Core JAVA

26 October 2021
12:58

What is Annotations?

What is module-info.java in project creation?

```
Finding missing values in array 1-20
Difference between throws and throw and examples
Mutable v/s immutable and examples
immutable calss creation steps
String Operations:
        finding substrings
        count of words
        removing special characters from string
String pool concept
checked and unchecked exceptions with examples
User Defined Exception / Custom Exception Example
NoSuchmethod/class Exception  when it will come?
How to handle NullPointer Exception
Access Specifiers or modifiers
Designe Patterns
Singleton pattern and example
Threads:
=======
overriding start() in threads
calling run() directly instead of start() on thread , what will
happen?
Difference between soap and rest

method overloading and overriding : abc(int)  , abc(Integer)
method overloading and overriding : return type : int abc()  and
Integer abc()
============================
        abc(int)  , abc(Integer)
main();
                abc(10);

        Which Method will be invoked?
============
============
What is Generics ?
Reading values from Hashmap
How to Copy Array to ArrayList?
Set v/s List
Comparable v/s Comparator?
Hashmap v/s Hashtable
Hashmap v/s Treemap
Hashset v/s Treeset
ArrayList v/s LinkedList
ArrayList v/s Vector
List<Object> list = new ArrayList<String>();   ??????????????
is it okay?
```

ANJISEK

```
if hashmap has 3 keys which having same hash code . then what
will be the size() hashmap.
emplyee is immutable and address is mutable then how to inject
address vallue.
StringUtils class
How to avoid NullPointer exception ?
Diff b/w heap and stack memory
what will happen if we store same key in hashmap?
list.add("Dilip");
list.add(2,"Singh");  what will happen?
Hibernate :
        save() v/s saveUpdate()
        update() v/s merge()

        SessionFactory v/s LocalSessionFactory

        integreation with Spring

        Hql v/s Criteria API

        mappings  : 1 to many : how to config
load() and get() : difference
        cache mechanism in hibernate
        Diff between build session factory and application
factory

Spring :
        Dependency injection : which is better either setter or
constructor
        Bean Life Cycle

        ClassA scope is Singleton
        ClassB scope is Prototye
        ClassA have Dependency of ClassB , what will hepen?

        Class is defined as abstract=false in xml bean
configuration, what will happen?
        Diff B/W Controller v/s RestController

        How to get ApplicationContext in Spring MVC?
        stereo thype annotaions in spring mvc
```

1. oops concept
2. what is abstraction how we achieve it?
3. Can we pass by reference. dfeault type of passing parameter :
   There is only call by value in java, not call by reference.
4. What happen if we override static method

   // As per overriding rules this should call to class Child static overridden method. Since static method cannot be overridden, it calls Parent static method . Method Hiding in child class
   Parent p = new Child();
   p.staticMethod();

5. can we call non-static method from static method and how

ACTISEK

160

The only way to call a non-static method from a static method is to have an instance of the class containing the non-static method. By definition, a non-static method is one that is called ON an instance of some class, whereas a static method belongs to the class itself.

6. what are methods of object class
7. diff between serialization and externalization
8. can we serialize static variable
9. comparable an compareto interface and implementation
10. how to implement immutable class
11. string builder/ string buffer

collection:

---

1. which data type is ideal for map key.
2. what needs to be done if we want to use custom object as key
3. what is hash collision
4. how java 8 improved performance in case of more hash collision:

Java 8 has come with the following **improvements/changes** of HashMap objects in case of high collisions.

- The alternative String hash function added in Java 7 has been removed.
- Buckets containing a large number of colliding keys will store their entries in a balanced tree instead of a linked list after certain threshold is reached.

Above changes ensure performance of O(log(n)) in worst case scenarios (hash function is not distributing keys properly) and O(1) with proper **hashCode()**.

How linked list is replaced with binary tree?

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

5. what is copyOnWrite hashmap/set/list
6. which collection type is best for searching and why
7. Dictonaory

multi-thread:

---
1. can we call run/call method ouside thread and how it will behave
2. why wait/notify/notifyall method associated with object class
3. can we call wait/notify/notifyall methods outside synchronization
   https://www.zdnet.com/article/calling-wait-notify-and-notifyall-within-a-non-synchronized-method/
4. what is class level locking and bucket level locking
5. what is countdown latch

design pattern:
---
1. singleton implementation
2. how to break singleton pattern
3. how to prevent it
4. Factory and AbstractFactory pattern in details
5. Builder pattern

Spring/Spring boot:
---
1. What are the benefits/disadvantages of using spring framework and springboot
2. What are the events generated when running springboot application
3. what are springboot annotation
4. benefits/features of using @service/@repository/@controller/@restcontroller
5. diff between @primary and @qualifier
6. centralize exception handling (@controlleradvice)
7. session scopes in spring
8. @transaction and its attributes
9. repository interfaces/classes(repository/curdrepository/pagingandsortingrepository/jparepository/simplejparepository)
10. how to configure multiple database/datasource in springboot application
11. sessions in jpa/hibernate
12. entity object represents whole table data or single row
13. bean initialization @lazy and eager initialization
14. how to register external bean in spring context
15. how to use propertise in class
16. how to configure external server in springboot

stream api:
---
1. advantage of stream api over collection
2. what is intermediate and terminal operations
3. how streams internally works
4. calculate min/min/sum/count distinct/grouping by/

Reduce
Diamond Problem

ABHISEK

Can we override mian method ?
Can we override static method?

- Please collect Employees Data
  10 employees
    - Eid
    - Ename
    - salry

1, Dilip ,90000

Store Above 10 emp objects int hashmap
Key : eid
Value : Employee

Please print every employs salary's are incremented with 10000

1, Dilip ,100000

Interview Programs

11 October 2022
20:17

String is Palindrome or not
Factorial of a Number
Duplicate values in An Array
Reverse of String
Count Of Characters In a String
Fibonacci Series
Reverse Of a String
Square Root Of Number
Sum of an Array Values
Print the elements of an array in reverse order
Print the elements of an array present on even position
Print the elements of an array present on odd position
Most Repeated character from String
Most Repeated number from int Array
Print the largest element in an array
Find 2nd Largest Number in an array
Print the smallest element in an array

Checking for prime number. Program to verify whether a given number is a prime or composite.

ABHISEK

https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/

Access Specifiers

17 March 2022
07:15

In Java, methods and data members can be encapsulated by the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

1) **private** (accessible within the class where defined)
2) **default** or package-private (when no access specifier is specified)
3) **protected** (accessible only to classes that subclass your class directly within the current or different package)
4) **public** (accessible from any class)

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.
1) public
2) default (when no access specifier is specified)

Note: Nested interfaces and classes can have all access specifiers.
Note: We cannot declare class/interface with private or protected access specifiers.

Access specifier of methods in interfaces`

In Java, all methods in an interface are public even if we do not specify public with method names. Also, data fields are public static final even if we do not mention it with fields names. Therefore, data fields must be initialized.
Consider the following example, x is by default public static final and foo() is public even if there are no specifiers.

```
interface Test {
  int x = 10;  // x is public static final and must be
initialized here
  void foo();  // foo() is public
}
```

ABHISEK

Strings

18 November 2021

19:55

1. Find every character count from String
2. print the reverse of the string

Threads

22 December 2021
13:49

What is Deadlock ? How do we rectify it.?

In Java, **deadlock** is a part of **multithreading**. The multithreading environment allows us to run multiple threads simultaneously for multitasking. Sometimes the threads find themselves in the waiting state, forever that is a **deadlock** situation.
The **deadlock** is a situation when two or more threads try to access the same object that is acquired by another thread. Since the threads wait for releasing the object, the condition is known as **deadlock**. The situation arises with more than two threads.

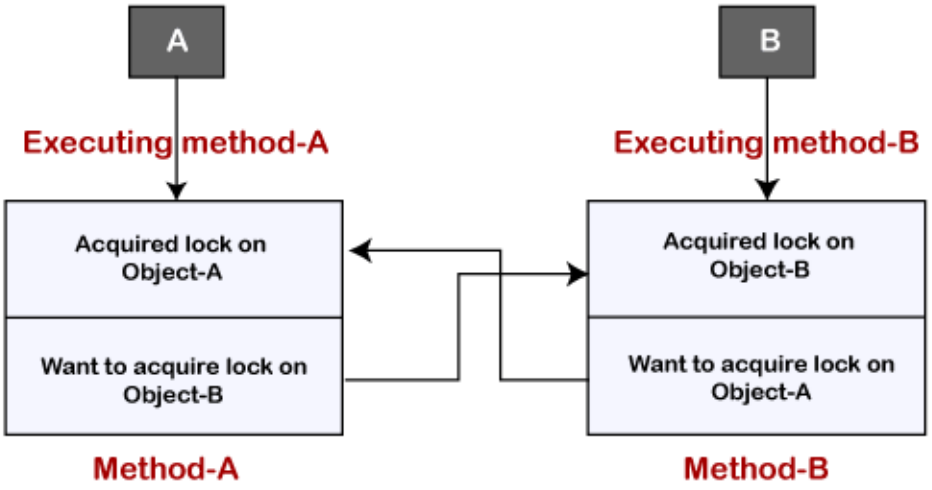
Example

Suppose, there are two threads A and B. The thread A and B acquired the lock of Object-A and Object-B, respectively. Assume that thread A executing method A and wants to acquire the lock on Object-B, while thread B is already acquired a lock on Object-B.
On the other hand, thread B also tries to acquire a lock on Object-A, while thread A is acquired a lock on Object-A. In such a situation both threads will not complete their execution and wait for releasing the lock. The situation is known as, **deadlock**.

How to detect deadlock in Java?

There are following ways to detect a deadlock:

- First, we look and understand the code if we found nested synchronized block or trying to get a lock on a different object or calling a synchronized method from other synchronized method, these reason leads to a deadlock situation.



Deadlock condition in Java

How to avoid deadlock in Java?

Although it is not possible to avoid deadlock condition but we can avoid it by using the following ways:

- **Avoid Unnecessary Locks:** We should use locks only for those members on which it is required. Unnecessary use of locks leads to a deadlock situation. We recommend you to use a **lock-free** data structure. If possible, keep your code free form locks. For example, instead of using synchronized **ArrayList** use the **ConcurrentLinkedQueue**.
- **Avoid Nested Locks:** Another way to avoid deadlock is to avoid giving a lock to multiple threads if we have already provided a lock to one thread. Since we must avoid allocating a lock to multiple threads.
- **Using Thread.join() Method:** You can get a deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with the maximum time you want to wait for the thread to finish.
- **Use Lock Ordering:** Always assign a numeric value to each lock. Before acquiring the lock with a higher numeric value, acquire the locks with a lower numeric value.
- **Lock Time-out:** We can also specify the time for a thread to acquire a lock. If a thread does not acquire a lock, the thread must wait for a specific time before retrying to acquire a lock.

What is Race Condition and How we detect and fix it?

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the

threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act".

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"
// If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
}
```

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.
In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
            // Therefore y = 10
}
// release lock for x
```

https://stackoverflow.com/questions/34510/what-is-a-race-condition

Wait(), notify(), notifyAll() why these are part of Object but not Thread.

In the Java language, you wait() on a particular instance of an Object -- a monitor assigned to that object to be precise. If you want to send a signal to a single thread that is waiting on that specific object instance then you call notify() on that object. If you want to send a signal to all threads that are waiting on that object instance, you use notifyAll() on that object.

If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.

In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the object that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is

thread2 which is waiting for the resource -- they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.

In Java, we then use lock objects as synchronization, mutex, and communication points between threads. We synchronize on a lock object to get mutex access to an important code block and to synchronize memory. We wait on a lock object if we are waiting for some condition to change -- some resource to become available. We notify on an object if we want to awaken sleeping threads.

The threads can communicate with each other through wait(), notify() and notifyAll() methods in Java. These are final methods defined in the Object class and can be called only from within a synchronized context. The wait() method causes the current thread to wait until another thread invokes the notify() or notifyAll() methods for that object. The notify() method wakes up a single thread that is waiting on that object's monitor. The notifyAll() method wakes up all threads that are waiting on that object's monitor. A thread waits on an object's monitor by calling one of the wait() method. These methods can throw IllegalMonitorStateException if the current thread is not the owner of the object's monitor.

public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()

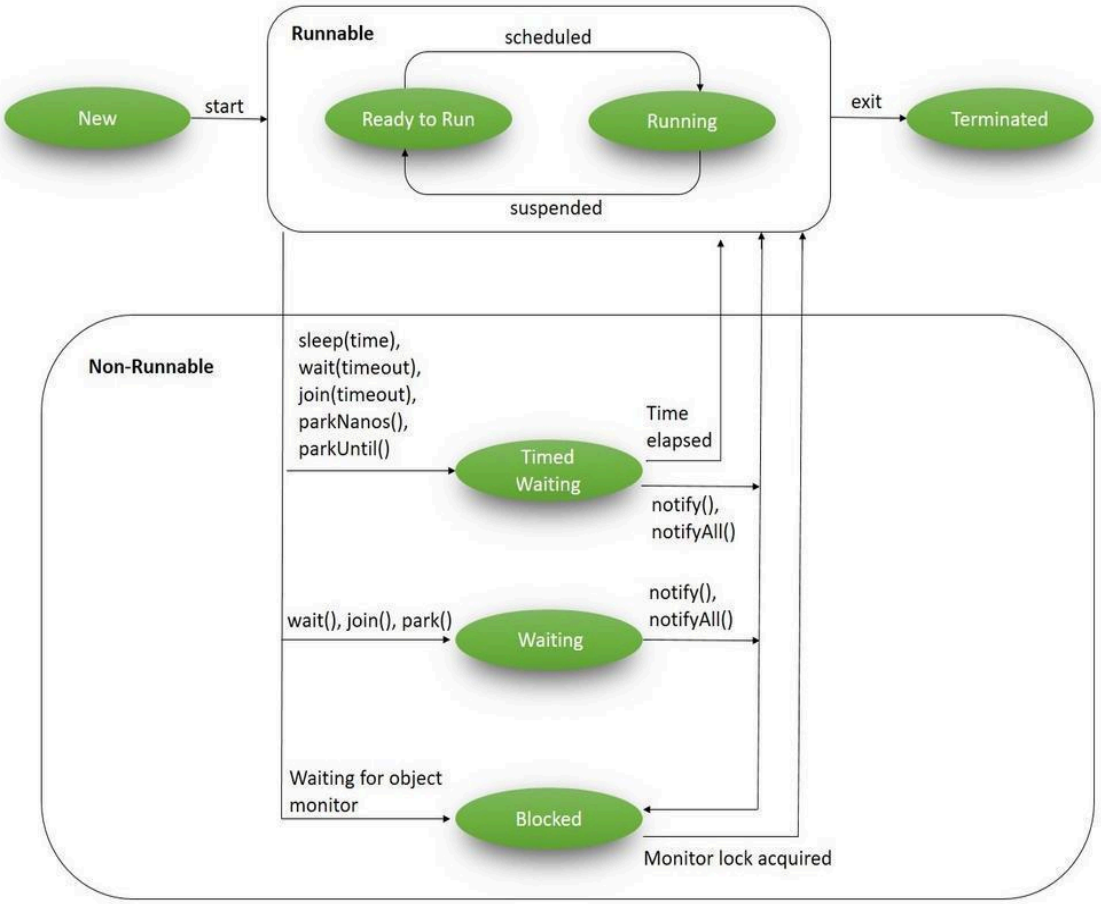
https://www.tutorialspoint.com/importance-of-wait-notify-and-notifyall-methods-in-java

Thread life Cycle

A thread lies only in one of the shown states at any instant:

