).forEach(y->System.out.println(y));

// demonstration of reduce method

int even =

number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

System.out.println(even);

}

}

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

Important Points/Observations:

A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.

Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

- Java Time API:

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API.

Not thread safe − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.

Poor design − Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.

Difficult time zone handling − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.

Local − Simplified date-time API with no complexity of timezone handling.

Zoned − Specialized date-time API to deal with various timezones.

Local Date-Time API

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required. Let's see them in action.

- Collection API improvements:

A lot of internal architecture/performance changes been happend but I am just going to discuss about the useful methods been added.

Sorting Map directly with Comparators.

As we know Map is in order, it is a lot of struggle to get it sorted. Now Map interface added default methods which gives you comparators for different styles like comparingByKey, comparingByValue.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.put("Z", "z");

List<Map.Entry<String, String>> sortedByKey = map.entrySet().stream().sorted(Map.Entry.comparingByKey())

.collect(Collectors.toList());

sortedByKey.forEach(System.out::println);

output :

B=b

C=c

Z=z

Iterate over map easily with forEach.

If you observe the above example code, while printing I used the method forEach method. This is very revealing feature so far in Map. We all know how ugly is the old fashioned way of iterating and finally implementers added a default method forEach. Now it is super easy to iterate over map just like List.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.put("Z", "z");

map.forEach((k, v) -> System.out.println("Key : " + k + " Value : " + v));

output :

Key : B Value : b

Key : C Value : c

Key : Z Value : z

Get rid off ugly if-else condition, use getOrDefault method.

Legacy code for checking containsKey got moved to default method getOrDefault. This method returns the value to which the specified key is mapped, otherwise returns the given defaultValue if this map contains no mapping for the key.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

String val = map.getOrDefault("B", "Nah!");

System.out.println(val); // prints Nah!

Replace and Remove utilities.

New utility default methods have been added now. replaceAll Can replace all the values in a single attempt

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.replaceAll((k, v) -> "x"); // values is "x" for all keys.

And replace(K key, V oldValue, V newValue) method replaces the entry for the specified key only if currently mapped to the specified value. In the same way you can use replace, remove methods to check by key and values pairs together.

Do not override keys accidentally use putIfAbsent

As the method name is self explanatory, here is an example.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.putIfAbsent("B", "x");

System.out.println(map.get("B")); // prints "b"

operate directly on values.

Gone are the days when you needed to get the value for specific keys, process it and put them back. Now you can directly modify with help of compute method.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.compute("B", (k, v) -> v.concat(" - new "));

System.out.println(map.get("B")); // prints "b - new"

Conditional computes are also available. Look at computeIfPresent, computeIfAbsent methods.

To merge maps use merge method.

This is little tricky and more useful when you are combining maps or appending values for duplicated keys.

Docs says

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null.

To demonstrate it simply, just merge the values for a key with old and new, see the below example.

Map<String, String> map = new HashMap<>();

map.put("C", "c");

map.put("B", "b");

map.merge("B", "NEW", (v1, v2) -> v1 + v2);

System.out.println(map.get("B")); // prints bNEW

That is all for now and lets meet in next article exploring another Collection changes with Java8.

- Concurrency API improvements:

Your java.util.concurrent package has 2 new interfaces and 4 new classes in Java 8.

Interface CompletableFuture.AsynchronousCompletionTask- This is a marker interface to identify Asynchronous tasks and also to monitor and debug them.

Interface CompletionStage<T>- An asynchronous computation stage, that performs an action when another Completion State completes.

Class CompletableFuture<T>- Future that can be explicitly completed and may be used as a Completion Stage, supporting dependent functions and actions.

Class ConcurrentHashMap.KeySetView<K,V>- View of a Concurrent Hash Map as a Set of Keys

Class CountedCompleter<T>- Basically a Fork Join Task, with a completion action triggered, when there are no pending actions.

New Classes have been added in java.util.concurrent.atomic package too. Scalability is supported through these classes, which internally provide huge throughput improvements as compared to Atomic variables. The new classes here are

DoubleAccumulator- One or more variables that together maintain a running double value.

DoubleAdder-One or more variables that maintains an intial zero double sum together.

LongAccumulator- One or more variables that together maintain a running long value.

LongAdder-One or more variables that maintains an intial zero long sum together.

New methods have been introduced in ConcurrentHashMap based on the newly added lambda expressions and streams facility. These methods include various forEach methods(forEach, forEachKey, forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods (reduce, reduceToDouble, reduceToLong etc.)

You also have a StampedLock class which adds a capability based lock with 3 modes for controlling read/write access.

- Java IO improvements:

IO improvements done in Java 8 include:

Files.list (Path dir): This returns a jlazily populated stream, whose each element is the entry in the directory.

Files.lines (Path path): Reads all the lines from a stream.

Files.find (): Search for files in the file tree rooted at a given starting file and returns a stream populated by a path.

BufferedReader.lines (): Returns a stream with its every element as the lines read from BufferedReader.

Miscellaneous Core API Improvements

We have the following misc API improvements:

Static method withInitial (Supplier supplier) of ThreadLocal to create instance easily.

The interface “Comparator” is extended with the default and static methods for natural ordering reverse order etc.

Integer, Long and Double wrapper classes have min (), max () and sum () methods.

Boolean class is enhanced with logicalAnd (), logicalOr () and logicalXor () methods.

Several utility methods are introduced in the Math class.

JDBC-ODBC Bridge is removed.

PermGen memory space is removed.

**2. What is immutability? How will u make a class immutable in java**

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

INTEGER is immutable when we try change a value it does not change the original value in that address but created a new value at different address

String buffer String builder mutable

Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable. StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for the single-threaded program. If thread safety is needed, then StringBuffer is used.

StringBuilder and StringBuffer to make the String manipulations easy

What will happen to the old value? Will it be left out or how to retrive the old value

All objects are allocated on the heap area managed by the JVM. ... As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory.

Member variables are known as instance variables in java. Instance variables are declared in a class, but outside a method, constructor or any block. When space is allocated for an object in the heap, a slot for each instance variable value is created. ... Instance variables have default values

Following are the requirements:

The class must be declared as final (So that child classes can’t be created)

Data members in the class must be declared as private (So that direct access is not allowed)

Data members in the class must be declared as final (So that we can’t change the value of it after object creation)

A parameterized constructor should initialize all the fields performing a deep copy (So that data members can’t be modified with object reference)

Deep Copy of objects should be performed in the getter methods (To return a copy rather than returning the actual object reference)

No setters (To not have the option to change the value of the instance variable)

Example to create Immutable class

import java.util.HashMap;

import java.util.Map;

// An immutable class

public final class Student {

private final String name;

private final int regNo;

private final Map<String, String> metadata;

public Student(String name, int regNo,

Map<String, String> metadata)

{

this.name = name;

this.regNo = regNo;

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

this.metadata = tempMap;

}

public String getName() { return name; }

public int getRegNo() { return regNo; }

public Map<String, String> getMetadata()

{

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

this.metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

return tempMap;

}

}

// Driver class

class Test {

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("1", "first");

map.put("2", "second");

Student s = new Student("ABC", 101, map);

System.out.println(s.getName());

System.out.println(s.getRegNo());

System.out.println(s.getMetadata());

// Uncommenting below line causes error

// s.regNo = 102;

map.put("3", "third");

System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in constructor

s.getMetadata().put("4", "fourth");

System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in getter

}

}

Output

ABC

101

{1=first, 2=second}

{1=first, 2=second}

{1=first, 2=second}

In this example, we have created a final class named Student. It has three final data members, a parameterized constructor and getter methods. Please note that there is no setter method here. Also note that we don’t need to perform deep copy or cloning of data members of wrapper types as they are already immutable.

**3. Difference between checked and unchecked exceptions**

Java generates two types of exceptions. These are:

Checked Exception

Unchecked Exception

|  |  |
| --- | --- |
| Checked exception | Unchecked exception |
| Checked exceptions occur at compile time. | Unchecked exceptions occur at runtime. |
| The compiler checks a checked exception. | The compiler does not check these types of exceptions. |
| These types of exceptions can be handled at the time of compilation. | These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program. |
| They are the sub-class of the exception class. | They are runtime exceptions and hence are not a part of the Exception class. |
| Here, the JVM needs the exception to catch and handle. | Here, the JVM does not require the exception to catch and handle. |
| Examples of Checked exceptions:  File Not Found Exception  No Such Field Exception  Interrupted Exception  No Such Method Exception  Class Not Found Exception | Examples of Unchecked Exceptions:  No Such Element Exception  Undeclared Throwable Exception  Empty Stack Exception  Arithmetic Exception  Null Pointer Exception  Array Index Out of Bounds Exception  Security Exception |

**4. Different ways of implementing multithreading**

Mutithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as Concurrency in Java. Each thread runs parallel to each other. Mulitple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

Example of Multi thread:

package demotest;

public class GuruThread1 implements Runnable

{

public static void main(String[] args) {

Thread guruThread1 = new Thread("Guru1");

Thread guruThread2 = new Thread("Guru2");

guruThread1.start();

guruThread2.start();

System.out.println("Thread names are following:");

System.out.println(guruThread1.getName());

System.out.println(guruThread2.getName());

}

@Override

public void run() {

}

}

Advantages of multithreading:

The users are not blocked because threads are independent, and we can perform multiple operations at times

As such the threads are independent, the other threads won't get affected if one thread meets an exception.'