- New
- Runnable
- Blocked
- Waiting
- Timed Waiting
- Terminated

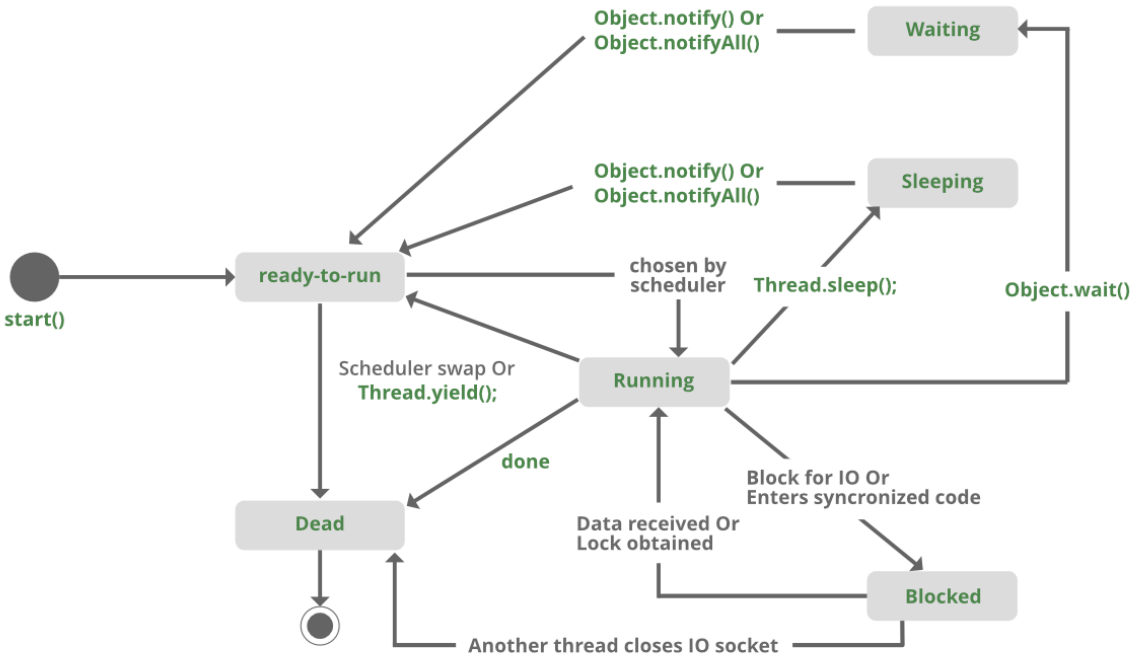
ABHISE



Difference between wait and sleep in Java

Sleep(): This Method is used to pause the execution of current thread for a specified time in Milliseconds. Here, Thread does not lose its ownership of the monitor and resume's it's execution

Wait(): This method is defined in object class. It tells the calling thread (a.k.a Current Thread) to wait until another thread invoke's the notify() or notifyAll() method for this object, The thread waits until it reobtains the ownership of the monitor and Resume's Execution.



What is Callable Interface?

Data Structures

16 December 2021
09:13

Time complexities of different data structures

Time Complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input. In other words, the time complexity is how long a program takes to process a given input. The efficiency of an algorithm depends on two parameters:

- Time Complexity
- Space Complexity

Time Complexity: It is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time took also depends on some external factors like the compiler used, processor's speed, etc.

Space Complexity: It is the total memory space required by the program for its execution.

Average time complexity of different data structures for different operations

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)

Exception Handling

11 December 2021
15:33

Exception Handling with Method Overriding

When Exception handling is involved with Method overriding, ambiguity occurs. The compiler gets confused as to which definition is to be followed.

Types of problems:

There are two types of problems associated with it which are as follows:

Problem 1: If The SuperClass doesn't declare an exception
Problem 2: If The SuperClass declares an exception

Problem 1: If The SuperClass doesn't declare an exception

In this problem, two cases that will arise are as follows:

Case 1: If SuperClass doesn't declare any exception and subclass declare checked exception
Case 2: If SuperClass doesn't declare any exception and SubClass declare Unchecked exception

Case 1: If SuperClass doesn't declare any exception and subclass declare checked exception.

```
package com.practice.exception.handling;

import java.io.IOException;

//Java Program to Illustrate Exception Handling with Method Overriding
```

ABHISEK

```
//Where SuperClass does not declare any exception and
//subclass declare checked exception

class SuperClass{

	// SuperClass doesn't declare any exception
	public void methodOne() {
		System.out.println("SuperClass : methodOne");
	}

}

class SubClass extends SuperClass{

	// declaring Checked Exception IOException
	public void methodOne() throws IOException{
		System.out.println("SubClass : methodOne");
	}

}

public class SuperClassNoExceptionSubClassCheckedException {

	public static void main(String[] args) {
		SuperClass s = new SubClass();
		s.methodOne();
	}

}
```

Result :

```
	Exception in thread "main" java.lang.Error: Unresolved compilation problem:
		Exception IOException is not compatible with throws clause in
		SuperClass.methodOne()
```

Case 2: If SuperClass doesn't declare any exception and SubClass declare Unchecked exception

```
package com.practice.exception.handling;

//Java Program to Illustrate Exception Handling with Method Overriding
//Where SuperClass does not declare any exception and
//subclass declare Un-checked exception

class ParentClass {

	// SuperClass doesn't declare any exception
	public void methodOne() {
		System.out.println("ParentClass : methodOne");
```

ABHISEK

```
	}

}

class ChildClass extends ParentClass {

	// declaring Un-Checked Exception IOException
	public void methodOne() throws ArrayIndexOutOfBoundsException {
		System.out.println("ChildClass : methodOne");
	}

}

public class SuperClassNoExceptionSubClassUnCheckedException {

	public static void main(String[] args) {
		ParentClass s = new ChildClass();
		s.methodOne();
	}

}
```

Result : ChildClass : methodOne

Problem 2: If The SuperClass declares an exception

The SuperClass declares an exception. In this problem 3 cases will arise as follows:

Case 1: If SuperClass declares an exception and SubClass declares exceptions other than the child exception of the SuperClass declared Exception.
Case 2: If SuperClass declares an exception and SubClass declares a child exception of the SuperClass declared Exception.
Case 3: If SuperClass declares an exception and SubClass declares without exception.

Case 1: If SuperClass declares an exception and SubClass declares exceptions other than the child exception of the SuperClass declared Exception.

```
package com.practice.exception.handling;

//If SuperClass declares an exception and SubClass declares exceptions
//other than the child exception of the SuperClass declared Exception.

class Parent {
	// SuperClass declare an exception
	public void methodOne() throws RuntimeException{
		System.out.println("Parent : methodOne");
	}

}
```

ABHISEK

```
class Child extends Parent {
	// SubClass declaring an exception not a child exception of RuntimeException
	public void methodOne()  throws Exception {
		System.out.println("Child : methodOne");
	}

}

public class SuperClassExceptionSubClassNoChildException {

	public static void main(String[] args) {
		Parent p = new Child();
		p.methodOne();
	}

}
```

Result : Compilation Error

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Exception Exception is not compatible with throws clause in Parent.methodOne()
```

Case 2: If SuperClass declares an exception and SubClass declares a child exception of the SuperClass declared Exception.

```
package com.practice.exception.handling;

//If SuperClass declares an exception and SubClass declares exceptions child exception of the
SuperClass declared Exception.

class Parent {
	// SuperClass declare an exception
	public void methodOne() throws RuntimeException{
		System.out.println("Parent : methodOne");
	}

}

class Child extends Parent {
	// SubClass declaring an exception  a child exception of RuntimeException

	public void methodOne()  throws ArrayIndexOutOfBoundsException {
		System.out.println("Child : methodOne");
	}

}

public class SuperClassExceptionSubClassChildException {
```

ABHISEK

```
	public static void main(String[] args) {
		Parent p = new Child();
		p.methodOne();
	}

}
```

Result : Child : methodOne

Case 3: If SuperClass declares an exception and SubClass declares without exception.

```
package com.practice.exception.handling;

//If SuperClass declares an exception and SubClass No Exception declared

class ParentTwo{
	// SuperClass declare an exception
	public void methodOne() throws RuntimeException {
		System.out.println("ParentTwo : methodOne");
	}

}

class ChildTwo extends ParentTwo {
	// SubClass not declaring an exception
	public void methodOne() {
		System.out.println("ChildTwo : methodOne");
	}

}

public class SuperClassExceptionSubClassNoException {

	public static void main(String[] args) {
		ParentTwo p = new ChildTwo();
		p.methodOne();
	}

}
```

Result : ChildTwo : methodOne

Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.
Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.
Rule 3: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.

ABHISEK

Question 1 : Parent class method contains unchecked exception, child class method contains checked exception , is this works?
As per Rule 3 , Compilation issue.

HISEK

Inheritance

07 December 2021
12:49

Association in Java

Association in Java defines the connection between two classes that are set up through their objects. Association manages **one-to-one, one-to-many**, and **many-to-many** relationships.

Types of Association

In Java, two types of **Association** are possible:

1. IS-A Association
2. HAS-A Association
    1. Aggregation
    2. Composition

1) IS-A Association

The IS-A Association is also referred to as Inheritance

2) HAS-A Association

The **HAS-A Association** is further classified into two parts, i.e., Aggregation and Composition. Let's understand the difference between both of them one by one.

1) Aggregation

In Java, the **Aggregation** association defines the **HAS-A** relationship. Aggregation follows the one-to-one or one-way relationship. If two entities are in the aggregation composition, and one entity fails due to some error, it will not affect the other entity.
Let's take the example of a toy and its battery. The battery belongs to a toy, and if the toy breaks and deletes from our database, the battery will still remaining in our database, and it may still be working. So in Aggregation, objects always have their own lifecycles when the ownership exists there.

2) Composition

A restricted form of the **Aggregation** where the entities are strongly dependent on each other. Unlike Aggregation, Composition
represents the **part-of** relationship. When there is an aggregation between two entities, the aggregate object can exist without the other entity, but in the case of Composition, the composed object can't exist. To learn more about Composition, click here
.
Let's take an example to understand the concept of **Composition**.
We create a class **Mobile** that contains variables, i.e., **name, ram** and **rom**. We also create a class **MobileStore** that has a reference to refer to the list of mobiles. A mobile store can have more than one mobile. So, if a mobile store is destroyed, then all mobiles within that particular mobile store will also be destroyed because mobiles cannot exist without a mobile store. The relationship between the mobile store and mobiles is Composition.

Exception Handling in Inheritance :

- If **SuperClass does not declare an exception**, then the **SubClass can only declare unchecked exceptions**, but not the checked exceptions.
- If **SuperClass declares an exception**, then the **SubClass can only declare the same or child exceptions of the exception declared by the SuperClass and any new Runtime Exceptions**, just not any new checked exceptions at the same level or higher.
- If **SuperClass declares an exception**, then the **SubClass can declare without exception**.

Miscellaneous

27 November 2021
14:34

1. Shallow Copy and Deep Copy in Java

Cloning is a process of creating a replica or copy of java object, clone method Java.lang.Object is used to create copy or replica of an object. java objects which implement Cloneable interface are eligible for using the clone method.

Creating Copy of Java Object

We can create a replica or copy of java object by
1. Creating a copy of object in a different memory location. This is called a Deep copy.
2. Creating a new reference that points to the same memory location. This is also called a Shallow copy.

Shallow Copy

The default implementation of the clone method creates a shallow copy of the source object, it means a new instance of type Object is created, it copies all the fields to a new instance and returns a new object of type 'Object'. This Object explicitly needs to be typecast in object type of source object.
This object will have an exact copy of all the fields of source object including the primitive type and object references. If the source object contains any references to other objects in field then in the new instance will have only references to those objects, a copy of those objects is not created. This means if we make changes in shallow copy then changes will get reflected in the source object. Both instances are not independent.
The clone method in Object class is protected in nature, so not all classes can use the clone() method. You need to implement Cloneable interface and override the clone method. If the Cloneable interface is not implemented then you will get CloneNotSupportedException.super.clone () will return shallow copy as per implementation in Object class.

ABHISEK

Code for Shallow Copy

```
1  package com.test;
2
3  class Department {
4      String empId;
5
6      String grade;
7
8      String designation;
```

```
9
10      public Department(String empId, String grade, String designation)
11  {
12          this.empId = empId;
13
14          this.grade = grade;
15
16          this.designation = designation;
17      }
18  }
19
20  class Employee implements Cloneable {
21      int id;
22
23      String name;
24
25      Department dept;
26
27      public Employee(int id, String name, Department dept) {
28          this.id = id;
29
30          this.name = name;
31
32          this.dept = dept;
33      }
34
35      // Default version of clone() method. It creates shallow copy of an
36  object.
37
38      protected Object clone() throws CloneNotSupportedException {
39          return super.clone();
40      }
41  }
42
43  public class ShallowCopyInJava {
44
45      public static void main(String[] args) {
46
47          Department dept1 = new Department ("1", "A", "AVP");
48
49          Employee emp1 = new Employee (111, "John", dept1);
50
51          Employee emp2 = null;
52
53          try {
54              // Creating a clone of emp1 and assigning it to emp2
55
56              emp2 = (Employee) emp1.clone();
57          } catch (CloneNotSupportedException e) {
58              e.printStackTrace();
59          }
60
61          // Printing the designation of 'emp1'
62
63          System.out.println(emp1.dept.designation); // Output : AVP
64
65          // Changing the designation of 'emp2'
```

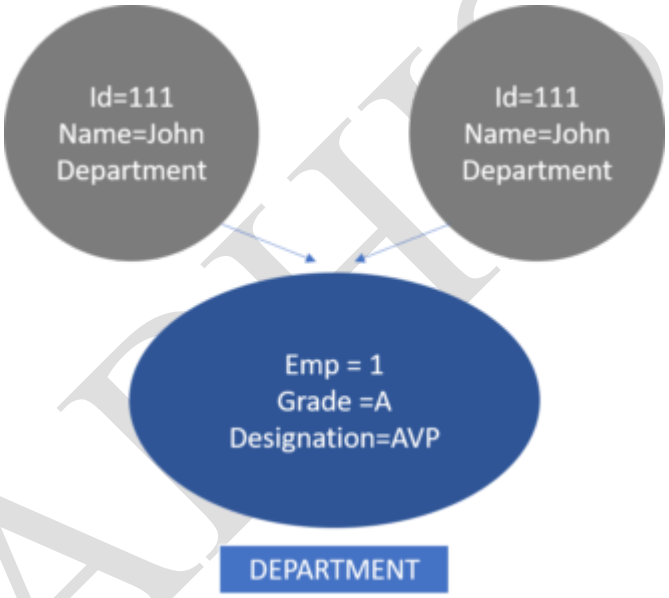
ABHISEK

```
66
67          emp2.dept.designation = "Director";
68
69          // This change will be reflected in original Employee 'emp1'
70
71          System.out.println(emp1.dept.designation); // Output : Director
        }
    }
```

Output:

```
<terminated> ShallowCopyInJava [Java Application] C:\Program Files\Java\jdk1.8.0_221\bin\javaw.exe (Oct 18, 2019, 12:33:08 AM)
AVP
Director
```

In the above example, we have an Employee class emp1 which has three class variable id (int), name (String) and department (Department).
We now cloned emp1 to emp2 to create a shallow copy, after that we changed designation using emp2 object and verified that the same changes got reflected in emp1 also.



Deep Copy

The deep copy of an object will have an exact copy of all the fields of source object like a shallow copy, but unlike sallow copy if the source object has any reference to object as fields, then a replica of the object is created by calling clone method. This means that both source and destination objects are independent of each other. Any change made in the cloned object will not impact the source object.

Code for Deep Copy

```
1  package com.test;
2
3  class Department implements Cloneable{
4      String empId;
5
6      String grade;
7
8      String designation;
9
10     public Department(String empId, String grade, String
11 designation) {
12         this.empId = empId;
13
14         this.grade = grade;
15
16         this.designation = designation;
17     }
18     //Default version of clone() method.
19     protected Object clone() throws
20 CloneNotSupportedException {
21         return super.clone();
22     }
23 }
24
25 class Employee implements Cloneable {
26     int id;
27
28     String name;
29
30     Department dept;
31
32     public Employee(int id, String name, Department dept) {
33         this.id = id;
34
35         this.name = name;
36
37         this.dept = dept;
38     }
39
40     // Overriding clone() method to create a deep copy of an
41 object.
42
43     protected Object clone() throws
44 CloneNotSupportedException {
45         Employee emp = (Employee) super.clone();
46
47         emp.dept = (Department) dept.clone();
48
49         return emp;
50     }
51 }
52
53 public class DeepCopyInJava {
54     public static void main(String[] args) {
```

ABHISEK

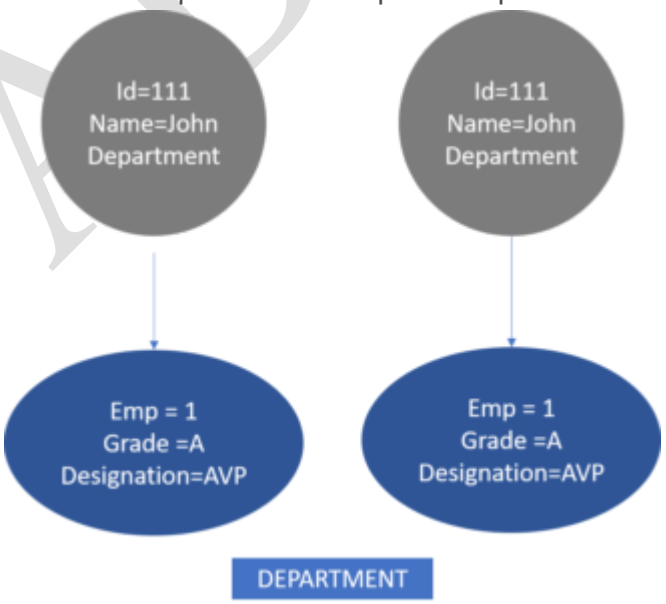
```
55          Department dept1 = new Department("1", "A", "AVP");
56
57          Employee emp1 = new Employee(111, "John", dept1);
58
59          Employee emp2 = null;
60
61          try {
62             // Creating a clone of emp1 and assigning it to emp2
63
64             emp2 = (Employee) emp1.clone();
65          } catch (CloneNotSupportedException e) {
66             e.printStackTrace();
67          }
68
69          // Printing the designation of 'emp1'
70
71          System.out.println(emp1.dept.designation); // Output : AVP
72
73          // Changing the designation of 'emp2'
74
75          emp2.dept.designation = "Director";
76
77          // This change will be reflected in original Employee 'emp1'

            System.out.println(emp1.dept.designation); // Output : AVP
        }
    }
```

Output:

```
<terminated> DeepCopyInJava [Java Application] C:\Program Files\Java\jdk1.8.0_221\bin\javaw.exe (Oct 18, 2019, 1:34:30 AM)
AVP
AVP
```

In the above example of Deep copy, unlike shallow copy, both source and destination objects are independent of each other. Any change made in emp2 will not impact emp1.



ABHISEK

Difference Between Shallow Copy and Deep Copy

Shallow Copy	Deep Copy
Cloned object and source object are not disjoint completely	Cloned objects and source objects are completely independent of each other.
Changes made in the cloned instance will impact the reference variable of the source object	Changes made in the cloned instance will not impact the reference variable of the source object.
The default version of the clone is the shallow copy	To create deep copy we need to override the clone method of Object class.
Shallow copy is preferred if class variables of the object are only primitive type as fields	A deep copy is preferred if the object's class variables have references to other objects as fields.
It is relatively fast	It is relatively slow.

ARHISEK

1. Composition Vs Aggregation Java

Composition

Composition is a "belongs-to" type of relationship. It means that one of the objects is a logically larger structure, which contains the other object. In other words, it's part or member of the other object.

Alternatively, **we often call it a "has-a" relationship** (as opposed to an "is-a" relationship, which is inheritance).

For example, a room belongs to a building, or in other words a building has a room. So basically, whether we call it "belongs-to" or "has-a" is only a matter of point of view.

Aggregation

Aggregation is also a “has-a” relationship. What distinguishes it from composition, that it doesn't involve owning. As a result, the lifecycles of the objects aren't tied: every one of them can exist independently of each other.

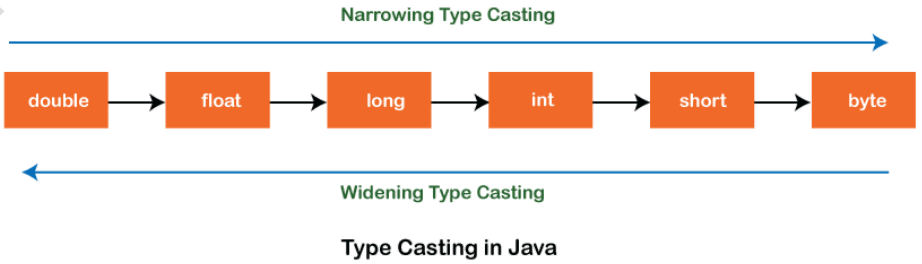
For example, a car and its wheels. **We can take off the wheels, and they'll still exist.** We can mount other (preexisting) wheels, or install these to another car and everything will work just fine.

Of course, a car without wheels or a detached wheel won't be as useful as a car with its wheels on. But that's why this relationship existed in the first place:
to **assemble the parts to a bigger construct, which is capable of more things than its parts**.

Since aggregation doesn't involve owning, **a member doesn't need to be tied to only one container**. For example, a triangle is made of segments. But triangles can share segments as their sides.

1. Type Casting (Widening and Narrowing) in Java

Type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

Types of Type Casting

1. Widening Type Casting

2. Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

1. Both data types must be compatible with each other.
2. The target type must be larger than the source type.

byte → **short** → **char** → **int** → **long** → **float** → **double**

Example :

```
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
```

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double → **float** → **long** → **int** → **char** → **short** → **byte**

Example:

```
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
```

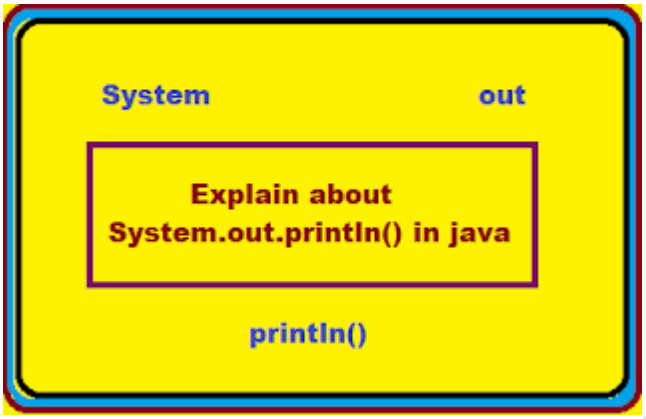
What is System.out.println in java?

What is System in java ?

The System is a predefined class. Which is present in java. lang package.

ABI SEK

1. The System class is a final class and do not provide any public constructors. Because of this all of the members are methods contained in this class are static in nature.
2. The System class also provides facilities like standard input, standard output, and error output Streams. it can't be instantiated.



Methods present in System Class :-

```
public final class java.lang.System {
    public static final java.io.InputStream in;
    public static final java.io.PrintStream out;
    public static final java.io.PrintStream err;
    public static void setIn(java.io.InputStream);
    public static void setOut(java.io.PrintStream);
    public static void setErr(java.io.PrintStream);
    public static java.io.Console console();
    public static java.nio.channels.Channel inheritedChannel() throws
java.io.IOException;
    public static void setSecurityManager(java.lang.SecurityManager);
    public static java.lang.SecurityManager getSecurityManager();
    public static native long currentTimeMillis();
    public static native long nanoTime();
    public static native void arraycopy(java.lang.Object, int, java.lang.Object, int,
int);
    public static native int identityHashCode(java.lang.Object);
    public static java.util.Properties getProperties();
    public static java.lang.String lineSeparator();
    public static void setProperties(java.util.Properties);
    public static java.lang.String getProperty(java.lang.String);
    public static java.lang.String getProperty(java.lang.String, java.lang.String);
    public static java.lang.String setProperty(java.lang.String, java.lang.String);
    public static java.lang.String clearProperty(java.lang.String);
    public static java.lang.String getenv(java.lang.String);
    public static java.util.Map<java.lang.String, java.lang.String> getenv();
    public static void exit(int);
    public static void gc();
    public static void runFinalization();
    public static void runFinalizersOnExit(boolean);
```

ABHISEK

```
    public static void load(java.lang.String);
    public static void loadLibrary(java.lang.String);
    public static native java.lang.String mapLibraryName(java.lang.String);
    static {};
}
```

What is Out in java ?

Out is a constant defined in System class but it is a reference of printStream class. and also Out is a Static field of System class.

What is print/println in java ?

print & println are non-static method defined in printStream Class .

1. As both the methods are object methods we call it outside class by the help of object.

Difference between print() and println() in java.

Print() :- It prints all the messages in a single line. There is no new line / break line after printing the message.

Println() :- First it print the messages then it break that line and create a new line. It is used for creating a new line.

What is System.out.println() in java ?

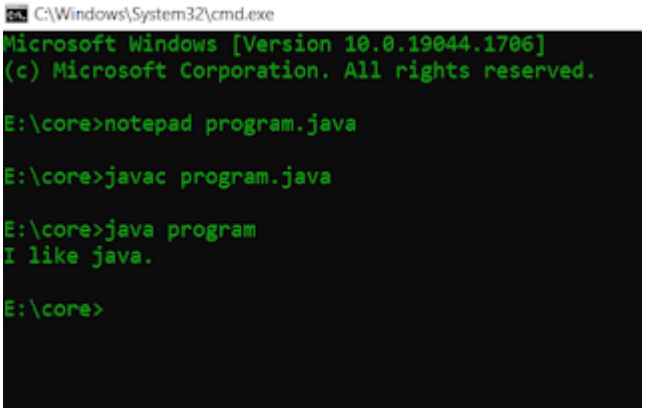
In java System.out.println() is a Statement. Which is used for print the argument passed to it.

Program for System.out.println() :-

```
public class program
 {
      public static void main(String args[])
      {
           System.out.println("I like java.");
      }
 }
```

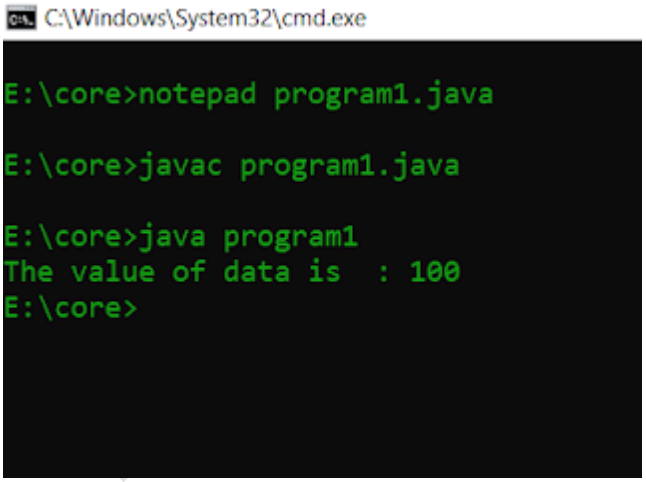
Output :-

ABHISEK



Program - 2

```
class datas
{
	public static final int data =100;
}
public class program1
{
	public static void main(String args[])
	{
		System.out.printf("The value of data is  : "+datas.data);
	}
}
```



File Operations

27 November 2021
14:28

Diff in filereader and fileinputstream

JAVA8

22 November 2021
16:29

1. **What is Optional class and it's methods?**

public final class Optional<T> extends Object

A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as orElse() (return a default value if value not present) and ifPresent() (execute a block of code if the value is present). This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.

empty()
Returns an empty Optional instance.

filter(**Predicate**<? super **T**> predicate)
If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.

get()
If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

ifPresent(**Consumer**<? super **T**> consumer)
If a value is present, invoke the specified consumer with the value, otherwise do nothing.

isPresent()
Return true if there is a value present, otherwise false.

ofNullable(T value)
Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.

orElse(**T** other)
Return the value if present, otherwise return other.

orElseThrow(**Supplier**<? extends X> exceptionSupplier)
Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.

1. **Difference Between map() And flatMap() In Java Stream**

ABHISEK

the Stream interface has a map() and flatmap() methods and both have intermediate stream operation and return another stream as method output. Both of the functions map() and flatMap are used for transformation and mapping operations. map() function produces one output for one input value, whereas flatMap() function produces an arbitrary no of values as output (ie zero or more than zero) for each input value.

The Syntax of the map() is represented as:

<R> Stream<R> map(Function<? super T, ? extends R> mapper)

The Syntax of the flatMap() is represented as:-

<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

map() can be used where we have to map the elements of a particular collection to a certain function, and then we need to return the stream which contains the updated results.
Example: Multiplying All the elements of the list by 3 and returning the updated list.

flatMap() can be used where we have to flatten or transform out the string, as we cannot flatten our string using map().
Example: Getting the 1st Character of all the String present in a List of Strings and returning the result in form of a stream.
Difference Between map() and flatmap()

map()	flatMap()
One-to-one mapping occurs in map().	One too many mapping occurs in flatMap().
The function passed to map() operation returns a single value for a single input.	The function you pass to flatmap() operation returns an arbitrary number of values as the output.
Only perform the mapping.	Perform mapping as well as flattening.
Produce a stream of value.	Produce a stream of stream value.
map() is used only for transformation.	flatMap() is used both for transformation and mapping.

2. **Functional Interfaces and Existing Functional interfaces.**

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Package java.util.function Description

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

ABHISEK

```
 // Assignment context
     Predicate<String> p = String::isEmpty;
// Method invocation context
     stream.filter(e -> e.getSize() > 10)...
// Cast context
     stream.map((ToIntFunction) e -> e.getSize())...
```

The interfaces in this package are general purpose functional interfaces used by the JDK, and are available to be used by user code as well. While they do not identify a complete set of function shapes to which lambda expressions might be adapted, they provide enough to cover common requirements. Other functional interfaces provided for specific purposes, such as FileFilter, are defined in the packages where they are used.
The interfaces in this package are annotated with FunctionalInterface. This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.
Functional interfaces often represent abstract concepts like functions, actions, or predicates. In documenting functional interfaces, or referring to variables typed as functional interfaces, it is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object". When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...", this is understood to mean a *non-null* reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.
The functional interfaces in this package follow an extensible naming convention, as follows:

- There are several basic function shapes, including Function (unary function from T to R), Consumer (unary function from T to void), Predicate (unary function from T to boolean), and Supplier (nilary function to R).
- Function shapes have a natural arity based on how they are most commonly used. The basic shapes can be modified by an arity prefix to indicate a different arity, such as BiFunction (binary function from T and U to R).
- There are additional derived function shapes which extend the basic function shapes, including UnaryOperator (extends Function) and BinaryOperator (extends BiFunction).
- Type parameters of functional interfaces can be specialized to primitives with additional type prefixes. To specialize the return type for a type that has both generic return type and generic arguments, we prefix ToXxx, as in ToIntFunction. Otherwise, type arguments are specialized left-to-right, as in DoubleConsumer or ObjIntConsumer. (The type prefix Obj is used to indicate that we don't want to specialize this parameter, but want to move on to the next parameter, as in ObjIntConsumer.) These schemes can be combined, as in IntToDoubleFunction.
- If there are specialization prefixes for all arguments, the arity prefix may be left out (as in ObjIntConsumer).

What is Method Reference?

Answer: In Java 8, a new feature was introduced known as Method Reference. This is used to refer to the method of functional interface. It can be used to replace Lambda Expression while referring to a method.
For Example: If the Lambda Expression looks like

ABHISHEK

num -> System.out.println(num)

Then the corresponding Method Reference would be,

System.out::println

where “::” is an operator that distinguishes class name from the method name.

Optional Class

21 September 2022
19:20

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

Method References

17 March 2022
07:54

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

ABHISEK

Lambda Expressions

17 March 2022
07:54

It provides a clear and concise way to represent one method interface using an expression.
Lambda expressions basically express instances of functional interfaces

Enable to treat functionality as a method argument, or code as data.

Java lambda expression is consisted of three components.
1) Argument-list: It can be empty or non-empty as well.
2) Arrow-token: It is used to link arguments-list and body of expression.
3) Body: It contains expressions and statements for lambda expression.

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

2nd Example : With return/ Without return keyword

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample6 {
    public static void main(String[] args) {

        // Lambda expression without return keyword.
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Lambda expression with return keyword.
```