Java offers implementation of multi-threading by two ways:

By extending Thread class

By Implementing Runnable interface

You should extend Thread class only when you need to override, use or modify some of the code of Thread class. and do this only when you are sure that your class won’t need to extend other class.

Since it is recommended to use Runnable interface because it is efficient than extending Thread class, because extending Thread class means inserting all the code of Thread class in your current class just for the sake of implementing multi-threading where you are not at all using Thread class’ content except overriding run method

**5. How to make a class singleton**

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.

After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

To design a singleton class:

Make constructor as private.

Write a static method that has return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

Normal class vs Singleton class: Difference in normal and singleton class in terms of instantiation is that, For normal class we use constructor, whereas for singleton class we use getInstance() method (Example code:I). In general, to avoid confusion we may also use the class name as method name while defining this method (Example code:II).

Implementing Singleton class with getInstance() method

// Java program implementing Singleton class

// with getInstance() method

class Singleton

{

// static variable single\_instance of type Singleton

private static Singleton single\_instance = null;

// variable of type String

public String s;

// private constructor restricted to this class itself

private Singleton()

{

s = "Hello I am a string part of Singleton class";

}

// static method to create instance of Singleton class

public static Singleton getInstance()

{

if (single\_instance == null)

single\_instance = new Singleton();

return single\_instance;

}

}

// Driver Class

class Main

{

public static void main(String args[])

{

// instantiating Singleton class with variable x

Singleton x = Singleton.getInstance();

// instantiating Singleton class with variable y

Singleton y = Singleton.getInstance();

// instantiating Singleton class with variable z

Singleton z = Singleton.getInstance();

// changing variable of instance x

x.s = (x.s).toUpperCase();

System.out.println("String from x is " + x.s);

System.out.println("String from y is " + y.s);

System.out.println("String from z is " + z.s);

System.out.println("\n");

// changing variable of instance z

z.s = (z.s).toLowerCase();

System.out.println("String from x is " + x.s);

System.out.println("String from y is " + y.s);

System.out.println("String from z is " + z.s);

}

}

Output:

String from x is HELLO I AM A STRING PART OF SINGLETON CLASS

String from y is HELLO I AM A STRING PART OF SINGLETON CLASS

String from z is HELLO I AM A STRING PART OF SINGLETON CLASS

String from x is hello i am a string part of singleton class

String from y is hello i am a string part of singleton class

String from z is hello i am a string part of singleton class

Singleton class

Explanation: In the Singleton class, when we first time call getInstance() method, it creates an object of the class with name single\_instance and return it to the variable. Since single\_instance is static, it is changed from null to some object. Next time, if we try to call getInstance() method, since single\_instance is not null, it is returned to the variable, instead of instantiating the Singleton class again. This part is done by if condition.

In the main class, we instantiate the singleton class with 3 objects x, y, z by calling static method getInstance(). But actually after creation of object x, variables y and z are pointed to object x as shown in the diagram. Hence, if we change the variables of object x, that is reflected when we access the variables of objects y and z. Also if we change the variables of object z, that is reflected when we access the variables of objects x and y.

**6. How to make a thread safe singleton class**

There are multiple ways you can create singletons in Java, depending upon the requirements — Lazy vs Eager Initialization, thread-safety, you can construct appropriate singleton in Java.

Single Thread Version (non thread-safe)

// Single-threaded version

class Foo {

private Helper helper;

public Helper getHelper() {

if (helper == null) {

helper = new Helper();

}

return helper;

}

// other functions and members...

}

Thread-safe version

The problem with single threaded version is that it will not work predictably when multiple threads tries to access singleton. A lock must be obtained in case two or more threads call getHelper() simultaneously, otherwise there may be problems:

Both threads may try to create the object at the same time, leading to two objects.

One thread may end up getting a reference to an incompletely initialized object.

In this version of singleton, we obtain a lock by expensive method level synchronization technique, as illustrated in the below example:

// Correct but possibly expensive multithreaded version

class Foo {

private Helper helper;

public synchronized Helper getHelper() {

if (helper == null) {

helper = new Helper();

}

return helper;

}

// other functions and members...}

This approach is thread-safe and behavior of the Singleton is predictable in multi-threaded environment, but there are performance issues with this approach:

synchronizing a method could in some extreme cases decrease the performance by a factor of 100 or higher.

once the singleton object is initialized completely, acquiring and releasing the locks would appear unnecessary.

Optimization Algorithm:

Check that the variable is initialized without obtaining the lock. If it is initialized, return it immediately.

Obtain the lock

Double-check whether the variable has already been initialized, if the another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.

Otherwise, initialize and return the variable.

Optimized version of thread-safe Singleton version using locking:

class Foo {

private volatile Helper helper;

public Helper getHelper() {

Helper localRef = helper;

if (localRef == null) {

synchronized (this) {

localRef = helper;

if (localRef == null) {

helper = localRef = new Helper();

}

}

}

return localRef;

}

// other functions and members...

}

Using localRef, we are reducing the access of volatile variable to just one for positive usecase.

The volatile keyword ensures that multiple threads handle the singleton instance correctly.

**7. Solid design principle**

Simply put, Martin and Feathers' design principles encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

The following five concepts make up our SOLID principles:

Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

While these concepts may seem daunting, they can be easily understood with some simple code examples. In the following sections, we'll take a deep dive into these principles, with a quick Java example to illustrate each one.

**8. What is load factor and initial capacity wrt to collection**

The HashMap is one of the high-performance data structure in the Java collections framework. It gives a constant time performance for insertion and retrieval. There are two factors which affect the performance of the hashmap.

Initial Capacity

Load Factor

We have to choose these two factors very carefully while creating the HashMap object. Load Factor and initial capacity can be configured while we create a constructor of HashMap class, as shown below:

HashMap hm=new HashMap(int initialCapacity, float loadFactor);

Initial Capacity of HashMap

The initial capacity of the HashMap is the number of buckets in the hash table. It creates when we create the object of HashMap class. The initial capacity of the HashMap is 24, i.e., 16. The capacity of the HashMap is doubled each time it reaches the threshold. The capacity is increased to 25=32, 26=64, and so on.

Suppose we have implemented the hashCode() method, which makes sure that key-value pair will be distributed among 16 buckets equally.

Consider the following scenarios:

If there are 16 elements in the HashMap, the hashCode() method will distribute one element in each bucket. The searching for any item, in this case, will take the only lookup.

If there are 32 elements in the HashMap, the hashCode() method will distribute two elements in each bucket. The searching for any item, in this case, will take the maximum of two lookups.

Similarly, if there are 128 elements in HashMap, the hashCode() method will distribute eight elements in each bucket. The searching for any item, in this case, will take the maximum eight lookups.

We can observe from the above scenarios that the number of items in HashMap is doubled. The maximum lookup time in each bucket is not increasing very high and remain almost constant.

Alternatively, the hashmap grows in the power of 2n and keep on growing when starting point it reached its limit.

Load Factor

The Load factor is a measure that decides when to increase the HashMap capacity to maintain the get() and put() operation complexity of O(1). The default load factor of HashMap is 0.75f (75% of the map size).

Problem

The problem is, keeping the bucket size fixed (i.e., 16), we keep on increasing the total number of items in the map that disturbs time complexity.

Solution

When we increase the total number of buckets, total items in each bucket starts increasing. Now we are able to keep the constant number of items in each bucket and maintain the time complexity of O(1) for get() and put() operation.

How Load Factor is calculated

Load Factor decides "when to increase the number of buckets."

We can find when to increase the hashmap size by using the following formula:

initial capacity of the hashmap\*Load factor of the hashmap.

The initial capacity of hashmap is=16

The default load factor of hashmap=0.75

According to the formula as mentioned above: 16\*0.75=12

It represents that 12th key-value pair of hashmap will keep its size to 16. As soon as 13th element (key-value pair) will come into the Hashmap, it will increase its size from default 24 = 16 buckets to 25 = 32 buckets.

Another way to calculate size:

When the load factor ratio (m/n) reaches 0.75 at that time, hashmap increases its capacity.

Where,

m is the number of entries in a hashmap.

n is the total size of hashmap.

Example of Load Factor

Let's understand the load factor through an example.

We know that the default bucket size of the hashmap is 16. We insert the first element, now check that we need to increase the hashmap capacity or not. It can be determined by the formula:

Size of hashmap (m)/number of buckets (n)

In this case, the size of the hashmap is 1, and the bucket size is 16. So, 1/16=0.0625. Now compare this value with the default load factor.