ABHISEK

```
        Addable ad2=(int a,int b)->{
                        return (a+b);
                        };
        System.out.println(ad2.add(100,200));
    }
}
```

Java Lambda Expression Example: Filter Collection Data

```
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class LambdaExpressionExample11{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();
        list.add(new Product(1,"Samsung A5",17000f));
        list.add(new Product(3,"Iphone 6S",65000f));
        list.add(new Product(2,"Sony Xperia",25000f));

        // using lambda to filter data
        Stream<Product> filtered_data = list.stream().filter(p -> p.pric
e > 20000);

        // using lambda to iterate through collection
        filtered_data.forEach(
                product -> System.out.println(product.name+": "+produc
t.price)
        );
    }
}
```

ABHISEK

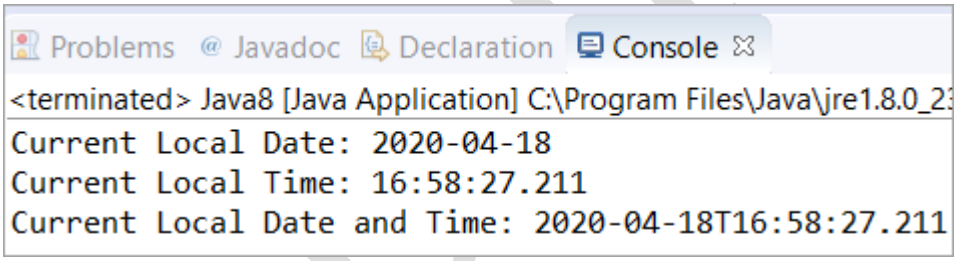
Date And Time

19 December 2021
12:26

How will you get the current date and time using Java 8 Date and Time API?
Answer: The below program is written with the help of the new API introduced in Java 8. We have made use of LocalDate, LocalTime, and LocalDateTime API to get the current date and time.
In the first and second print statement, we have retrieved the current date and time from the system clock with the time-zone set as default. In the third print statement, we have used LocalDateTime API which will print both date and time.

```
class Java8 {
   public static void main(String[] args) {
      System.out.println("Current Local Date: " + java.time.LocalDate.now());
      //Used LocalDate API to get the date
      System.out.println("Current Local Time: " + java.time.LocalTime.now());
      //Used LocalTime API to get the time
      System.out.println("Current Local Date and Time: " + java.time.LocalDateTime.now());
      //Used LocalDateTime API to get both date and time
   }
}
```

Output:



Streams

25 November 2021
08:11

https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/

What is the difference between intermediate and terminal operations on Stream?

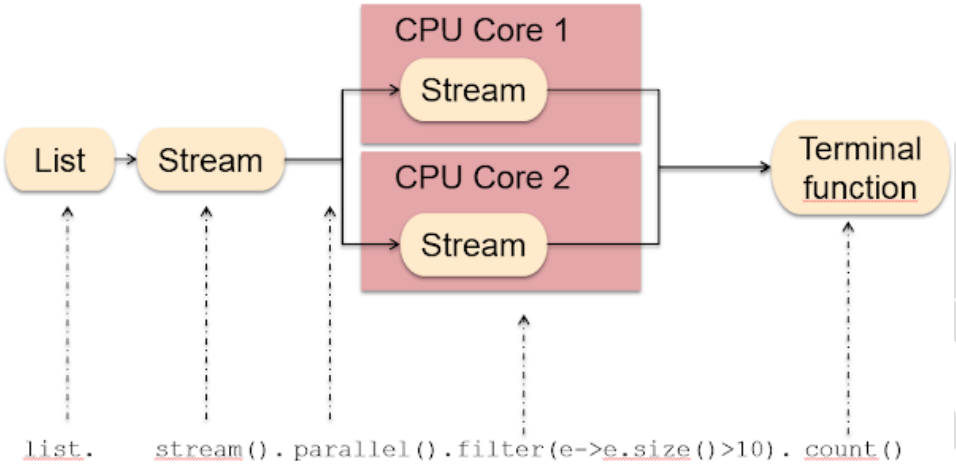
The intermediate Stream operation returns another Stream, which means you can further call other methods of Stream class to compose a pipeline.

For example after calling `map()` or `flatMap()` you can still call `filter()` method on `Stream`.

ABHISEK

On the other hand, the terminal operation produces a result other than Streams like a value or a `Collection`.

Once a terminal method like `forEach()` or `collect()` is called, you cannot call any other method of Stream or reuse the Stream.



What does the peek() method do? When should you use it?

The `peek()` method of Stream class allows you to see through a Stream pipeline. You can peek through each step and print meaningful messages on the console. It's generally used for debugging issues related to lambda expression and Stream processing.

This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline".

Example :

```
Stream.of("one", "two", "three", "four").filter(e -> e.length() > 3)
                .peek(e -> System.out.println("Filtered value: " + e))
                .map(String::toUpperCase).peek(e -> System.out.println("Mapped value: " + e))
                .collect(Collectors.toList());
```

Output :

```
Filtered value: three
Mapped value: THREE
Filtered value: four
Mapped value: FOUR
```

What do you mean by saying Stream is lazy?

When we say Stream is lazy, we mean that most of the methods are defined on Java `.util.stream.Stream` class is lazy i.e. they will not work by just including them on the Stream pipeline.

They only work when you call a terminal method on the Stream and finish as soon as they find the data they are looking for rather than scanning through the whole set of data.

What is a Predicate interface?

A Predicate is a functional interface that represents a function, which takes an Object and returns a boolean. It is used in several Stream methods like filter(), which uses `Predicate` to filter unwanted elements.

here is how a Predicate function looks like:

```
Public boolean test(T object){
  return boolean;
}
```

You can see it just has one `test()` method which takes an object and returns a boolean. The method is used to test a condition if it passes; it returns true otherwise false.

What are Supplier and Consumer Functional interface?

The `Supplier` is a functional interface that returns an object. It's similar to the factory method or `new()`, which returns an object.

The `Supplier` has `get()` functional method, which doesn't take any argument and return an object of type T. This means you can use it anytime you need an object.

Since it is a functional interface, you can also use it as the assignment target for a lambda expression or method reference.

A `Consumer` is also a functional interface in JDK 8, which represents an operation that accepts a single input argument and returns no result.

Unlike other functional interfaces, `Consumer` is expected to operate via side-effects. The functional method of `Consumer` is `accept(T t)`, and because it's a functional interface, you can use it as the assignment target for a lambda expression or method interface in Java 8.

What is the parallel Stream? How can you get a parallel stream from a List?

A parallel stream can parallel execute stream processing tasks. For example, if you have a parallel stream of 1 million orders and you are looking for orders worth more than 1 million, then you can use a filter to do that.

Unlike sequential Stream, the parallel Stream can launch multiple threads to search for those orders on the different parts of the Stream and then combine the result.

List<String> alpha2 = getData(); alpha2.parallelStream().forEach(System.out::println);

ABHISEK

Streams Notes

04 September 2022
11:54

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections. For example:

```
int sum = widgets.stream()
              .filter(b -> b.getColor() == RED)
              .mapToInt(b -> b.getWeight())
              .sum();
```

Here we use widgets, a Collection<Widget>, as a source for a stream, and then perform a filter-map-reduce on the stream to obtain the sum of the weights of the red widgets. (Summation is an example of a reduction operation.)

The key abstraction introduced in this package is *stream*. The classes Stream, IntStream, LongStream, and DoubleStream are streams over objects and the primitive int, long and double types. Streams differ from collections in several ways:

- No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
- Functional in nature. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
- Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
- Consumable. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Streams can be obtained in a number of ways. Some examples include:

- From a Collection via the stream() and parallelStream() methods;
- From an array via Arrays.stream(Object[]);
- From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);
- The lines of a file can be obtained from BufferedReader.lines();

- Streams of file paths can be obtained from methods in Files;
- Streams of random numbers can be obtained from Random.ints();
- Numerous other stream-bearing methods in the JDK, including BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().

Additional stream sources can be provided by third-party libraries using these techniques.

Stream operations and pipelines

Stream operations are divided into *intermediate* and *terminal* operations, and are combined to form *stream pipelines*. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

Intermediate operations return a new stream. They are always *lazy*; executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are *eager*, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.

Processing streams lazily allows for significant efficiencies; in a pipeline such as the filter-map-sum example above, filtering, mapping, and summing can be fused into a single pass on the data, with minimal intermediate state. Laziness also allows avoiding examining all the data when it is not necessary; for operations such as "find the first string longer than 1000 characters", it is only necessary to examine just enough strings to find one that has the desired characteristics without examining all of the strings available from the source. (This behavior becomes even more important when the input stream is infinite and not merely large.)

Intermediate operations are further divided into stateless and stateful operations. Stateless operations, such as filter and map, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

Stateful operations may need to process the entire input before producing a result. For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream. As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data. Pipelines containing exclusively stateless intermediate operations can be processed in a single pass, whether sequential or parallel, with minimal data buffering.

Further, some operations are deemed short-circuiting operations. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. A

ABHISEK

terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time. Having a short-circuiting operation in the pipeline is a necessary, but not sufficient, condition for the processing of an infinite stream to terminate normally in finite time.

Parallelism

Processing elements with an explicit for-loop is inherently serial. Streams facilitate parallel execution by reframing the computation as a pipeline of aggregate operations, rather than as imperative operations on each individual element. All streams operations can execute either in serial or in parallel. The stream implementations in the JDK create serial streams unless parallelism is explicitly requested. For example, Collection has methods Collection.stream() and Collection.parallelStream(), which produce sequential and parallel streams respectively; other stream-bearing methods such as IntStream.range(int, int) produce sequential streams but these streams can be efficiently parallelized by invoking their BaseStream.parallel() method. To execute the prior "sum of weights of widgets" query in parallel, we would do:

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "parallelStream()" instead of "stream()". When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the orientation of the stream on which it is invoked. Whether a stream will execute in serial or parallel can be determined with the isParallel() method, and the orientation of a stream can be modified with the BaseStream.sequential() and BaseStream.parallel() operations. When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the mode of the stream on which it is invoked.

Except for operations identified as explicitly nondeterministic, such as findAny(), whether a stream executes sequentially or in parallel should not change the result of the computation.

Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these behavioral parameters must be non-interfering, and in most cases must be stateless. Such parameters are always instances of a functional interface such as Function, and are often lambda expressions or method references.

Non-interference

Streams enable you to execute possibly-parallel aggregate operations over a variety of data sources, including even non-thread-safe collections such as ArrayList. This is possible only if we can prevent *interference* with the data source during the execution of a stream pipeline. Except for the escape-hatch operations iterator() and spliterator(), execution begins when the terminal operation is invoked, and ends when the terminal operation completes. For most data sources, preventing interference means ensuring that the data source is *not modified at all* during the execution of the stream pipeline. The notable exception to this are streams whose sources are concurrent collections, which are specifically designed to handle concurrent modification. Concurrent stream sources are those whose Spliterator reports the CONCURRENT characteristic.

Accordingly, behavioral parameters in stream pipelines whose source might not be concurrent should never modify the stream's data source. A behavioral parameter is said to interfere with a non-concurrent data source if it modifies, or causes to be modified, the stream's data source. The

ABHISEK

need for non-interference applies to all pipelines, not just parallel ones. Unless the stream source is concurrent, modifying a stream's data source during execution of a stream pipeline can cause exceptions, incorrect answers, or nonconformant behavior. For well-behaved stream sources, the source can be modified before the terminal operation commences and those modifications will be reflected in the covered elements. For example, consider the following code:

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
    Stream<String> sl = l.stream();
    l.add("three");
    String s = sl.collect(joining(" "));
```

ABHISEK

Functional Interfaces

14 December 2021
10:03

As the name suggests, a functional interface is an interface that represents a function. Technically, an interface with just one abstract method is called a functional interface.

You can also use `@FunctionalInterface` to annotated a functional interface. In that case, the compiler will verify if the interface actually contains just one abstract method or not. It's like the @Override annotation, which prevents you from accidental errors.

Another useful thing to know is that If a method accepts a functional interface, then you can pass a lambda expression to it.

Some examples of the functional interface are **`Runnable`**, **`Callable`**, **`Comparator`**, and **`Comparable`** from old API and **`Supplier`**, **`Consumer`**, and **`Predicate`**, etc. from new function API.

Arrays

18 November 2021
19:56

Find Max value from array : {1,2,3,4,5,6,7,8,9};
Find @ values sum is 10 from above array.

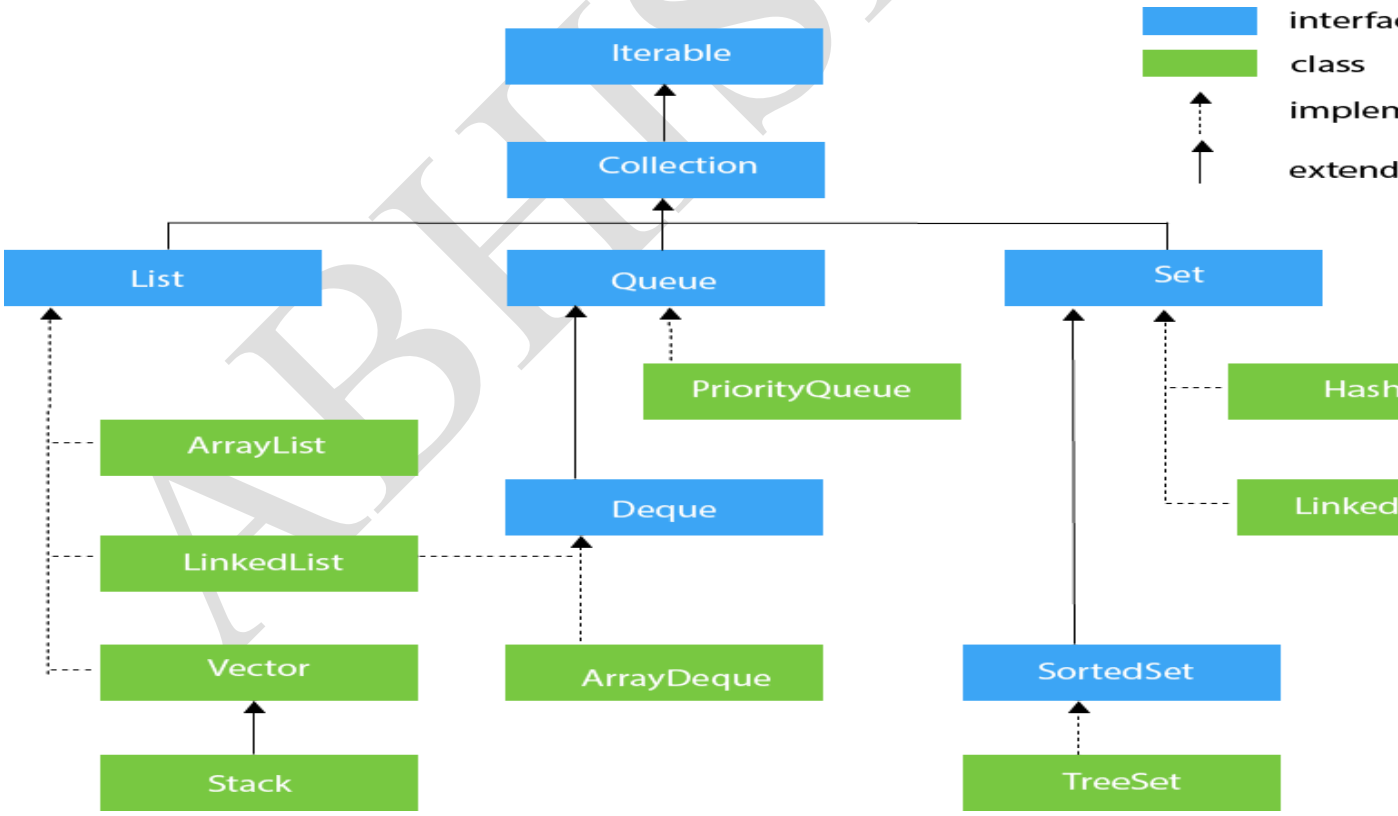
```
int temp;
int maxValue = 0 ;

for(int i=0;i++;i<numbers.length()){

    temp = numbers[i];
    if(temp > maxVlue){
          maxValue = temp;
    }
}
```

Collections

26 October 2021
12:59



Concurrent Modification Exception

Fialfast and FailSafe

ABHISEK

Add Employee Objects in Hashmap as a key

Comparator V/S Comparable

Comparable and Comparator both are interfaces and can be used to sort collection elements.
However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

```
public class Comparator {

	public static void main(String[] args) {

		ArrayList<Employee> employees = new ArrayList<>();
		employees.add(new Employee(111, "Adele", "Los Angeles"));
		employees.add(new Employee(141, "Aria", "Chicago"));
		employees.add(new Employee(121, "Ally", "Houston"));


		// Sorting Based On Employee ID
		System.out.println("Sorting Based On Employee ID");
		Collections.sort(employees, new EmpIdSort());
		employees.forEach(System.out::println);

		System.out.println("Sorting Based On Employee Name");
		Collections.sort(employees, new EmpNameSort());
		employees.forEach(System.out::println);
```

```
	}

}

class EmpIdSort implements Comparator<Employee> {

	@Override
	public int compare(Employee emp1, Employee emp2) {

		/*
		 * if (emp1.getEmp_id() == emp2.getEmp_id()) { return 0; } else if
		 * (emp1.getEmp_id() > emp2.getEmp_id()) { return 1; } return -1;
		 */

		return emp1.getEmp_id() - emp2.getEmp_id();

	}

}
```

What is Generics?

Time Complexity of Array list Operations, Linked List Operations , Hashmap Operations

Array Blocking Queue

Iterators

27 November 2021
07:34

Fail Fast and Fail Safe Iterator in Java

The Java Collection supports two types of iterators; Fail Fast and Fail Safe. These iterators are very useful in exception handling.

The Fail fast iterator aborts the operation as soon it exposes failures and stops the entire operation. Comparatively, Fail Safe iterator doesn't abort the operation in case of a failure. Instead, it tries to avoid failures as much as possible.

Concurrent Modification

The Concurrent modification in Java is to modify an object concurrently while another task is running over it. In simple terms, concurrent

ABHISEK

modification is the process of modifying objects while another thread is running over them. It will change the structure of the data collection, either by removing, adding, or updating the value of the elements in the collection.
Not all iterators support this behavior; implementation of some iterator may throw ConcurrentModificationException.

Fail Fast and Fail Safe Systems

The Fail Fast system is a system that shuts down immediately after an error is reported. All the operations will be aborted instantly in it.
The Fail Safe is a system that continues to operate even after an error or fail has occurred. These systems do not abort the operations instantly; instead, they will try to hide the errors and will try to avoid failures as much as possible.

Fail Fast:

The Fail Fast iterators immediately throw ConcurrentModificationException in case of structural modification of the collection. Structural modification means adding, removing, updating the value of an element in a data collection while another thread is iterating over that collection. Some examples of Fail Fast iterator are iterator on **ArrayList, HashMap collection classes.**

How it Works

The Fail Fast iterator uses an internal flag called **modCount** to know the status of the collection, whether the collection is structurally modified or not. The modCount flag is updated each time a collection is modified; it checks the next value; if it finds, then the modCount will be modified after this iterator has been created. It will throw ConcurrentModificationException.
Consider the below example to understand the behaviour of the Fail Fast iterator:

```
1.  import java.util.HashMap;
2.     import java.util.Iterator;
3.     import java.util.Map;
4.  public class FailFastDemo {
5.      public static void main(String[] args)
6.      {
7.          Map<String, String> empName = new HashMap<String, Str
    ing>();
8.          empName.put("Sam Hanks", "New york");
9.          empName.put("Will Smith", "LA");
10.         empName.put("Scarlett", "Chicago");
11.         Iterator iterator = empName.keySet().iterator();
12.         while (iterator.hasNext()) {
13.            System.out.println(empName.get(iterator.next()));
14.            // adding an element to Map
15.            // exception will be thrown on next call
```

```
            // of next() method.
            empName.put("Istanbul", "Turkey");
        }
    }
}
```

Output:

```
LA
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1445)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1469)
    at FailFastDemo.main(FailFastDemo.java:14)
```

From the above output, we can notice the following points:

1. The Fail Fast iterator throws a ConcurrentModificationException if a collection is modified while iterating over it.
2. The Fail Fast iterator uses an original collection to traverse over the collection's elements.
3. They are memory savers, don't require extra memory.
4. The Fail Fast iterators returned by ArrayList, HashMap, Vector classes.

Fail Safe Iterator

The Fail Safe iterators are just opposite to Fail Fast iterators; unlike them, A fail-safe iterator does not throw any exceptions unless it can handle if the collection is modified during the iteration process. This can be done because they operate on the copy of the collection object instead of the original object. The structural changes performed on the original collection ignored by them and affect the copied collection, not the original collection. So, the original collection will be kept structurally unchanged.

However, there is not any actual word written as Fail Safe in Java SE documentation; instead, the Fail Safe is termed as a non-Fail fast iterator.

FailSafeDemo1.java:

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;
public class FailSafeDemo1 {
    public static void main(String[] args)
    {
        // Initializing a ConcurrentHashMap
        ConcurrentHashMap<String, Integer> m
            = new ConcurrentHashMap<String, Integer>();
        m.put("ONE", 1); //Adding values
        m.put("SEVEN", 7);
        m.put("FIVE", 5);
        m.put("EIGHT", 8);
```

```
        // Getting an iterator using map
        Iterator it = m.keySet().iterator();
        while (it.hasNext()) {
            String key = (String)it.next();
            System.out.println(key + " : " + m.get(key));
            // This will reflect in iterator.
            // This means it has not created separate copy of objects
            m.put("NINE", 9);
        }
    }
}
```

Output:

```
EIGHT : 8
FIVE : 5
NINE : 9
ONE : 1
SEVEN : 7
```

From the above example, we can see we are iterating the collection while the other thread is performing. The iteration result is placed in the same collection, which means it is not creating any separate copy of the object and also does not throwing any ConcurrentModificationException.

From the above examples, we can notice the following points about the Fail Safe iterators:

1. We can perform the modification operations on a collection while iterating over it.
2. They will not throw ConcurrentModificationException during the iteration.
3. The Fail Safe iterators use a copy of the collection to traverse over the elements.
4. Unlike the Fail Fast, they require more memory as they cloned the collection.
5. The examples of Fail Safe iterators are ConcurrentHashMap, CopyOnWriteArrayList, etc.

Difference Between Fail Fast and Fail Safe Iterators

The Major difference between Fail Fast and Fail Safe iterator is that the Fail Safe does not throw any ConcurrentModificationException in modifying the object during the iteration process, contrary to fail fast, which throws an exception in such scenarios. This is because the Fail Safe iterator works on a cloned collection instead of the original collection.

There are several other comparisons between them on the basis of different parameters. Let's discuss them:

Base of Comparison	Fail Fast Iterator	Fail Safe Iterator
Exception	It throws a ConcurrentModificationException in modifying the object during the iteration process.	It does not throw Exception.
Clone Object	No clone object is created during the iteration process.	A copy or clone object is created during the iteration process.
Memory utilization	It requires low memory during the process.	It requires more memory during the process.
Modification	It does not allow modification during iteration.	It allows modification during the iteration process.
Performance	It is fast.	It is slightly slower than Fail Fast.
Examples	HashMap, ArrayList, Vector, HashSet, etc	CopyOnWriteArrayList, ConcurrentHashMap, etc.

ABH

List and Set

24 November 2021
08:02

ArrayList

The ArrayList class implements the List interface.
It uses a dynamic array to store the duplicate element of different data types.
The ArrayList class maintains the insertion order and is non-synchronized.
The elements stored in the ArrayList class can be randomly accessed.
Consider the following example.

Compare Based on Name and Quantity using Comparator

```
List<Laptop> laptopList = new ArrayList<>();
laptopList.add(new Laptop("HCL", 16, 800));
    laptopList.add(new Laptop("Apple", 8, 100));
    laptopList.add(new Laptop("Dell", 4, 600));
```

LinkedList

LinkedList implements the Collection interface.
It uses a doubly linked list internally to store the elements.
It can store the duplicate elements.
It maintains the insertion order and is not synchronized.
In LinkedList, the manipulation is fast because no shifting is required.

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.
Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

ALHISEK

Map

18 November 2021
19:59

Add Employee Objects in Hashmap as a key
Remove Duplicates from Hashmap values without using Hashset

ConcurrentHashMap	SynchronizedMap	HashTable
We will get thread safety without locking the total map object just with a bucket level lock.	We will get thread safety by locking the whole map object.	We will get thread safety b locking the whole map obj
At a time multiple threads are allowed to operate on map objects safely.	At a time only one thread is allowed to perform any operation on a map object.	At a time one thread is allo to operate on a map objec
Read operation can be performed without lock but write operation can be performed with bucket level lock.	Every read and write operations required total map object	Every read and write oper required total map object
While one thread iterating map objects the other thread is allowed to modify the map and won't get ConcurrentModificationException.	While one thread iterating map object the other threads are not allowed to modify the map otherwise we will get ConcurrentModificationExceptio n	While one thread iterating object the other threads a allowed to modify the map otherwise we will get ConcurrentModificationEx n
Iterator of ConcurrentHashMap is fail-safe and won't raise ConcurrentModificationException	Iterator of SynchronizedMap is fail-fast and it will raise ConcurrentModificationExceptio n	Iterator of HashTable is fa and it will raise ConcurrentModificationEx n
Null is not allowed for both keys and values.	Null is allowed for both keys and values	Null is not allowed for both and values.
Introduce in java 1.5version	Introduce in java 1.2 version	Introduce in java 1.0ve

Advantages of ConcurrentHashMap?

Spring Boot

16 November 2021
13:33

Why we are using @SpringBootApplication
What is @EnableAutoConfiguration

> Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need. Auto-configuration classes are usually applied based on your classpath and what beans you have defined. For example, if you have tomcat-embedded.jar on your classpath you are likely to want a TomcatServletWebServerFactory.

Diff B/W Configuration and ComponentScan
Why we should use SpringBoot

Difference between Controller and Rest Controller

Differnece Between PathParam and RequestParam :

Framework	Path segment	http query parameter
Jersey (JAX-RS)	@PathParam	@QueryParam
Spring RESTFul	**@PathVariable**	**@RequestParam**

@Bean and @Component. What is the different? Which one should use be used?

The most benefit of using a framework like Spring is auto-configuration. By only introducing the library or the module to the classpath, Spring could automatically scan it and create the necessary object and @Autowrired it.

@Component is an annotation that annotates a class. It tells Spring to use this class to create a bean if there is somewhere else depend on this class. The creation of the class is totally controlled by Spring. And only one bean created per class.

```
@Component
class UserService {
   public void updateUser(User user) {
   …
   }
}

@Controller
class UserController {
```

ABHISEK

```
    private final UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }
}
```

In this example, Spring will recognize the appearance of the UserSerivce class because it was marked with @Component annotation. So when it tries to initiate the UserController object, it knows that it needs a UserService to put to UserController constructor. And the creation of the UserService will be controlled totally by Spring. It is a pretty declarative way to define dependency.

With @Component our life is so good. So why do we even need something like @Bean annotation? When should we use it? First, @Bean is an annotation that used for annotating the function (not a class) that will return an object of a class that will be registered as a bean object by Spring.

You could use it in case you are using a third-party library when you don't have access to the source code of the library. So you can't just put @Component annotation of the class that you to create a bean.
You can combine with the @Configuration annotated the class that contains the method return the beans or otherwise Spring won't catch and register it as a bean for your dependency resolution.
Let's consider this example.

```
class UserService {
    private final PasswordEncoder encoder;

    @Autowired
    public UserService(PasswordEncoder encoder) {

          this.encoder = encoder;
    }

    public String createUser(UserCreateReq req) {
        UserEntity user = UserEntity.builder().setPassword(this.encoder.encode(req.getPassword()))
                        …
                        .build();
        repo.save(user);
    }
}

@Configuration
class PasswordEncoderConfiguration {
    @Bean
    public PasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

ABHISEK

In this example, I couldn't change the code base of the bcrypt library so to create the encoder object. I have to use a @Configuration class to create the bean that I need in a method inside that @Configuration class. In this way, we could create as many as you want bean objects. And you have to explicitly configure those bean by yourself when the conflict happens. One way to resolve that conflict is by using the @Qualifier annotation in the constructor of the dependent class.

When for 2 different methods making as sēāāēḥūr̥r̥ṁame endpoint.

```
@GetMapping(path = "/{id}")
public UserDetails getUserDetails() {}

@GetMapping(path = "/{userId}")
public String getUser(@PathVariable String userId) {}
```

ERROR:
java.lang.IllegalStateException: Ambiguous handler methods mapped for '/users/1': {public java.lang.String com.kidskart.app.controller.UsersController.getUser(java.lang.String), public com.kidskart.app.pojo.UserDetails com.kidskart.app.controller.UsersController.getUserDetails()}

Xml converter not found in class path: produces = {MediaType.APPLICATION_XML_VALUE}

2021-03-27 12:08:10.948 WARN 20828 --- [nio-8080-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.http.converter.HttpMessageNotWritableException: No converter for [class com.kidskart.app.pojo.UserDetails] with preset Content-Type 'null']

Solution: Add dependency in pom file : jakson

ABHISEK

Core Module

14 July 2022
15:23

Types of autowiring ?

Life cycle of Bean:

Types of bean injections
Scope of bean:

Bean scopes:

https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html

When you create a bean definition what you are actually creating is a *recipe* for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the *scope* of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to *choose* the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware ApplicationContext).

The scopes supported out of the box are listed below:

Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

Let's create a *Person* entity to exemplify the concept of scopes:

```
publicclassPerson{
    privateString name;
// standard constructor, getters and setters}
```

Afterwards, we define the bean with the *singleton* scope by using the *@Scope* annotation:

```
@Bean@Scope("singleton")publicPerson personSingleton(){
    returnnewPerson();
}
```

We can also use a constant instead of the *String* value in the following manner:

```
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
```

ABHISEK

Question:

Circular Dependencies

Simply put, circular dependencies occur when two or more classes depend on each other. Because of these dependencies, it's impossible to construct objects, and the execution can end up with runtime errors or infinite loops.

he use of field injection can result in circular dependencies going unnoticed:

```
@Componentpublicclass DependencyA{
@AutowiredprivateDependencyB dependencyB;
}
@Componentpublicclass DependencyB{
@AutowiredprivateDependencyA dependencyA;
}
```

Since the dependencies are injected when needed and not on the context load, Spring won't throw *BeanCurrentlyInCreationException*.

With constructor injection, it's possible to detect circular dependencies at compile time since they would create unresolvable errors.

Moreover, if we have circular dependencies in our code, it might be a sign something is wrong with our design. Therefore, we should consider redesigning our application if possible.

However, since Spring Boot 2.6. version circular dependencies are no longer allowed by default.

Properties

26 March 2022
06:38

Placeholders in Properties

The values in `application.properties` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties).

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

Using YAML Instead of Properties

YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The `SpringApplication` class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.

> Note: If you use “Starters”, SnakeYAML is automatically provided by `spring-boot-starter`.

ABHISEK

Stateless Vs Stateful

17 March 2022
07:57

Stateless Protocol:
Stateless Protocols are the type of network protocols in which Client send request to the server and server response back according to current state. It does not require the server to retain session information or a status about each communicating partner for multiple request.

2. Stateful Protocol:
In Stateful Protocol If client send a request to the server then it expects some kind of response, if it does not get any response then it resend the request. FTP (File Transfer Protocol), Telnet are the example of **Stateful Protocol**.
Silent features of Stateful Protocol:

Stateless Protocol	Stateful Protocol
Stateless Protocol does not require the server to retain the server information or session details.	Stateful Protocol require server to save the status and session information.
In Stateless Protocol, there is no tight dependency between server and client.	In Stateful protocol, there is tight dependency between server and client

The Stateless protocol design simplify the server design.	The Stateful protocol design makes the design of server very complex and heavy.
Stateless Protocols works better at the time of crash because there is no state that must be restored, a failed server can simply restart after a crash.	Stateful Protocol does not work better at the time of crash because stateful server have to keep the information of the status and session details of the internal states.
Stateless Protocols handle the transaction very fastly.	Stateful Protocols handle the transaction very slowly.
Stateless Protocols are easy to implement in Internet.	Stateful protocols are logically heavy to implement in Internet.

Stand Alone Spring Boot Application

17 March 2022
07:52

External Server Configuration

06 March 2022
20:51

Deploy and Run Spring Boot WAR with External Tomcat

There are a few steps that we must do for deploying and running the Spring Boot Application WAR with external Tomcat.

Step 1: Set War Packaging in pom.xml File

The very first step that we have to do is setting up the packaging of the artifact to WAR so that it creates a WAR file for us instead of the jar.