0.0625<0.75

So, no need to increase the hashmap size.

We do not need to increase the size of hashmap up to 12th element, because

12/16=0.75

This load factor is equal to the default load factor, i.e., 0.75.

As soon as we insert the 13th element in the hashmap, the size of hashmap is increased because:

13/16=0.8125

Which is greater than the default hashmap size.

0.8125>0.75

Now we need to increase the hashmap size.

If you want to keep get and put complexity O(1), it is advisable to have a load factor around 0.75.

**9. Hashmap, Arraylist, Treeset, Hashset**

HASHMAP:

HashMap<K, V> is a part of Java’s collection since Java 1.2. This class is found in java.util package. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs, and you can access them by an index of another type (e.g. an Integer). One object is used as a key (index) to another object (value). If you try to insert the duplicate key, it will replace the element of the corresponding key.

HashMap is similar to HashTable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the map. To use this class and its methods, you need to import java.util.HashMap package or its superclass.

// Java program to illustrate HashMap class of java.util

// package

// Importing HashMap class

import java.util.HashMap;

// Main class

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Create an empty hash map by declaring object

// of string and integer type

HashMap<String, Integer> map = new HashMap<>();

// Adding elements to the Map

// usiing standard add() method

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);

// Print size and content of the Map

System.out.println("Size of map is:- "

+ map.size());

// Printing elements in object of Map

System.out.println(map);

// Checking if a key is present and if

// present, print value by passing

// random element

if (map.containsKey("vishal")) {

// Mapping

Integer a = map.get("vishal");

// Printing value fr the corresponding key

System.out.println("value for key"

+ " \"vishal\" is:- " + a);

}

}

}

Output

Size of map is:- 3

{vaibhav=20, vishal=10, sachin=30}

value for key "vishal" is:- 10

ARRAYLIST:

The ArrayList class is a resizable array, which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

Example

Create an ArrayList object called cars that will store strings:

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

If you don't know what a package is, read our Java Packages Tutorial.

Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

Example

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

System.out.println(cars);

}

}

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

Example

cars.get(0);

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Item

To modify an element, use the set() method and refer to the index number:

Example

cars.set(0, "Opel");

Remove an Item

To remove an element, use the remove() method and refer to the index number:

Example

cars.remove(0);

To remove all the elements in the ArrayList, use the clear() method:

Example

cars.clear();

ArrayList Size

To find out how many elements an ArrayList have, use the size method:

Example

cars.size();

Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

Example

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (int i = 0; i < cars.size(); i++) {

System.out.println(cars.get(i));

}

}

}

You can also loop through an ArrayList with the for-each loop:

Example

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

for (String i : cars) {

System.out.println(i);

}

}

}

Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: Integer. For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Example

Create an ArrayList to store numbers (add elements of type Integer):

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

myNumbers.add(10);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(25);

for (int i : myNumbers) {

System.out.println(i);

}

}

}

Sort an ArrayList

Another useful class in the java.util package is the Collections class, which include the sort() method for sorting lists alphabetically or numerically:

Example

Sort an ArrayList of Strings:

import java.util.ArrayList;

import java.util.Collections; // Import the Collections class

public class Main {

public static void main(String[] args) {

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("Mazda");

Collections.sort(cars); // Sort cars

for (String i : cars) {

System.out.println(i);

}

}

}

Example

Sort an ArrayList of Integers:

import java.util.ArrayList;

import java.util.Collections; // Import the Collections class

public class Main {

public static void main(String[] args) {

ArrayList<Integer> myNumbers = new ArrayList<Integer>();

myNumbers.add(33);

myNumbers.add(15);

myNumbers.add(20);

myNumbers.add(34);

myNumbers.add(8);

myNumbers.add(12);

Collections.sort(myNumbers); // Sort myNumbers

for (int i : myNumbers) {

System.out.println(i);

}

}

}

TREESET:

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting AbstractSet class.

Set-TreeSet-SortedSet-In-Java-Collection

In the above image, the navigable set extends the sorted set interface. Since a set doesn’t retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

Example: The following implementation demonstrates how to create and use a TreeSet.

// Java program to demonstrate TreeSet

import java.util.\*;

class TreeSetExample {

public static void main(String[] args)

{

Set<String> ts1 = new TreeSet<>();

// Elements are added using add() method

ts1.add("A");

ts1.add("B");

ts1.add("C");

// Duplicates will not get insert

ts1.add("C");

// Elements get stored in default natural

// Sorting Order(Ascending)

System.out.println(ts1);

}

}

Output:

[A, B, C]

HASHSET:

The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. No guarantee is made as to the iteration order of the set which means that the class does not guarantee the constant order of elements over time. This class permits the null element. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming the hash function disperses the elements properly among the buckets, which we shall see further in the article.

Few important features of HashSet are:

Implements Set Interface.

The underlying data structure for HashSet is Hashtable.

As it implements the Set Interface, duplicate values are not allowed.

Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.

NULL elements are allowed in HashSet.

HashSet also implements Serializable and Cloneable interfaces.

The Hierarchy of HashSet

Hierarchy of HashSet

HashSet extends Abstract Set<E> class and implements Set<E>, Cloneable and Serializable interfaces where E is the type of elements maintained by this set. The directly known subclass of HashSet is LinkedHashSet.

Now for the maintenance of constant time performance, iterating over HashSet requires time proportional to the sum of the HashSet instance’s size (the number of elements) plus the “capacity” of the backing HashMap instance (the number of buckets). Thus, it’s very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Initial Capacity: The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.

Load Factor: The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

Number of stored elements in the table

Load Factor = -----------------------------------------

Size of the hash table

Example: If internal capacity is 16 and the load factor is 0.75 then the number of buckets will automatically get increased when the table has 12 elements in it.

Effect on performance: Load factor and initial capacity are two main factors that affect the performance of HashSet operations. A load factor of 0.75 provides very effective performance with respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in the hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Note: The implementation in a HashSet is not synchronized, in the sense that if multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set as shown below:

Set s = Collections.synchronizedSet(new HashSet(...));

Declaration of HashSet:

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

where E is the type of elements stored in a HashSet.

Constructors of HashSet class

In order to create a HashSet, we need to create an object of the HashSet class. The HashSet class consists of various constructors that allow the possible creation of the HashSet. The following are the constructors available in this class.

1. HashSet(): This constructor is used to build an empty HashSet object in which the default initial capacity is 16 and the default load factor is 0.75. If we wish to create an empty HashSet with the name hs, then, it can be created as:

HashSet<E> hs = new HashSet<E>();

2. HashSet(int initialCapacity): This constructor is used to build an empty HashSet object in which the initialCapacity is specified at the time of object creation. Here, the default loadFactor remains 0.75.

HashSet<E> hs = new HashSet<E>(int initialCapacity);

3. HashSet(int initialCapacity, float loadFactor): This constructor is used to build an empty HashSet object in which the initialCapacity and loadFactor are specified at the time of object creation.

HashSet<E> hs = new HashSet<E>(int initialCapacity, float loadFactor);

4. HashSet(Collection): This constructor is used to build a HashSet object containing all the elements from the given collection. In short, this constructor is used when any conversion is needed from any Collection object to the HashSet object. If we wish to create a HashSet with the name hs, it can be created as:

HashSet<E> hs = new HashSet<E>(Collection C);

Example:

// Java program to demonstrate working of HashSet

import java.util.\*;

class HashSetDemo {

// Main Method

public static void main(String[] args)

{

HashSet<String> h = new HashSet<String>();

// Adding elements into HashSet usind add()

h.add("India");

h.add("Australia");

h.add("South Africa");

h.add("India"); // adding duplicate elements

// Displaying the HashSet

System.out.println(h);

System.out.println("List contains India or not:"

+ h.contains("India"));

// Removing items from HashSet using remove()

h.remove("Australia");

System.out.println("List after removing Australia:"

+ h);

// Iterating over hash set items

System.out.println("Iterating over list:");

Iterator<String> i = h.iterator();

while (i.hasNext())

System.out.println(i.next());

}

}

Output:

[South Africa, Australia, India]

List contains India or not:true

List after removing Australia:[South Africa, India]

Iterating over list:

South Africa

India

**10. Type Erasure in generics**

Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime.

For example:

public static <E> boolean containsElement(E [] elements, E element){

for (E e : elements){

if(e.equals(element)){

return true;

}

}

return false;

}

The compiler replaces the unbound type E with an actual type of Object:

public static boolean containsElement(Object [] elements, Object element){

for (Object e : elements){

if(e.equals(element)){

return true;

}

}

return false;

}

Therefore the compiler ensures type safety of our code and prevents runtime errors.

3. Types of Type Erasure

Type erasure can occur at class (or variable) and method levels.

3.1. Class Type Erasure

At the class level, the compiler discards the type parameters on the class and replaces them with its first bound, or Object if the type parameter is unbound.

Let's implement a Stack using an array:

public class Stack<E> {

private E[] stackContent;

public Stack(int capacity) {

this.stackContent = (E[]) new Object[capacity];

}

public void push(E data) {

// ..

}

public E pop() {

// ..

}

}

Upon compilation, the compiler replaces the unbound type parameter E with Object:

public class Stack {

private Object[] stackContent;

public Stack(int capacity) {

this.stackContent = (Object[]) new Object[capacity];

}

public void push(Object data) {

// ..

}

public Object pop() {

// ..

}

}

In a case where the type parameter E is bound:

freestar

public class BoundStack<E extends Comparable<E>> {

private E[] stackContent;

public BoundStack(int capacity) {

this.stackContent = (E[]) new Object[capacity];

}

public void push(E data) {

// ..

}

public E pop() {

// ..

}

}

The compiler will replace the bound type parameter E with the first bound class, Comparable in this case:

public class BoundStack {

private Comparable [] stackContent;

public BoundStack(int capacity) {

this.stackContent = (Comparable[]) new Object[capacity];

}

public void push(Comparable data) {

// ..

}

public Comparable pop() {

// ..

}

}

3.2. Method Type Erasure

For method-level type erasure, the method's type parameter is not stored but rather converted to its parent type Object if it's unbound or it's first bound class when it's bound.

Let's consider a method to display the contents of any given array:

public static <E> void printArray(E[] array) {

for (E element : array) {

System.out.printf("%s ", element);

}

}

Upon compilation, the compiler replaces the type parameter E with Object:

public static void printArray(Object[] array) {

for (Object element : array) {

System.out.printf("%s ", element);

}

}

For a bound method type parameter:

public static <E extends Comparable<E>> void printArray(E[] array) {

for (E element : array) {

System.out.printf("%s ", element);

}

}

We'll have the type parameter E erased and replaced with Comparable:

public static void printArray(Comparable[] array) {

for (Comparable element : array) {

System.out.printf("%s ", element);

}

}

**11. Wild card in generics**

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

Types of wildcards in Java:

Upper Bounded Wildcards: These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.

public static void add(List<? extends Number> list)

Implementation:

//Java program to demonstrate Upper Bounded Wildcards

import java.util.Arrays;

import java.util.List;

class WildcardDemo

{

public static void main(String[] args)

{

//Upper Bounded Integer List

List<Integer> list1= Arrays.asList(4,5,6,7);

//printing the sum of elements in list

System.out.println("Total sum is:"+sum(list1));

//Double list

List<Double> list2=Arrays.asList(4.1,5.1,6.1);

//printing the sum of elements in list

System.out.print("Total sum is:"+sum(list2));

}

private static double sum(List<? extends Number> list)

{

double sum=0.0;

for (Number i: list)

{

sum+=i.doubleValue();

}

return sum;

}

}

Output:

Total sum is:22.0

Total sum is:15.299999999999999

In the above program, list1 and list2 are objects of the List class. list1 is a collection of Integer and list2 is a collection of Double. Both of them are being passed to method sum which has a wildcard that extends Number. This means that list being passed can be of any field or subclass of that field. Here, Integer and Double are subclasses of class Number.

Lower Bounded Wildcards: It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.

Syntax: Collectiontype <? super A>

Implementation:

//Java program to demonstrate Lower Bounded Wildcards

import java.util.Arrays;

import java.util.List;

class WildcardDemo

{

public static void main(String[] args)

{

//Lower Bounded Integer List

List<Integer> list1= Arrays.asList(4,5,6,7);

//Integer list object is being passed

printOnlyIntegerClassorSuperClass(list1);

//Number list

List<Number> list2= Arrays.asList(4,5,6,7);

//Integer list object is being passed

printOnlyIntegerClassorSuperClass(list2);

}

public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list)

{

System.out.println(list);

}

}

Output:

[4, 5, 6, 7]

[4, 5, 6, 7]

Here arguments can be Integer or superclass of Integer(which is Number). The method printOnlyIntegerClassorSuperClass will only take Integer or its superclass objects. However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed . Double is not the superclass of Integer.

Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don’t use wildcard when you get and put values in a structure.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Unbounded Wildcard: This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases

When writing a method which can be employed using functionality provided in Object class.

When the code is using methods in the generic class that don’t depend on the type parameter

Implementation:

//Java program to demonstrate Unbounded wildcard

import java.util.Arrays;

import java.util.List;

class unboundedwildcardemo

{

public static void main(String[] args)

{

//Integer List

List<Integer> list1= Arrays.asList(1,2,3);

//Double list

List<Double> list2=Arrays.asList(1.1,2.2,3.3);

printlist(list1);

printlist(list2);

}

private static void printlist(List<?> list)

{

System.out.println(list);

}

}

Output:

[1, 2, 3]

[1.1, 2.2, 3.3]

**12. Benefits of generics**

The Object is the superclass of all other classes and Object reference can refer to any type object. These features lack type safety. Generics add that type safety feature. We will discuss that type of safety feature in later examples.

Generics in Java is similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. There are some fundamental differences between the two approaches to generic types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we can not use primitives like

'int','char' or 'double'.

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.

2. Type Safety: Generics make errors to appear compile time than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

// A Simple Java program to demonstrate that NOT using

// generics can cause run time exceptions

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creatinga an ArrayList without any type specified

ArrayList al = new ArrayList();

al.add("Sachin");

al.add("Rahul");

al.add(10); // Compiler allows this

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

// Causes Runtime Exception

String s3 = (String)al.get(2);

}

}

Output :

Exception in thread "main" java.lang.ClassCastException:

java.lang.Integer cannot be cast to java.lang.String

at Test.main(Test.java:19)

How generics solve this problem?

At the time of defining ArrayList, we can specify that this list can take only String objects.

// Using generics converts run time exceptions into

// compile time exception.

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Now Compiler doesn't allow this

al.add(10);

String s1 = (String)al.get(0);

String s2 = (String)al.get(1);

String s3 = (String)al.get(2);

}

}

Output:

15: error: no suitable method found for add(int)

al.add(10);

^

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example every time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not typecast it every time.

// We don't need to typecast individual members of ArrayList

import java.util.\*;

class Test

{

public static void main(String[] args)

{

// Creating a an ArrayList with String specified

ArrayList <String> al = new ArrayList<String> ();

al.add("Sachin");

al.add("Rahul");

// Typecasting is not needed

String s1 = al.get(0);

String s2 = al.get(1);

}

}

4. Generics promotes code reusability.

5. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same, they are type safe too.

**13. Dynamic polymorphism**

Dynamic Method Dispatch or Runtime Polymorphism in Java

Difficulty Level : Medium

Last Updated : 07 Sep, 2018

Prerequisite: Overriding in java, Inheritance

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

Blank Diagram - Page 1 (4)

Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

// A Java program to illustrate Dynamic Method

// Dispatch using hierarchical inheritance

class A

{

void m1()

{

System.out.println("Inside A's m1 method");

}

}

class B extends A

{

// overriding m1()

void m1()

{

System.out.println("Inside B's m1 method");

}

}

class C extends A

{

// overriding m1()

void m1()

{

System.out.println("Inside C's m1 method");

}

}

// Driver class

class Dispatch

{

public static void main(String args[])

{

// object of type A

A a = new A();

// object of type B

B b = new B();

// object of type C

C c = new C();

// obtain a reference of type A

A ref;

// ref refers to an A object

ref = a;

// calling A's version of m1()

ref.m1();

// now ref refers to a B object

ref = b;

// calling B's version of m1()

ref.m1();

// now ref refers to a C object

ref = c;

// calling C's version of m1()

ref.m1();

}

}

Output:

Inside A's m1 method

Inside B's m1 method

Inside C's m1 method

Explanation :

The above program creates one superclass called A and it’s two subclasses B and C. These subclasses overrides m1( ) method.

Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

A a = new A(); // object of type A

B b = new B(); // object of type B

C c = new C(); // object of type C

Blank Diagram - Page 1 (1)

Now a reference of type A, called ref, is also declared, initially it will point to null.

A ref; // obtain a reference of type A

w

Now we are assigning a reference to each type of object (either A’s or B’s or C’s) to ref, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined by the type of object being referred to at the time of the call.

ref = a; // r refers to an A object

ref.m1(); // calling A's version of m1()

q

ref = b; // now r refers to a B object

ref.m1(); // calling B's version of m1()

q

ref = c; // now r refers to a C object

ref.m1(); // calling C's version of m1()

Blank Diagram - Page 1 (3)

Runtime Polymorphism with Data Members

In Java, we can override methods only, not the variables(data members), so runtime polymorphism cannot be achieved by data members. For example :

// Java program to illustrate the fact that

// runtime polymorphism cannot be achieved

// by data members

// class A

class A

{

int x = 10;

}

// class B

class B extends A

{

int x = 20;

}

// Driver class

public class Test

{

public static void main(String args[])

{

A a = new B(); // object of type B

// Data member of class A will be accessed

System.out.println(a.x);

}

}

Output:

10

Explanation : In above program, both the class A(super class) and B(sub class) have a common variable ‘x’. Now we make object of class B, referred by ‘a’ which is of type of class A. Since variables are not overridden, so the statement “a.x” will always refer to data member of super class.