```
<packaging>war</packaging>
```

Step 2: Add and Set Tomcat Dependency to Provided

The second important step that is needed is that we have to add Tomcat server dependency and set its scope to provided.

set the tomcat server dependency to provide:

```
1
    <dependency>
2
      <groupid>org.springframework.boot</groupid>
```

ABHISEK

```
        <artifactid>spring-boot-starter-tomcat</artifactid>
        <scope>provided</scope>
      </dependency>
```

if some of your spring boot libraries contain it by default, exclude it:

```
<dependency>
<groupid>org.springframework.boot</groupid>
<artifactid>spring-boot-starter-web</artifactid>
      <exclusions>
        <exclusion>
          <groupid>org.springframework.boot</groupid>
          <artifactid>spring-boot-starter-tomcat</artifactid>
        </exclusion>
      </exclusions>
</dependency>
```

Step 3: Extend Spring Boot Application with SpringBootServletInitializer

Below is the way that you have to use it for extending your Spring Boot Application. Basically, you need to extend your Application with *SpringBootServletInitializer* like below mention code snippet.

```
@SpringBootApplicationpublicclassYourSpringBootApplicationextendsSpringBootServletInitializ
er{/**
   * The entry point of application.
   *
   * @param args the input arguments

*/publicstaticvoidmain(String[]args){SpringApplication.run(YourSpringBootApplication.class,
args);}@OverrideprotectedSpringApplicationBuilderconfigure(SpringApplicationBuilderbuilder)
{returnbuilder.sources(YourSpringBootApplication.class);}}
```

After doing all these steps you will be able to run your Spring Boot Application WAR on an external tomcat.

Why It is necessary to extend the Spring Boot Application with SpringBootServletInitializer for running it on an external Tomcat?

Basic information about this is that if you want to deploy a *Servlet-based* web application like Spring then actually you need to provide the traditional XML file that is web.xml file.

And if you go through the official docs of Spring then we can do the same things with the WebApplicationInitializer interface.

ABHISEK

Spring Boot simply suggests using Java Configuration over XML configuration. In very simple terms we can say that it uses Java Configuration instead of XML. It has *SpringBootServletInitializer* class which eventually implements the *WebApplicationInitializer* interface and overrides its *onStartup* and configure things for us.

That's the reason for extending the application with *SpringBootServletInitializer*.

Security

04 January 2022
08:13

OAuth2 Login on Specific Endpoints

We may want to enable Spring Security OAuth2.0 Login only on a specific endpoints in our application. To do that, we can use *HttpSecurity* to configure the OAuth2 Login setting.

For example, using URI Path to configure OAuth2 Login on all endpoints except one.

```
http.authorizeRequests()
    .antMatchers("/students/{id}").permitAll()
    .anyRequest().authenticated()
    .and()
    .oauth2Login();
```

As can be seen above, any user can access "/students/{id}" endpoint without any authentication requirement. However, all other endpoints need authentication.

Similarly, we may want only a specific HTTP methods on specific endpoints to be authenticated. Like, we may want to allow free access to read only endpoints like GET students, but mandate authentication for any PUT or POST.

Example of enabling OAuth2 Login on specific HTTP Methods.

ABISEK

```
http.authorizeRequests()
    .antMatchers(HttpMethod.GET, "/students").permitAll()
    .anyRequest().authenticated()
    .and()
    .auth2Login();
```

With this only GET requests on students resource are allowed a free access. While other methods need authentication.

Autowiring

16 December 2021
10:43

https://stackoverflow.com/questions/56642356/when-to-use-qualifier-and-primary-in-spring

Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection. Autowiring can't be used to inject primitive and string values. It works with reference only.

Profiles

13 December 2021
18:28

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded:

```
@Configuration@Profile("production")publicclassProductionConfiguration
{
// ...}
```

In the normal Spring way, you can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the usual ways, for example you could include it in your `application.properties`:

```
spring.profiles.active=dev,hsqldb
```

ABISEK

or specify on the command line using the
switch `--spring.profiles.active=dev,hsqldb`.

Transactions Management

11 December 2021
12:34

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency.

The concept of transactions can be described with the following four key properties described as

ACID:

- **Atomicity** − A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** − This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** − There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
- **Durability** − Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html

https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth

The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

ABHISEK

- Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO).
- Support for declarative transaction management.
- Simpler API for programmatic transaction management than complex transaction APIs such as JTA.
- Excellent integration with Spring's data access abstractions.

Spring resolves the disadvantages of global and local transactions. It enables application developers to use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Most users prefer declarative transaction management, which is recommended in most cases.

The TransactionDefinition interface specifies:

Isolation:
The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

Propagation:
Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. Spring offers all of the transaction propagation options familiar from EJB CMT. To read about the semantics of transaction propagation in Spring, see Section 16.5.7, "Transaction propagation".

Timeout:
How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.

Read-only status:
A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

The **TransactionStatus** interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus extends SavepointManager {

    boolean isNewTransaction();

    boolean hasSavepoint();
```

ABHISEK

```
    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();

}
```

How to use Spring's **@Transactional** annotation (**Declarative Transaction Management**)
Now let's have a look at what modern Spring transaction management usually looks like:

```
public class UserService {

    @Transactional
    public Long registerUser(User user) {
      // execute some SQL that e.g.
       // inserts the user into the db and retrieves the autogenerated id
       // userDao.save(user);
       return id;
    }
}
```

How is this possible? There is no more XML configuration and there's also no other code needed. Instead, you now need to do two things:

Make sure that your Spring Configuration is annotated with the **@EnableTransactionManagement** annotation (**In Spring Boot this will be done automatically for you**).

Make sure you specify a transaction manager in your Spring Configuration (this you need to do anyway).

And then Spring is smart enough to transparently handle transactions for you: Any bean's public method you annotate with the **@Transactional** annotation, will execute inside a database transaction (note: there are some pitfalls).

So, to get the @Transactional annotation working, all you need to do is this:

```
@Configuration
@EnableTransactionManagement
public class MySpringConfig {

    @Bean
    public PlatformTransactionManager txManager() {
```

ARHISEK

```
        return yourTxManager; // more on that later
    }

}
```

Now, when I say Spring transparently handles transactions for you. What does that really mean?

Armed with the knowledge from the JDBC transaction example, the **@Transactional** UserService code above translates (simplified) directly to this:

```
public class UserService {

    public Long registerUser(User user) {
        Connection connection = dataSource.getConnection(); //
(1)
        try (connection) {
            connection.setAutoCommit(false); // (1)

            // execute some SQL that e.g.
            // inserts the user into the db and retrieves the
autogenerated id
            // userDao.save(user); <(2)

            connection.commit(); // (1)
        } catch (SQLException e) {
            connection.rollback(); // (1)
        }
    }
}
```

1. This is all just standard opening and closing of a JDBC connection. That's what Spring's transactional annotation does for you automatically, without you having to write it explicitly.
2. This is your own code, saving the user through a DAO or something similar.

This example might look a bit *magical*, but let's have a look at how Spring inserts this connection code for you.

CGlib & JDK Proxies - @Transactional under the covers

Spring cannot really rewrite your Java class, like I did above, to insert the connection code (unless you are using advanced techniques like bytecode weaving, but we are ignoring that for now).

ABHISEK

Your registerUser() method really just calls userDao.save(user), there's no way to change that on the fly.

But Spring has an advantage. At its core, it is an IoC container. It instantiates a UserService for you and makes sure to autowire that UserService into any other bean that needs a UserService.

Now whenever you are using @Transactional on a bean, Spring uses a tiny trick. It does not just instantiate a UserService, but also a transactional *proxy* of that UserService.

It does that through a method called *proxy-through-subclassing* with the help of the Cglib library. There are also other ways to construct proxies (like Dynamic JDK proxies), but let's leave it at that for the moment.

Let's see proxies in action in this picture:

As you can see from that diagram, the proxy has one job.

- Opening and closing database connections/transactions.
- And then delegating to the *real UserService*, the one you wrote.
- And other beans, like your UserRestController will never know that they are talking to a proxy, and not the *real* thing.

Quick Exam

Have a look at the following source code and tell me what type of UserService Spring automatically constructs, assuming it is marked with @Transactional or has a @Transactional method.

```
@Configuration
@EnableTransactionManagement
```

```
public static class MyAppConfig {

   @Bean
   public UserService userService() {  // (1)
      return new UserService();
   }
}
```

Correct. **Spring constructs a dynamic CGLib proxy of your UserService class here that can open and close database transactions for you. You or any other beans won't even notice that it is not your UserService, but a proxy wrapping your UserService.**

For what do you need a Transaction Manager (like PlatformTransactionManager)?

Now there's only one crucial piece of information missing, even though we have mentioned it a couple of times already.

Your UserService gets proxied on the fly, and the proxy manages transactions for you. But it is not the proxy itself handling all this transactional state (open, commit, close), the proxy delegates that work to a transaction manager.

Spring offers you a PlatformTransactionManager / TransactionManager interface, which, by default, comes with a couple of handy implementations. One of them is the datasource transaction manager.

It does exactly what you did so far to manage transactions, but first, let's look at the needed Spring configuration:

```
@Bean
public DataSource dataSource() {
   return new MysqlDataSource(); // (1)
}

@Bean
public PlatformTransactionManager txManager() {
   return new DataSourceTransactionManager(dataSource());
// (2)
}
```

APHSEK

You create a database-specific or connection-pool specific datasource here. MySQL is being used for this example.

Here, you create your transaction manager, which needs a data source to be able to manage transactions.

Simple as. All transaction managers then have methods like "doBegin" (for starting a transaction) or "doCommit", which look like this - taken straight from Spring's source code and simplified a bit:

```
public class DataSourceTransactionManager implements
PlatformTransactionManager {

   @Override
   protected void doBegin(Object transaction,
TransactionDefinition definition) {
      Connection newCon =
obtainDataSource().getConnection();
      // ...
      con.setAutoCommit(false);
      // yes, that's it!
   }

   @Override
   protected void doCommit(DefaultTransactionStatus status) {
      // ...
      Connection connection =
status.getTransaction().getConnectionHolder().getConnection(
);
      try {
         con.commit();
      } catch (SQLException ex) {
         throw new TransactionSystemException("Could not
commit JDBC transaction", ex);
      }
   }
}
```

ABHISEK

So, the datasource transaction manager uses exactly the same code that you saw in the JDBC section, when managing transactions.

With this in mind, let's extend our picture from above:

To sum things up:

1. If Spring detects the @Transactional annotation on a bean, it creates a dynamic proxy of that bean.
2. The proxy has access to a transaction manager and will ask it to open and close transactions / connections.
3. The transaction manager itself will simply do what you did in the plain Java section: Manage a good, old JDBC connection.

What is the difference between physical and logical transactions?

Imagine the following two transactional classes.

```
@Service
public class UserService {

  @Autowired
  private InvoiceService invoiceService;

  @Transactional
  public void invoice() {
    invoiceService.createPdf();
    // send invoice as email, etc.
  }
}

@Service
public class InvoiceService {

  @Transactional
  public void createPdf() {
    // ...
  }
```

ABHISEK

```
}
```

UserService has a transactional invoice() method. Which calls another transactional method, createPdf() on the InvoiceService.

Now in terms of database transactions, this should really just be one database transaction. (Remember: getConnection(). setAutocommit(false). commit().) Spring calls this physical transaction, even though this might sound a bit confusing at first.

From Spring's side however, there's two logical transactions happening: First in UserService, the other one in InvoiceService. Spring has to be smart enough to know that both @Transactional methods, should use the same underlying, physical database transaction.

How would things be different, with the following change to InvoiceService?

```
@Service
public class InvoiceService {

   @Transactional(propagation =
Propagation.REQUIRES_NEW)
   public void createPdf() {
     // ...
   }
}
```

Changing the propagation mode to requires_new is telling Spring that createPDF() needs to execute in its own transaction, independent of any other, already existing transaction. Thinking back to the plain Java section of this guide, did you see a way to "split" a transaction in half? Neither did I.

Which basically means your code will open two (physical) connections/transactions to the database. (Again: getConnection() x2. setAutocommit(false) x2. commit() x2) Spring now has to be smart enough that the two logical transactional pieces (invoice()/createPdf()) now also map to two different, physical database transactions.

So, to sum things up:

ABUSER

1. Physical Transactions: Are your actual JDBC transactions.
2. Logical Transactions: Are the (potentially nested) @Transactional-annotated (Spring) methods.

This leads us to covering propagation modes in more detail.

What are @Transactional Propagation Levels used for?

When looking at the Spring source code, you'll find a variety of propagation levels or modes that you can plug into the @Transactional method.

```
@Transactional(propagation = Propagation.REQUIRED)

// or

@Transactional(propagation =
Propagation.REQUIRES_NEW)
// etc
```

The full list:

1. **REQUIRED**
2. **SUPPORTS**
3. **MANDATORY**
4. **REQUIRES_NEW**
5. **NOT_SUPPORTED**
6. **NEVER**
7. **NESTED**

Exercise:

In the plain Java section, I showed you *everything* that JDBC can do when it comes to transactions. Take a minute to think about what every single Spring propagation mode at the end *REALLY* does to your datasource or rather, your JDBC connection.

ABHISEK

Then have a look at the following answers.

Answers:

- **Required (default)**: My method needs a transaction, either open one for me or use an existing one → *getConnection(). setAutocommit(false). commit()*.
- **Supports**: I don't really care if a transaction is open or not, i can work either way → nothing to do with JDBC
- **Mandatory**: I'm not going to open up a transaction myself, but I'm going to cry if no one else opened one up → nothing to do with JDBC
- **Require_new:** I want my completely own transaction → *getConnection(). setAutocommit(false). commit()*.
- **Not_Supported:** I really don't like transactions, I will even try and suspend a current, running transaction → nothing to do with JDBC
- **Never:** I'm going to cry if someone else started up a transaction → nothing to do with JDBC
- **Nested:** It sounds so complicated, but we are just talking savepoints! → *connection.setSavepoint()*

As you can see, most propagation modes really have nothing to do with the database or JDBC, but more with how you structure your program with Spring and how/when/where Spring expects transactions to be there.

Look at this example:

```
public class UserService {

    @Transactional(propagation =
Propagation.MANDATORY)
    public void myMethod() {
      // execute some sql
    }

}
```

In this case, Spring will expect a transaction to be open, whenever you call myMethod() of the UserService class. It does not open one itself, instead, if you call that method without a pre-existing transaction,

ABHISEK

Spring will throw an exception. Keep this in mind as additional points for "logical transaction handling".

What are @Transactional Isolation Levels used for?

This is almost a trick question at this point, but what happens when you configure the @Transactional annotation like so?