Advantages of Dynamic Method Dispatch

Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.

It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Static vs Dynamic binding

Static binding is done during compile-time while dynamic binding is done during run-time.

private, final and static methods and variables uses static binding and bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object

**14. Difference between PUT and POST**

It has been observed that many people struggle to choose between HTTP PUT vs. POST methods when designing a system. Though, RFC 2616 has been very clear in differentiating between the two – yet complex wordings are a source of confusion for many of us. Let’s try to solve the puzzle when to use PUT or POST.

Let’s compare them for better understanding.

PUT vs POST

RFC-2616 clearly mention that PUT method requests for the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource – an update operation will happen, otherwise create operation should happen if Request-URI is a valid resource URI (assuming client is allowed to determine resource identifier).

PUT /questions/{question-id}

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It essentially means that POST request-URI should be of a collection URI.

POST /questions

PUT method is idempotent. So if you send retry a request multiple times, that should be equivalent to single request modification. POST is NOT idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on server.

Use PUT when you want to modify a singular resource which is already a part of resources collection. PUT replaces the resource in its entirety. Use PATCH if request updates part of the resource. Use POST when you want to add a child resource under resources collection.

Though PUT is idempotent, we shall not cache it’s response. Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields. However, the 303 (See Other) response can be used to direct the user agent to retrieve a cacheable resource.

Generally, in practice, always use PUT for UPDATE operations. Always use POST for CREATE operations.

**15. What is inversion of control and dependency injection in Spring**

I shall write down my simple understanding of this two terms: (For quick understanding just read examples)

Dependency Injection(DI):

Dependency injection generally means passing a dependent object as a parameter to a method, rather than having the method create the dependent object.

What it means in practice is that the method does not have a direct dependency on a particular implementation; any implementation that meets the requirements can be passed as a parameter.

With this implementation of objects defines their dependencies. And spring makes it available.

This leads to loosely coupled application development.

Quick Example:EMPLOYEE OBJECT WHEN CREATED,IT WILL AUTOMATICALLY CREATE ADDRESS OBJECT (if address is defines as dependency by Employee object)\*.

Inversion of Control(IoC) Container:

This is common characteristic of frameworks, IoC manages java objects

- from instantiation to destruction through its BeanFactory.

- Java components that are instantiated by the IoC container are called beans, and the IoC container manages a bean's scope, lifecycle events, and any AOP features for which it has been configured and coded.

QUICK EXAMPLE:

Inversion of Control is about getting freedom, more flexibility, and less dependency. When you are using a desktop computer, you are slaved (or say, controlled). You have to sit before a screen and look at it. Using keyboard to type and using mouse to navigate. And a bad written software can slave you even more. If you replaced your desktop with a laptop, then you somewhat inverted control. You can easily take it and move around. So now you can control where you are with your computer, instead of computer controlling it.

By implementing Inversion of Control, a software/object consumer get more controls/options over the software/objects, instead of being controlled or having less options.

Inversion of control as a design guideline serves the following purposes:

- There is a decoupling of the execution of a certain task from implementation.

- Every module can focus on what it is designed for.

- Modules make no assumptions about what other systems do but rely on their contracts.

- Replacing modules has no side effect on other modules

**16. What is enum? Difference Between enum and constant**

What the value inside enum known as in terms of java?

An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

Difference between Enums and Classes

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Additionally, Enum classes are type-safe, whereas static fields aren't. Compile time error checking is now possible, versus the run-time potential errors that will occur with a constant class. This furthermore improves readability, because instead of having errors where an index of a list of constants is unavailable. This is all well explained in this post, by the current best answer.

public enum Light {

RED, GREEN, YELLOW;

}

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list. Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience.

So when declaring an enum, you have to provide a list of all possible values. From compiler’s perspective, these will be the only compatible values that can be assigned to the variable anywhere in a program.

OO languages usually implement enums as a class where each “option” is represented by a fixed instance of the class. The instances are created automatically at runtime and the compiler prevents you from creating them manually outside enum declaration. This has the added benefit of a type safety, auto completion, serialization and so on.

You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

The class Color is an enumeration (or enum)

The attributes Color.RED, Color.GREEN, etc., are enumeration members (or enum members) and are functionally constants.

The enum members have names and values (the name of Color.RED is RED, the value of Color.BLUE is 3, etc.)

**17. Equals and hascode method and their significance**

Equals() and Hashcode() in Java

The equals() and hashcode() are the two important methods provided by the Object class for comparing objects. Since the Object class is the parent class for all Java objects, hence all objects inherit the default implementation of these two methods. In this topic, we will see the detailed description of equals() and hashcode() methods, how they are related to each other, and how we can implement these two methods in Java.

Java equals()

The java equals() is a method of lang.Object class, and it is used to compare two objects.

To compare two objects that whether they are the same, it compares the values of both the object's attributes.

By default, two objects will be the same only if stored in the same memory location.

Syntax:

public boolean equals(Object obj)

Parameter:

obj: It takes the reference object as the parameter, with which we need to make the comparison.

Returns:

It returns the true if both the objects are the same, else returns false.

General Contract of equals() method

There are some general principles defined by Java SE that must be followed while implementing the equals() method in Java. The equals() method must be:

reflexive: An object x must be equal to itself, which means, for object x, equals(x) should return true.

symmetric: for two given objects x and y, x.equals(y) must return true if and only if equals(x) returns true.

transitive: for any objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

consistent: for any objects x and y, the value of x.equals(y) should change, only if the property in equals() changes.

For any object x, the equals(null) must return false.

Java hashcode()

A hashcode is an integer value associated with every object in Java, facilitating the hashing in hash tables.

To get this hashcode value for an object, we can use the hashcode() method in Java. It is the means hashcode() method that returns the integer hashcode value of the given object.

Since this method is defined in the Object class, hence it is inherited by user-defined classes also.

The hashcode() method returns the same hash value when called on two objects, which are equal according to the equals() method. And if the objects are unequal, it usually returns different hash values.

Syntax:

public int hashCode()

Returns:

It returns the hash code value for the given objects.

Contract for hashcode() method in Java

If two objects are the same as per the equals(Object) method, then if we call the hashCode() method on each of the two objects, it must provide the same integer result.

Equals() and Hashcode() in Java

Note: As per the Java documentation, both the methods should be overridden to get the complete equality mechanism; using equals() alone is not sufficient. It means, if we override the equals(), we must override the hashcode() method.

Example:

class Test\_hash\_equal{

public static void main(String[] args){

String a = "Andrew";

String b = "Andrew";

if(a.equals(b)){ //checking the equality of objects using equals() methods

System.out.println("a & b are equal variables, and their respective hashvalues are:" + " "+ a.hashCode() + " & " + b.hashCode());

}

String c = "Maria";

String d= "Julie";

if(!c.equals(d)){ //checking the equality of objects using equals() method

System.out.println("\nc & d are Un-equal variables, and their respective hashvalues are:" + " "+ c.hashCode() + " & " + d.hashCode());

}

}

}

Output:

a & b are equal variables, and their respective hash values are: 1965574029 & 1965574029

c & d are Un-equal variables, and their respective hash values are: 74113750 & 71933245

In the above example, we have taken two 4 variables, out of which two are equal, and two are unequal. First, we have compared the objects whether they are equal or unequal, and based on that, printed their hash values.

et's try to understand it with an example of what would happen if we override equals() without overriding hashCode() and attempt to use a Map.

Say we have a class like this and that two objects of MyClass are equal if their importantField is equal (with hashCode() and equals() generated by eclipse)

public class MyClass {

private final String importantField;

private final String anotherField;

public MyClass(final String equalField, final String anotherField) {

this.importantField = equalField;

this.anotherField = anotherField;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result

+ ((importantField == null) ? 0 : importantField.hashCode());

return result;

}

@Override

public boolean equals(final Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

final MyClass other = (MyClass) obj;

if (importantField == null) {

if (other.importantField != null)

return false;

} else if (!importantField.equals(other.importantField))

return false;

return true;

}

}

Imagine you have this

MyClass first = new MyClass("a","first");

MyClass second = new MyClass("a","second");

Override only equals

If only equals is overriden, then when you call myMap.put(first,someValue) first will hash to some bucket and when you call myMap.put(second,someOtherValue) it will hash to some other bucket (as they have a different hashCode). So, although they are equal, as they don't hash to the same bucket, the map can't realize it and both of them stay in the map.

Although it is not necessary to override equals() if we override hashCode(), let's see what would happen in this particular case where we know that two objects of MyClass are equal if their importantField is equal but we do not override equals().

Override only hashCode

If you only override hashCode then when you call myMap.put(first,someValue) it takes first, calculates its hashCode and stores it in a given bucket. Then when you call myMap.put(second,someOtherValue) it should replace first with second as per the Map Documentation because they are equal (according to the business requirement).

But the problem is that equals was not redefined, so when the map hashes second and iterates through the bucket looking if there is an object k such that second.equals(k) is true it won't find any as second.equals(first) will be false.

**18. Cloning**

Object Cloning in Java

constructor in javaThe object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

The clone() method is defined in the Object class. Syntax of the clone() method is as follows:

protected Object clone() throws CloneNotSupportedException

Why use clone() method ?

The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.

It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.

Clone() is the fastest way to copy array.

Disadvantage of Object cloning

Following is a list of some disadvantages of clone() method:

To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.

We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.

Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.

Object.clone() doesn't invoke any constructor so we don't have any control over object construction.

If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

class Student18 implements Cloneable{

int rollno;

String name;

Student18(int rollno,String name){

this.rollno=rollno;

this.name=name;

}

public Object clone()throws CloneNotSupportedException{

return super.clone();

}

public static void main(String args[]){

try{

Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);

System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}

}

Test it Now

Output:101 amit

101 amit

**19. Basic knowledge of reflections**

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

The required classes for reflection are provided under java.lang.reflect package.

Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.

Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

reflection

Reflection can be used to get information about –

Class The getClass() method is used to get the name of the class to which an object belongs.

Constructors The getConstructors() method is used to get the public constructors of the class to which an object belongs.

Methods The getMethods() method is used to get the public methods of the class to which an objects belongs.

// A simple Java program to demonstrate the use of reflection

import java.lang.reflect.Method;

import java.lang.reflect.Field;

import java.lang.reflect.Constructor;

// class whose object is to be created

class Test

{

// creating a private field

private String s;

// creating a public constructor

public Test() { s = "GeeksforGeeks"; }

// Creating a public method with no arguments

public void method1() {

System.out.println("The string is " + s);

}

// Creating a public method with int as argument

public void method2(int n) {

System.out.println("The number is " + n);

}

// creating a private method

private void method3() {

System.out.println("Private method invoked");

}

}

class Demo

{

public static void main(String args[]) throws Exception

{

// Creating object whose property is to be checked

Test obj = new Test();

// Creating class object from the object using

// getclass method

Class cls = obj.getClass();

System.out.println("The name of class is " +

cls.getName());

// Getting the constructor of the class through the

// object of the class

Constructor constructor = cls.getConstructor();

System.out.println("The name of constructor is " +

constructor.getName());

System.out.println("The public methods of class are : ");

// Getting methods of the class through the object

// of the class by using getMethods

Method[] methods = cls.getMethods();

// Printing method names

for (Method method:methods)

System.out.println(method.getName());

// creates object of desired method by providing the

// method name and parameter class as arguments to

// the getDeclaredMethod

Method methodcall1 = cls.getDeclaredMethod("method2",

int.class);

// invokes the method at runtime

methodcall1.invoke(obj, 19);

// creates object of the desired field by providing

// the name of field as argument to the

// getDeclaredField method

Field field = cls.getDeclaredField("s");

// allows the object to access the field irrespective

// of the access specifier used with the field

field.setAccessible(true);

// takes object and the new value to be assigned

// to the field as arguments

field.set(obj, "JAVA");

// Creates object of desired method by providing the

// method name as argument to the getDeclaredMethod

Method methodcall2 = cls.getDeclaredMethod("method1");

// invokes the method at runtime

methodcall2.invoke(obj);

// Creates object of the desired method by providing

// the name of method as argument to the

// getDeclaredMethod method

Method methodcall3 = cls.getDeclaredMethod("method3");

// allows the object to access the method irrespective

// of the access specifier used with the method

methodcall3.setAccessible(true);

// invokes the method at runtime

methodcall3.invoke(obj);

}

}

Output :

The name of class is Test

The name of constructor is Test

The public methods of class are :

method2

method1

wait

wait

wait

equals

toString

hashCode

getClass

notify

notifyAll

The number is 19

The string is JAVA

Private method invoked

Important observations :

We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose

getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

Class.getDeclaredMethod(name, parametertype)

name- the name of method whose object is to be created

parametertype - parameter is an array of Class objects

invoke() : To invoke a method of the class at runtime we use following method–

Method.invoke(Object, parameter)

If the method of the class doesn’t accepts any

parameter then null is passed as argument.

Through reflection we can access the private variables and methods of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.

Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.

Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Advantages of Using Reflection:

Extensibility Features: An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.

Debugging and testing tools: Debuggers use the property of reflection to examine private members on classes.

Drawbacks:

Performance Overhead: Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

Exposure of Internals: Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

**20. Spring Profiles**

Profiles are a core feature of the framework — allowing us to map our beans to different profiles — for example, dev, test, and prod.

We can then activate different profiles in different environments to bootstrap only the beans we need.

Further reading:

Configuring Separate Spring DataSource for Tests

A quick, practical tutorial on how to configure a separate data source for testing in a Spring application.

Read more →

Properties with Spring and Spring Boot

Tutorial for how to work with properties files and property values in Spring.

Read more →

2. Use @Profile on a Bean

Let's start simple and look at how we can make a bean belong to a particular profile. We use the @Profile annotation — we are mapping the bean to that particular profile; the annotation simply takes the names of one (or multiple) profiles.

Consider a basic scenario: We have a bean that should only be active during development but not deployed in production.

freestar

We annotate that bean with a dev profile, and it will only be present in the container during development. In production, the dev simply won't be active:

@Component

@Profile("dev")

public class DevDatasourceConfig

As a quick sidenote, profile names can also be prefixed with a NOT operator, e.g., !dev, to exclude them from a profile.

In the example, the component is activated only if dev profile is not active:

@Component

@Profile("!dev")

public class DevDatasourceConfig

3. Declare Profiles in XML

Profiles can also be configured in XML. The <beans> tag has a profile attribute, which takes comma-separated values of the applicable profiles:

<beans profile="dev">

<bean id="devDatasourceConfig"

class="org.baeldung.profiles.DevDatasourceConfig" />

</beans>

4. Set Profiles

The next step is to activate and set the profiles so that the respective beans are registered in the container.

freestar

This can be done in a variety of ways, which we'll explore in the following sections.

4.1. Programmatically via WebApplicationInitializer Interface

In web applications, WebApplicationInitializer can be used to configure the ServletContext programmatically.

It's also a very handy location to set our active profiles programmatically:

@Configuration

public class MyWebApplicationInitializer

implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletContext) throws ServletException {

servletContext.setInitParameter(

"spring.profiles.active", "dev");

}

}

4.2. Programmatically via ConfigurableEnvironment

We can also set profiles directly on the environment:

@Autowired

private ConfigurableEnvironment env;

...

env.setActiveProfiles("someProfile");

4.3. Context Parameter in web.xml

Similarly, we can define the active profiles in the web.xml file of the web application, using a context parameter:

freestar

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-config.xml</param-value>

</context-param>

<context-param>

<param-name>spring.profiles.active</param-name>

<param-value>dev</param-value>

</context-param>

4.4. JVM System Parameter

The profile names can also be passed in via a JVM system parameter. These profiles will be activated during application startup:

-Dspring.profiles.active=dev

4.5. Environment Variable

In a Unix environment, profiles can also be activated via the environment variable:

export spring\_profiles\_active=dev

4.6. Maven Profile

Spring profiles can also be activated via Maven profiles, by specifying the spring.profiles.active configuration property.

In every Maven profile, we can set a spring.profiles.active property:

<profiles>

<profile>

<id>dev</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<spring.profiles.active>dev</spring.profiles.active>

</properties>

</profile>

<profile>

<id>prod</id>

<properties>

<spring.profiles.active>prod</spring.profiles.active>

</properties>

</profile>

</profiles>

Its value will be used to replace the @spring.profiles.active@ placeholder in application.properties:

spring.profiles.active=@spring.profiles.active@

Now we need to enable resource filtering in pom.xml:

<build>

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

...

</build>

and append a -P parameter to switch which Maven profile will be applied:

mvn clean package -Pprod

This command will package the application for prod profile. It also applies the spring.profiles.active value prod for this application when it is running.

4.7. @ActiveProfile in Tests

Tests make it very easy to specify what profiles are active using the @ActiveProfile annotation to enable specific profiles:

@ActiveProfiles("dev")

So far, we've looked at multiple ways of activating profiles. Let's now see which one has priority over the other and what happens if we use more than one, from highest to lowest priority:

Context parameter in web.xml

WebApplicationInitializer

JVM System parameter

Environment variable

Maven profile

5. The Default Profile

Any bean that does not specify a profile belongs to the default profile.

Spring also provides a way to set the default profile when no other profile is active — by using the spring.profiles.default property.

6. Get Active Profiles

Spring's active profiles drive the behavior of the @Profile annotation for enabling/disabling beans. However, we may also wish to access the list of active profiles programmatically.

We have two ways to do it, using Environment or spring.active.profile.