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
```

Yes, it does simply lead to this:

```
connection.setTransactionIsolation(Connection.TRANSACTION
_REPEATABLE_READ);
```

Database isolation levels are, however, a complex topic, and you should take some time to fully grasp them. A good start is the official Postgres Documentation and their section on isolation levels.

Also note, that when it comes to switching isolation levels during a transaction, you must make sure to consult with your JDBC driver/database to understand which scenarios are supported and which not.

The most common @Transactional pitfall

There is one pitfall that Spring beginners usually run into. Have a look at the following code:

```
@Service
public class UserService {

   @Transactional
   public void invoice() {
      createPdf();
      // send invoice as email, etc.
   }

   @Transactional(propagation =
Propagation.REQUIRES_NEW)
   public void createPdf() {
      // ...
```

ABHISEK

```
    }
}
```

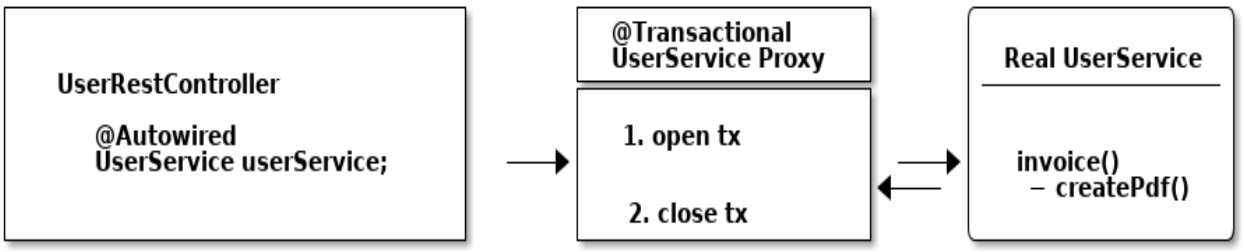
You have a UserService class with a transactional invoice method. Which calls createPDF(), which is also transactional.

How many physical transactions would you expect to be open, once someone calls invoice()?

Nope, the answer is not two, but one. Why?

Let's go back to the proxies' section of this guide. Spring creates that transactional UserService proxy for you, but once you are inside the UserService class and call other inner methods, there is no more proxy involved. This means, no new transaction for you.

Let's have a look at it with a picture:



There's some tricks (like self-injection), which you can use to get around this limitation. But the main takeaway is: always keep the proxy transaction boundaries in mind.

How to use @Transactional with Spring Boot or Spring MVC

So far, we have only talked about plain, core Spring. But what about Spring Boot? Or Spring Web MVC? Do they handle transactions any differently?
The short answer is: No.

With either frameworks (or rather: *all frameworks* in the Spring ecosystem), you will *always* use the @Transactional annotation, combined with a transaction manager and the @EnableTransactionManagement annotation. There is no other way.

The only difference with Spring Boot is, however, that it automatically sets the @EnableTransactionManagement annotation and creates a PlatformTransactionManager for you - with its JDBC auto-configurations. Learn more about auto-configurations here.

How Spring handles rollbacks (and default rollback policies)

The section on Spring rollbacks will be handled in the next revision of this guide.

How Spring and JPA / Hibernate Transaction Management works

The goal: Syncing Spring's @Transactional and Hibernate / JPA

At some point, you will want your Spring application to integrate with another database library, such as Hibernate (a popular JPA-implementation) or Jooq etc.

Let's take plain Hibernate as an example (note: it does not matter if you are using Hibernate directly,or Hibernate via JPA).

Rewriting the UserService from before to Hibernate would look like this:

```
public class UserService {

    @Autowired
    private SessionFactory sessionFactory; // (1)

    public void registerUser(User user) {

        Session session = sessionFactory.openSession(); // (2)
```

ABHISEK

```
            // lets open up a transaction. remember
        setAutocommit(false)!
            session.beginTransaction();

            // save == insert our objects
            session.save(user);

            // and commit it
            session.getTransaction().commit();

            // close the session == our jdbc connection
            session.close();
        }
    }
```

1. This is a plain, old Hibernate SessionFactory, the entry-point for all Hibernate queries.
2. Manually managing sessions (read: database connections) and transactions with Hibernate's API.

There is one huge problem with the above code, however:

- Hibernate would not know about Spring's @Transactional annotation.
- Spring's @Transactional would not know anything about Hibernate's transaction.

But we'd actually *love* for Spring and Hibernate to integrate seamlessly, meaning that they know about each others' transactions.

In plain code:

```
@Service
public class UserService {

    @Autowired
    private SessionFactory sessionFactory; // (1)

    @Transactional
    public void registerUser(User user) {
```

ABHISEK

```
            sessionFactory.getCurrentSession().save(user); //
(2)
        }

}
```

1. The same SessionFactory as before
2. But no more manual state management. Instead, getCurrentSession() and @Transactional are in sync.

How to get there?

Using the HibernateTransactionManager

There is a very simple fix for this integration problem:

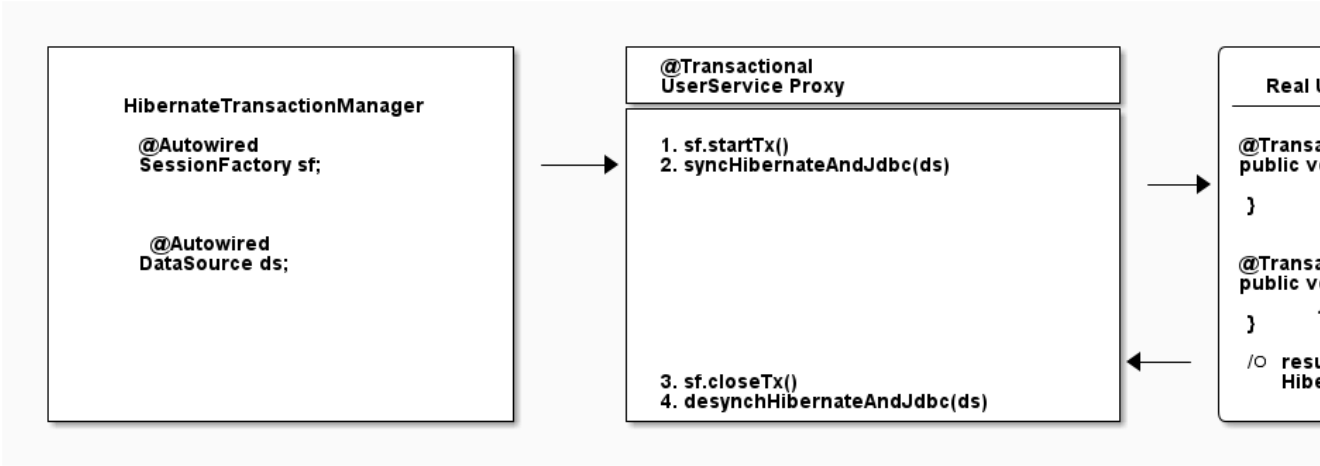
Instead of using a DataSourcePlatformTransactionManager in your Spring configuration, you will be using a HibernateTransactionManager (if using plain Hibernate) or JpaTransactionManager (if using Hibernate through JPA).

The specialized HibernateTransactionManager will make sure to:

1. Manage transactions through Hibernate, i.e. the SessionFactory.
2. Be smart enough to allow Spring to use that very same transaction in non-Hibernate, i.e. @Transactional Spring code.

As always, a picture might be simpler to understand (though note, the flow between the proxy and real service is only conceptually right and oversimplified).

ABHISEK



That is, in a nutshell, how you integrate Spring and Hibernate.

For other integrations or a more in-depth understanding, it helps to have a quick look at all possible PlatformTransactionManager implementations that Spring offers.

ABHISEK

Annotations

06 December 2021
10:41

https://javatechonline.com/spring-boot-annotations-with-examples/

This code uses Spring **@RestController** annotation, which marks the class as a controller where every method returns a domain object instead of a view.
It is shorthand for including both **@Controller** and **@ResponseBody.**

@SpringBootApplication is a convenience annotation that adds all of the following:

@Configuration: Tags the class as a source of bean definitions for the application context.

@EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
@ComponentScan: Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.

The main() method uses Spring Boot's SpringApplication.run() method to launch an application. Did you notice that there was not a single line of XML? There is no web.xml file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

ABHISEK

Hibernate

diff in hibernate session clear and close
what is hibernate session evict composite key candidate key
hibernate connection pooling

Hibernate mappings.

What is the easiest way to ignore a JPA field during persistence?

To ignore a field, annotate it with @Transient so it will not be mapped by hibernate.

but then jackson will not serialize the field when converting to JSON.

If you need mix JPA with JSON(omit by JPA but still include in Jackson) use @JsonInclude :

@JsonInclude()

```
@Transient
private String token;
```

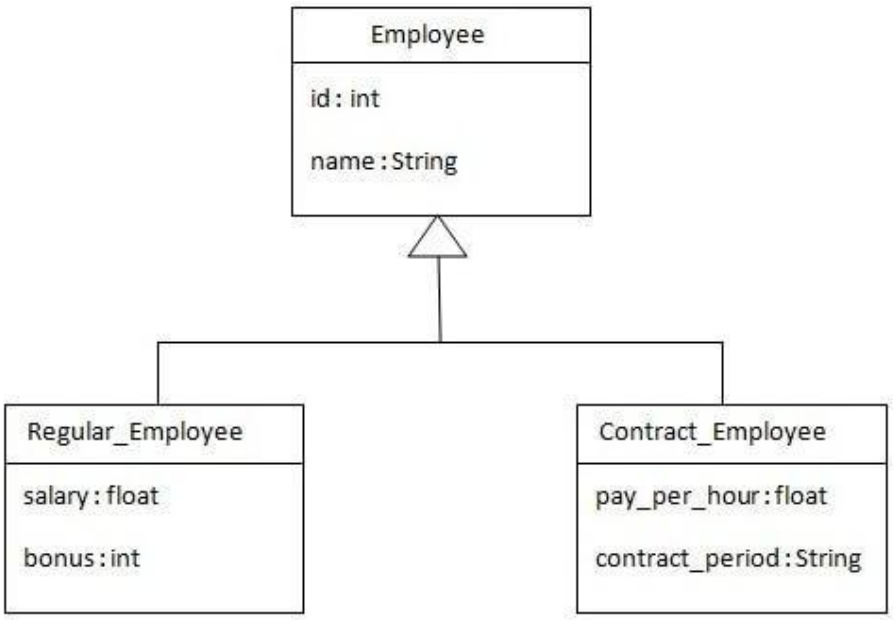
Inheritance Mapping

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy : **@Inheritance(strategy=InheritanceType.SINGLE_TABLE)**
2. Table Per Concrete class : **@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)**
3. Table Per Subclass : **@Inheritance(strategy=InheritanceType.JOINED)**

Table Per Hierarchy

You need to use **@Inheritance(strategy=InheritanceType.SINGLE_TABLE)**, @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.
In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.
Let's see the inheritance hierarchy:



The table structure for this hierarchy is as shown below:

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
TYPE	VARCHAR2(255)	No	-	-
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 7

```
@Entity
@Table(name = "employee101")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="employee")

public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

	@Column(name = "id")
	private int id;

	@Column(name = "name")
	private String name;

	//setters and getters
}
```

```
@Entity
@DiscriminatorValue("regularemployee")
public class Regular_Employee extends Employee{

	@Column(name="salary")
	private float salary;

	@Column(name="bonus")
	private int bonus;

	  //setters and getters
}
```

```
@Entity
@DiscriminatorValue("contractemployee")
public class Contract_Employee extends Employee{
```

ABHISEK

```
        @Column(name="pay_per_hour")
        private float pay_per_hour;

        @Column(name="contract_duration")
        private String contract_duration;

        //setters and getters
}


public class StoreTest {

   public static void main(String args[])
   {
      StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
      Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();

      SessionFactory factory=meta.getSessionFactoryBuilder().build();
      Session session=factory.openSession();

       Transaction t=session.beginTransaction();

         Employee e1=new Employee();
         e1.setName("Gaurav Chawla");

         Regular_Employee e2=new Regular_Employee();
         e2.setName("Vivek Kumar");
         e2.setSalary(50000);
         e2.setBonus(5);

         Contract_Employee e3=new Contract_Employee();
         e3.setName("Arjun Kumar");
         e3.setPay_per_hour(1000);
         e3.setContract_duration("15 hours");

         session.persist(e1);
         session.persist(e2);
         session.persist(e3);

         t.commit();
         session.close();
         System.out.println("success");
   }
}
```

Output:

ABHISEK

TYPE	ID	NAME	BONUS	SALARY	CONTRACT_DURATION	PAY_PER_HOUR
employee	1	Gaurav Chawla	-	-	-	-
regularemployee	2	Vivek Kumar	5	50000	-	-
contractemployee	3	Arjun Kumar	-	-	15 hours	1000

Table Per Concrete class

In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

Here, we need to use **@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)** annotation in the parent class and @AttributeOverrides annotation in the subclasses.

@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

@AttributeOverrides defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:

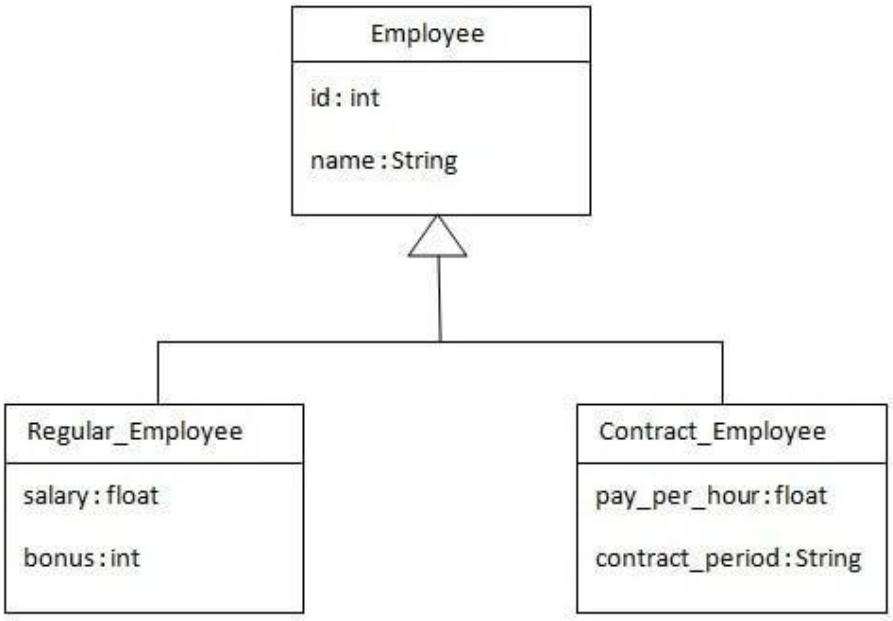


Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 4

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 4

ABHISEK

```
@Entity
@Table(name = "employee102")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

	@Column(name = "id")
	private int id;

	@Column(name = "name")
	private String name;

	//setters and getters
}

@Entity
@Table(name="regularemployee102")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="id")),
   @AttributeOverride(name="name", column=@Column(name="name"))
})
public class Regular_Employee extends Employee{

	@Column(name="salary")
	private float salary;
```

```
        @Column(name="bonus")
        private int bonus;

        //setters and getters
}


@Entity
@Table(name="contractemployee102")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="id")),
   @AttributeOverride(name="name", column=@Column(name="name"))
})
public class Contract_Employee extends Employee{

            @Column(name="pay_per_hour")
            private float pay_per_hour;

            @Column(name="contract_duration")
            private String contract_duration;

                //setters and getters

}


public class StoreData {

   public static void main(String[] args) {

      StandardServiceRegistry ssr=new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
      Metadata meta=new MetadataSources(ssr).getMetadataBuilder().build();

      SessionFactory factory=meta.getSessionFactoryBuilder().build();
      Session session=factory.openSession();

      Transaction t=session.beginTransaction();

      Employee e1=new Employee();
      e1.setName("Gaurav Chawla");

      Regular_Employee e2=new Regular_Employee();
      e2.setName("Vivek Kumar");
      e2.setSalary(50000);
      e2.setBonus(5);