6.1. Using Environment

We can access the active profiles from the Environment object by injecting it:

public class ProfileManager {

@Autowired

private Environment environment;

public void getActiveProfiles() {

for (String profileName : environment.getActiveProfiles()) {

System.out.println("Currently active profile - " + profileName);

}

}

}

6.2. Using spring.active.profile

Alternatively, we could access the profiles by injecting the property spring.profiles.active:

@Value("${spring.profiles.active}")

private String activeProfile;

Here, our activeProfile variable will contain the name of the profile that is currently active, and if there are several, it'll contain their names separated by a comma.

However, we should consider what would happen if there is no active profile at all. With our code above, the absence of an active profile would prevent the application context from being created. This would result in an IllegalArgumentException owing to the missing placeholder for injecting into the variable.

In order to avoid this, we can define a default value:

@Value("${spring.profiles.active:}")

private String activeProfile;

Now, if no profiles are active, our activeProfile will just contain an empty string.

And if we want to access the list of them just like in the previous example, we can do it by splitting the activeProfile variable:

public class ProfileManager {

@Value("${spring.profiles.active:}")

private String activeProfiles;

public String getActiveProfiles() {

for (String profileName : activeProfiles.split(",")) {

System.out.println("Currently active profile - " + profileName);

}

}

}

7. Example: Separate Data Source Configurations Using Profiles

Now that the basics are out of the way, let's take a look at a real example.

Consider a scenario where we have to maintain the data source configuration for both the development and production environments.

Let's create a common interface DatasourceConfig that needs to be implemented by both data source implementations:

public interface DatasourceConfig {

public void setup();

}

Following is the configuration for the development environment:

@Component

@Profile("dev")

public class DevDatasourceConfig implements DatasourceConfig {

@Override

public void setup() {

System.out.println("Setting up datasource for DEV environment. ");

}

}

And configuration for the production environment:

@Component

@Profile("production")

public class ProductionDatasourceConfig implements DatasourceConfig {

@Override

public void setup() {

System.out.println("Setting up datasource for PRODUCTION environment. ");

}

}

Now let's create a test and inject our DatasourceConfig interface; depending on the active profile, Spring will inject DevDatasourceConfig or ProductionDatasourceConfig bean:

public class SpringProfilesWithMavenPropertiesIntegrationTest {

@Autowired

DatasourceConfig datasourceConfig;

public void setupDatasource() {

datasourceConfig.setup();

}

}

When the dev profile is active, Spring injects DevDatasourceConfig object, and when calling then setup() method, the following is the output:

Setting up datasource for DEV environment.

8. Profiles in Spring Boot

Spring Boot supports all the profile configuration outlined so far, with a few additional features.

8.1. Activating or Setting a Profile

The initialization parameter spring.profiles.active, introduced in Section 4, can also be set up as a property in Spring Boot to define currently active profiles. This is a standard property that Spring Boot will pick up automatically:

spring.profiles.active=dev

However, starting Spring Boot 2.4, this property cannot be used in conjunction with spring.config.activate.on-profile, as this could raise a ConfigDataException (i.e. an InvalidConfigDataPropertyException or an InactiveConfigDataAccessException).

To set profiles programmatically, we can also use the SpringApplication class:

SpringApplication.setAdditionalProfiles("dev");

To set profiles using Maven in Spring Boot, we can specify profile names under spring-boot-maven-plugin in pom.xml:

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<profiles>

<profile>dev</profile>

</profiles>

</configuration>

</plugin>

...

</plugins>

and execute the Spring Boot-specific Maven goal:

mvn spring-boot:run

8.2. Profile-specific Properties Files

However, the most important profiles-related feature that Spring Boot brings is profile-specific properties files. These have to be named in the format application-{profile}.properties.

Spring Boot will automatically load the properties in an application.properties file for all profiles, and the ones in profile-specific .properties files only for the specified profile.

For example, we can configure different data sources for dev and production profiles by using two files named application-dev.properties and application-production.properties:

In the application-production.properties file, we can set up a MySql data source:

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/db

spring.datasource.username=root

spring.datasource.password=root

Then we can configure the same properties for the dev profile in the application-dev.properties file, to use an in-memory H2 database:

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1

spring.datasource.username=sa

spring.datasource.password=sa

In this way, we can easily provide different configurations for different environments.

Prior to Spring Boot 2.4, it was possible to activate a profile from profile-specific documents. But that is no longer the case; with later versions, the framework will throw – again – an InvalidConfigDataPropertyException or an InactiveConfigDataAccessException in these circumstances.

8.3. Multi-Document Files

To further simplify defining properties for separate environments, we can even club all the properties in the same file and use a separator to indicate the profile.

Starting version 2.4, Spring Boot has extended its support for multi-document files for properties files in addition to previously supported YAML. So now, we can specify the dev and production properties in the same application.properties:

my.prop=used-always-in-all-profiles

#---

spring.config.activate.on-profile=dev

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/db

spring.datasource.username=root

spring.datasource.password=root

#---

spring.config.activate.on-profile=production

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1

spring.datasource.username=sa

spring.datasource.password=sa

This file is read by Spring Boot in top to bottom order. That is, if some property, say my.prop, occurs once more at the end in the above example, the endmost value will be considered.

8.4. Profile Groups

Another feature added in Boot 2.4 is Profile Groups. As the name suggests, it allows us to group similar profiles together.

Let's consider a use case where we'd have multiple configuration profiles for the production environment. Say, a proddb for the database and prodquartz for the scheduler in the production environment.

To enable these profiles all at once via our application.properties file, we can specify:

spring.profiles.group.production=proddb,prodquartz

Consequently, activating the production profile will activate proddb and prodquartz as well.

9. Conclusion

In this article, we discussed how to define a profile on a bean and how to then enable the right profiles in our application.

**21. Annotation vs XML based configuration**

Advantages of the annotation:

1) All the information is in a single file (no need to open two files to configure a given behavior)

2) When the class changes, no need to modify the xml file

Advantages of xml file:

1) Clear separation between the POJO and its behavior

2) When you do not know which POJO is responsible for the behavior, it is easier to find that POJO (searching in a subset of files rather than all the source code)

irst of all we use annotations for many more things, than just configuration.

Now: Some advantages of using annotations for configuration

Readability. For example in JPA configuration its much more cleaner to declare new entities by Annotations instead of hbm.xml files. These things change only in development stage so there is no problem with recompiling code. When You use xml files You have to often open both- entity and hbm file to make changes.. That can cause some errors.

Flexibility. In XML files you have to write all configs in "only one proper way". It is advantage and disadvantage at the same time.

Length. XML-based configs are often very long (like pom's, or hbm's). Annotations are much simpler to use.

**22. Basic SQL Knowledge**

**23. Basic knowledge of unit testing using junit and mockito**

Mockito is a mocking framework. It is a Java-based library used to create simple and basic test APIs for performing unit testing of Java applications. It can also be used with other frameworks such as JUnit and TestNG.

What is Unit testing?

Unit testing is a software testing technique in which individual components/parts of the software is tested, i.e., a group of computer programs, usage procedure, etc. Unit testing of an object is done during the development of an application or project. The aim of unit testing is to isolate a segment of code (unit) and verifies its correctness. A unit is referred to as an individual function or procedure (program). The developers usually perform it during testing.

What is Mocking?

Mocking is a process of developing the objects that act as the mock or clone of the real objects. In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes. Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.

The mocking technique is not only used in Java but also used in any object-oriented programming language. There are many frameworks available in Java for mocking, but Mockito is the most popular framework among them.

To mock objects, you need to understand the three key concepts of mocking, i.e., stub, fake, and mock. Some of the unit tests involve only stubs, whereas some involve fake and mocks.

The brief description of the mocking concepts is given below:

Stub: Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for state verification.

Fake: Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.

Mock: Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for behavior verification.

Need for mocking

Before using the Mocking technique, we should know the reasons for using mocking, which are as follows:

If we want to test a component that depends on the other component, but it is under development. It generally uses when working in a team and parts are divided between several team-mates. In this case, mocking plays an essential role in the testing of that component. Without mocking, we need to wait for the completion of the required elements for testing.

If the real components perform slow operations while dealing with database connections or another complex read/ write operation. Sometimes the database queries can take 10, 20, or more seconds to execute. In such cases, we require mock objects to perform testing, and it can be done via mocking.

If there is an infrastructure concern that makes the testing impossible. It is very similar to the first case. For example, when we create a connection to the database, some issues related to configurations occur. It requires mocking for creating mock components to provide unit testing.

What is Mockito?

Mockito is a Java-based mocking framework used for unit testing of Java application. Mockito plays a crucial role in developing testable applications. Mockito was released as an open-source testing framework under the MIT (Massachusetts Institute of Technology) License. It internally uses the Java Reflection API to generate mock objects for a specific interface. Mock objects are referred to as the dummy or proxy objects used for actual implementations.

The main purpose of using the Mockito framework is to simplify the development of a test by mocking external dependencies and use them in the test code. As a result, it provides a simpler test code that is easier to read, understand, and modify. We can also use Mockito with other testing frameworks like JUnit and TestNG.

**24. What is persistence context and application context**

JPA/Hibernate Persistence

Persistence Context is an environment or cache where entity instances(which are capable of holding data and thereby having the ability to be persisted in a database) are managed by Entity Manager.It syncs the entity with database. All objects having @Entity annotation are capable of being persisted.

The persistence context is the first-level cache where all the entities are fetched from the database or saved to the database.

Entities are managed by javax.persistence.EntityManager instance using persistence context.

Each EntityManager instance is associated with a persistence context.

Within the persistence context, the entity instances and their lifecycle are managed.

Persistence context defines a scope under which particular entity instances are created, persisted, and removed.

A persistence context is like a cache which contains a set of persistent entities , So once the transaction is finished, all persistent objects are detached from the EntityManager's persistence context and are no longer managed.

The ApplicationContext is the central interface within a Spring application that is used for providing configuration information to the application. It's created when the application starts running.

It provides the entire configuration needed by our application:

Bean Factory - Responsible for creation of java objects called beans. One example is components in the application.

Application listeners - all listeners needed for events.

WebServer information.

Application current environment specific information.

Resource pattern resolver - resource loader with path matcher.

Life cycle Processor.

Class Loader.

Start and shutdown monitor.

Servlet Context.

Reader and Scanner.

Logger

**25. Explain ssl handshake**

it goes with all handshakes, the SSL/TLS Handshake is where it all starts. The SSL/TLS handshake involves a series of steps through which both the parties – client and server, validate each other and start communicating through the secure SSL/TLS tunnel.

SSL Handshake Explained

The reason it’s called a handshake is that it’s when two parties – client and server come across each other for the first time. The handshake involves a number of steps that start from validating the identity of the other party and concludes with the generation of a common key – secret key if you may call it.

A TLS handshake marks the onset of TLS communication between the client and the server. During a TLS handshake, both the parties exchange messages verify the identity, agree on cipher suite and TLS version to be used in communication, and finally establish the session’s keys.

Fundamentally, the SSL handshake is nothing but a conversation between two parties (client and server) wanting to accomplish the same purpose – securing the communication with the help of symmetric encryption.

Imagine this SSL Handshake Process as a dialog between the two.

Let’s see how it goes.

Client: “Hello there. I want to establish secure communication between the two of us. Here are my cipher suits and compatible SSL/TLS version.”

Server: “Hello Client. I have checked your cipher suits and SSL/TLS version. I think we’re good to go ahead. Here are my certificate file and my public key. Check ‘em out.”

Client: “Let me verify your certificate. (After a while) Okay, it seems fine, but I need to verify your private key. What I’ll do is, I will generate and encrypt a pre-master (shared secret key) key using your public key. Decrypt it using your private key and we’ll use thing master key to encrypt and decrypt the information”

Server: “Done.”

[Now that both the parties know who they’re talking to, the information transferred between them will be secured using the master-key. Keep in mind that once the verification part is over, the encryption takes place through the master-key only. This is symmetric encryption.]

Client: “I’m sending you this sample message to verify that our master-key works. Send me the decrypted version of this message. If it works, our data is in safe hands.”

Server: “Yeah, it works. I think we’ve accomplished what we were looking for.”

ssl tls handshake process - how does tls work

From now on, every bit of data transmitted back and forth between client and server will be encrypted. This is called the SSL/TLS handshake process.

**26. Benefits of jwt token**

There are benefits to using JWTs when compared to simple web tokens (SWTs) and Security Assertion Markup Language (SAML) tokens. More compact: JSON is less verbose than XML, so when it is encoded, a JWT is smaller than a SAML token. This makes JWT a good choice to be passed in HTML and HTTP environments.

a lot of web-related info can be found in a similar post here: Token Authentication vs. Cookies; I would like to call out some "architectural" differences:

JWTs are a standardized container format to encode user and client related information in a secure way using "claims" (whereas cookie contents and signing/encryption are not standardized)

JWTs are not restricted to present session-like information about the authenticated user itself; they can also be used to delegate access to clients that act on behalf of the user

JWTs allow for a more granular access model than cookies because JWTs can be limited in "scope" (what they allow the client to do) as well as time

Pros

No Database Table : This implies fewer DB queries, which implies faster response time. In case you are using paid services like DynamoDb that charge per query basis, JWT might reduce the costs marginally.

But these can be resolved using tools like Redis in case of sessions

Simpler to use if careful : If your architecture doesn’t user client Sessions and your security basics are clear, the development time in case of JWT is faster using the existing libraries.

Used across services : You can have one authorization server that deals with the Login/Registration and generates the token, all the subsequent requests will need not have to go to the authorization server as the only the Auth-server will have have the private key, and rest of the severs will have the public-key to verify the signature.

This is really useful in case of corporate systems where in the authorization server is in a secure environment. e.g. a user needs to be connected to the intranet to login but once done, the public servers can verify and proceed on.

Similar setup can be used for OAuth implementation.

The best part is that there is no connection between the the auth-server and the rest of the servers other than the pre-defined public key.

Cons

Compromised Secret Key : The best and the worst thing about JWT is that it relies on just one Key. Consider that the Key is leaked by a careless or a rogue developer/administrator, the whole system is compromised!

The attacker(who has access to the Key) can easily access all user data if he has the user-id which can be easily acquired.

The only way to recover from this point is to generate a new Key(Key-pair) that will be used across systems here on. This would me all the existing client tokens are invalidated and each user would have to login again. Image one day 100% of Facebook users will be logged out.

Well you might wonder, why is the same not possible if the developer/administrator leaks the Session table?

It is possible, but it is related to the practicality of the situation. Remember, most of the online breaches are done with social engineering than complicated technical hacks.

a) Practically it is really difficult to leak the whole table. In case of a single key, the admin just has pretend to take a photo of his friend in the office aaaaand the secret is on Reddit the next morning you fire him.

b) As well consider the OpenSSL Heartbleed bug. It is really easy to extract the secret key from just a couple of memory dumps with a simple string match script.

Cannot manage client from the server: We had several cases where we wanted the users at HelpTap to logout by cleaning up the cookies, but we cannot ask them to do so every time.

As well consider the case that a user’s mobile is stolen, and he wants to logout of all existing sessions(e.g. Gmail’s logout other sessions feature). Well its not possible in case of JWT.

In our case it used to be rogue users. We needed to log them out. Well, in case of of HelpTap it was quite easy as we just had to delete the session tokens. There was no way to do the same in case of Bottr cause we used JWT in that case.

You might argue, why not just delete the existing user-id from the table… But doing so means to create multiple dangling pointers and no one likes dangling pointers in a No-SQL database.

Cannot push Messages to clients (Identifying clients from server) : As we have no record about the logged-in clients on the DB end, we cannot push messages to all the clients.

In HelpTap we implemented a chatting platform wherein the client polls the server for new messages. Each client has an AWS SQS queue to itself where we push any new messages. In case of JWT this would not have been possible as identifying each client per user is not possible.

One can use the device ID but not all clients have a device ID, as well that would mean creating another table that is parallel to the Session table

This point overlaps point 2

Crypto-algo can be deprecated: JWT relies completely on the Signing algorithm. Now, though it is not frequent, but in the past many Encryption/Signing algorithms have been deprecated.

This article shows how you can crack the Wifi password of a WEP Encrypted Wifi which was the most common type of encryption not more than a year ago. The hack was based on the weakness of the crypto algorithm. So, in case of JWT, if such a thing happens, yet again, every user on the platform will have to login again.

Yet again one will have to wait till all the JWT libraries update with the latest crypto-algo.

Data Overhead : The size of the JWT token will be more than that of a normal Session token. The more data you add in the JWT token, the longer it gets linearly. Remember, each request needs the token in it for request verification. So say, a 1 KB JWT token implies each request will have 1KB over-head upload which is really bad in cases of low speed net connectivity.

In case of bad developer, some one might put more data in the JSON and that would increase the length. The length of the sessions tokens can be as small it can be and still be secure. e.g. the possible combinations for just a 5 letter alphanumeric session string is almost 1 billion combinations (62⁵)

Complicated to understand: JWT uses cryptographic Signature algorithms to verify the data and get the user-id from the token. Understanding the Signing Algo in itself requires basics of cryptography. So, in case if the developer is not completely educated s/he might introduce security loopholes in the system. My co-worker was surprised when I decoded the JWT token without using the secret key. He expected that the whole token was an encrypted one.

I came across a website that stored the whole user object in the JWT token. This included the user’s password hash.

Sessions tokens are pretty straightforward to understand and such issues can be easily avoided.

As JWT is a fairly new concept, one might not find the libraries in all the languages out there. Adding to it, neither JWT nor Sessions solve the CSRF or XSS issues, as it completely depends on how you send the data.

TLDR; JWT is fast(development) though less customizable, risky, slightly complicated to understand

Topic:

SpringBoot

Pipeline

Docker

CaaS

Good to have:

Angular

Microservices