      Contract_Employee e3=new Contract_Employee();
      e3.setName("Arjun Kumar");
      e3.setPay_per_hour(1000);
      e3.setContract_duration("15 hours");
```

ABHISEK

```
        session.persist(e1);
        session.persist(e2);
        session.persist(e3);

        t.commit();
        session.close();
        System.out.println("success");
    }
}
```

Table Per Subclass

As we have specified earlier, in case of table per subclass strategy, tables are created as per persistent classes but they are treated using primary and foreign key. So there will not be any duplicate column in the relation.

We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and @PrimaryKeyJoinColumn annotation in the subclasses.

Let's see the hierarchy of classes that we are going to map.

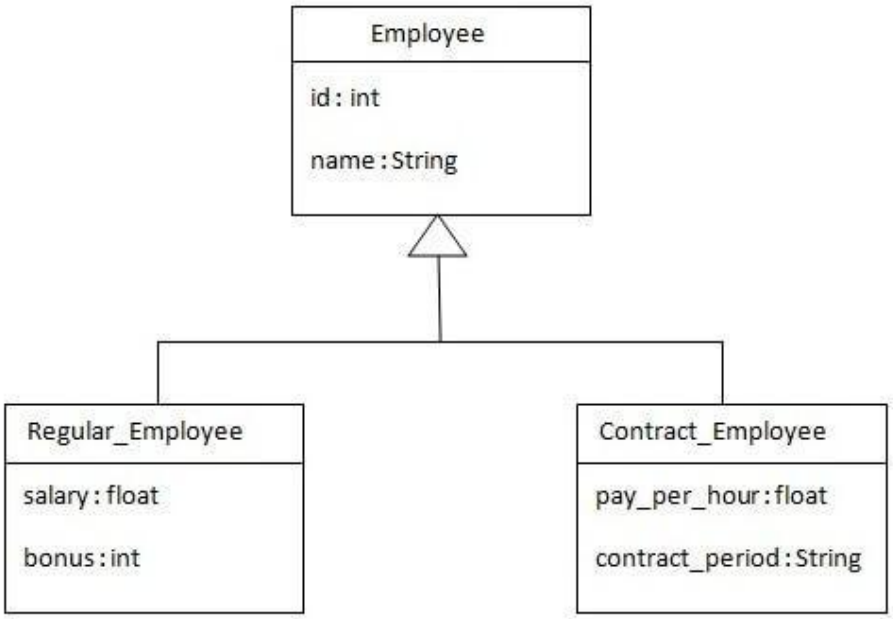


Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 3

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 3

```
@Entity
@Table(name = "employee103")
@Inheritance(strategy=InheritanceType.JOINED)

public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)

	@Column(name = "id")
	private int id;

	@Column(name = "name")
	private String name;

	//setters and getters
}

@Entity
@Table(name="regularemployee103")
@PrimaryKeyJoinColumn(name="ID")
public class Regular_Employee extends Employee{

	@Column(name="salary")
	private float salary;

	@Column(name="bonus")
	private int bonus;

	//setters and getters
}
```

ABHISEK

```
@Entity
@Table(name="contractemployee103")
@PrimaryKeyJoinColumn(name="ID")
public class Contract_Employee extends Employee{

        @Column(name="pay_per_hour")
        private float pay_per_hour;

        @Column(name="contract_duration")
        private String contract_duration;

        //setters and getters
}

public class StoreData {

   public static void main(String args[])
   {
      StandardServiceRegistry ssr = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
      Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();

      SessionFactory factory=meta.getSessionFactoryBuilder().build();
      Session session=factory.openSession();

       Transaction t=session.beginTransaction();

         Employee e1=new Employee();
         e1.setName("Gaurav Chawla");

         Regular_Employee e2=new Regular_Employee();
         e2.setName("Vivek Kumar");
         e2.setSalary(50000);
         e2.setBonus(5);

         Contract_Employee e3=new Contract_Employee();
         e3.setName("Arjun Kumar");
         e3.setPay_per_hour(1000);
         e3.setContract_duration("15 hours");

         session.persist(e1);
         session.persist(e2);
         session.persist(e3);

         t.commit();
         session.close();
         System.out.println("success");
   }
}
```

ABHISEK

Microservices

16 November 2021
13:35

Microservices:

Microservice architectures are the 'new normal'. Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code. Spring Boot's many purpose-built features make it easy to build and run your microservices in production at scale. And don't forget, no microservice architecture is complete without Spring Cloud – easing administration and boosting your fault-tolerance.

What are microservices?

Microservices are a modern approach to software whereby application code is delivered in small, manageable pieces, independent of others.

Why build microservices?

Their small scale and relative isolation can lead to many additional benefits, such as easier maintenance, improved productivity, greater fault tolerance, better business alignment, and more.

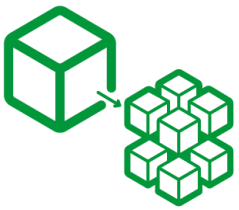
Microservices with Spring Boot:

With Spring Boot, your microservices can start small and iterate fast. That's why it has become the de facto standard for Java™ microservices. Quickstart your project with Spring Initializr and then package as a JAR. With Spring Boot's embedded server model, you're ready to go in minutes.

What are microservices?

In today's business environment, enterprises must respond to client needs and changing conditions more rapidly than ever. To keep up, software applications must be quick to deploy, easy to maintain, and always available. While traditional architecture can still handle a lot of this, there is a limit. At some point, a more dynamic, scalable approach to application development can become critical to the future of the business.

ARJISEK



One such approach is a microservice architecture. Microservices promise quicker and easier software changes compared to traditional monolithic architectures by modularizing complex applications. Developers then compose applications from the resulting interchangeable, upgradable, and scalable parts. In an ideal world, this modular architectural style accelerates business growth by enabling the agile deployment of innovative functionality. However, decomposing applications can also add complexity compared to a monolithic model. And that is only scratching the surface of the trade-offs.

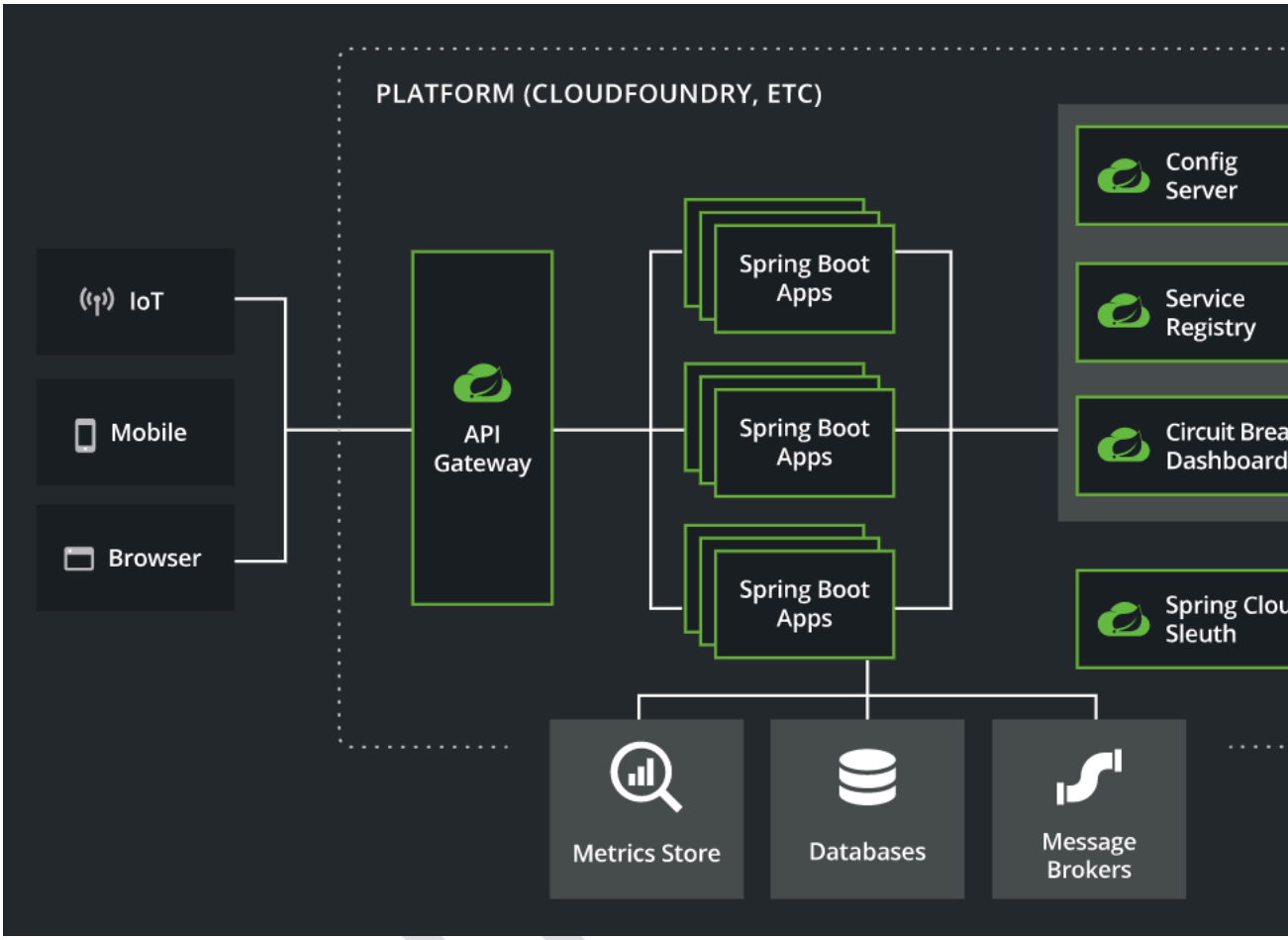
Microservices – or microservices architecture – are applications that are arranged or structured as a collection of loosely coupled services. In general, microservices have these characteristics:

Each microservice has its own data model and manages its own data.
Data moves between microservices using “dumb pipes” such as an event broker and/or a lightweight protocol like REST.
Small scope that encompasses a single piece of business functionality
Internal operations are a “black box”, accessible to external programs only via API

Microservice resilience with Spring Cloud

The distributed nature of microservices brings challenges. Spring helps you mitigate these. With several ready-to-run cloud patterns, Spring Cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring. It can even act as an API gateway.

ABHISEK



Advantages and Disadvantages of Micro Services?

The Advantages of Microservices

Microservices work well with agile development processes and satisfy the increasing need for a more fluid flow of information.

- **Microservices are independently deployable and allow for more team autonomy**
    - Each microservice can be deployed independently, as needed, enabling continuous improvement and faster app updates.
    - Specific microservices can be assigned to specific development teams, which allows them to focus solely on one service or feature. This means teams can work

autonomously without worrying what's going on with the rest of the app.

- **Microservices are independently scalable.**
    - As demand for an app increases, it's easier to scale using microservices. You can increase resources to the most needed microservices rather than scaling an entire app. This also means scaling is faster and often more cost-efficient as well.
- **Microservices reduce downtime through fault isolation.**
    - If a specific microservice fails, you can isolate that failure to that single service and prevent cascading failures that would cause the app to crash. This fault isolation means that your critical application can stay up and running even when one of its modules fails.
- **The smaller codebase enables teams to more easily understand the code, making it simpler to maintain.**
    - Microservice typically have small codebases, making them easier to maintain and deploy. It's also much easier to keep the code clean and for teams to be wholly responsible for specific services.

The Disadvantages of Microservices

Given the speed of business today and the rise of new technologies making microservices management even easier, the list of advantages is getting longer than the disadvantages — but there *are* still some disadvantages. While much of the development process is simplified with microservices, there are a few areas where microservices can actually cause *new* complexity.

- **Microservices create different types of complexity than monolithic applications for development teams.**

    - First, communication between services can be complex. An application can include dozens or even hundreds of different services, and they all need to communicate securely.
    - Second, debugging becomes more challenging with microservices. With an application consisting of multiple microservices and with each microservice having its own set of logs, tracing the source of the problem can be difficult.
    - And third, while unit testing may be easier with microservices, integration testing is not. The components are distributed, and developers can't test an entire system from their individual machines.

- **Interface control is even more critical.**
    - Each microservice has its own API, which apps rely on to be consistent. While you can easily make changes to a microservice without impacting the external systems interacting with it, if you change the API (the interface), any application using that microservice will be affected if the change is not backwards compatible.
    - A microservices architecture model results in a large number of APIs, all crucial to the operation of the enterprise — so interface control becomes mission-critical.
- **Up-front costs may be higher with microservices.**
    - For microservices architecture to work for your organization, you need sufficient hosting infrastructure with security and maintenance support, and you need skilled development teams who understand and manage all the services.
    - If you already have these things in place, the costs involved in moving to microservices may be lower — but most enterprises that are currently running

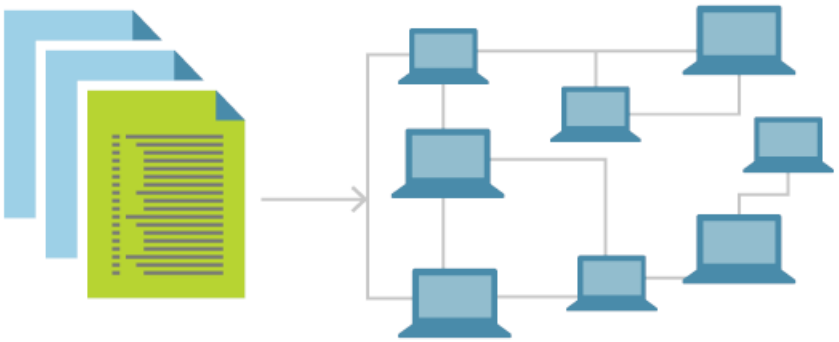
monolithic architecture will need to invest in the new infrastructure and developer resources in order to make the move.

What is API Gateway?

What is Service Discovery/Registry, use of it?

What is difference between monolithic and microservices architecture

What is Infrastructure as Code?



Infrastructure as Code (IaC) is the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model, using the same versioning as DevOps team uses for source code. Like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied. IaC is a key DevOps practice and is used in conjunction with continuous delivery.

Auto Scaling : https://dzone.com/articles/microservices-architecture-introduction-to-auto-sc

https://www.edureka.co/blog/interview-questions/microservices-interview-questions/

Q30. What is the use of PACT in Microservices architecture?

PACT is an open source tool to allow testing interactions between service providers and consumers in isolation against the contract made so that the reliability of Microservices integration increases.

Usage in Microservices:

- Used to implement Consumer Driven Contract in Microservices.
- Tests the consumer-driven contracts between consumer and provider of a Microservice.

What is a Consumer-Driven Contract (CDC)?

This is basically a pattern for developing Microservices so that they can be used by external systems. When we work on microservices, there is a particular provider who builds it and there are one or more consumers who use Microservice.
Generally, providers specify the interfaces in an XML document. But in Consumer Driven Contract, each consumer of service conveys the interface expected from the Provider.

Fallback Mechanism in Microservices.

Why service discovery in place of Resttemplate?

Restemplate is asynchronous or synchronous? synchronous

Fault Tolerance

Consider a scenario in which six microservices are communicating with each other. The **microservice-5** becomes down at some point, and all the other microservices are directly or indirectly depend on it, so all other services also go down.
The solution to this problem is to use a **fallback** in case of failure of a microservice. This aspect of a microservice is called **fault tolerance**.

Fault tolerance can be achieved with the help of a **circuit breaker**. It is a pattern that wraps requests to external services and detects when they fail. If a failure is detected, the circuit breaker opens. All the subsequent requests immediately return an error instead of making requests to the unhealthy service. It monitors and detects the service which is down and misbehaves with other services. It rejects calls until it becomes healthy again.

Feign Hystrix Fallbacks

Hystrix supports the notion of a fallback: a default code path that is executed when they circuit is open or there is an error. To enable fallbacks for a given `@FeignClient` set the `fallback` attribute to the class name that implements the fallback. You also need to declare your implementation as a Spring bean.

```
@FeignClient(name = "hello", fallback =
HystrixClientFallback.class)protectedinterfaceHystrixClient {
```

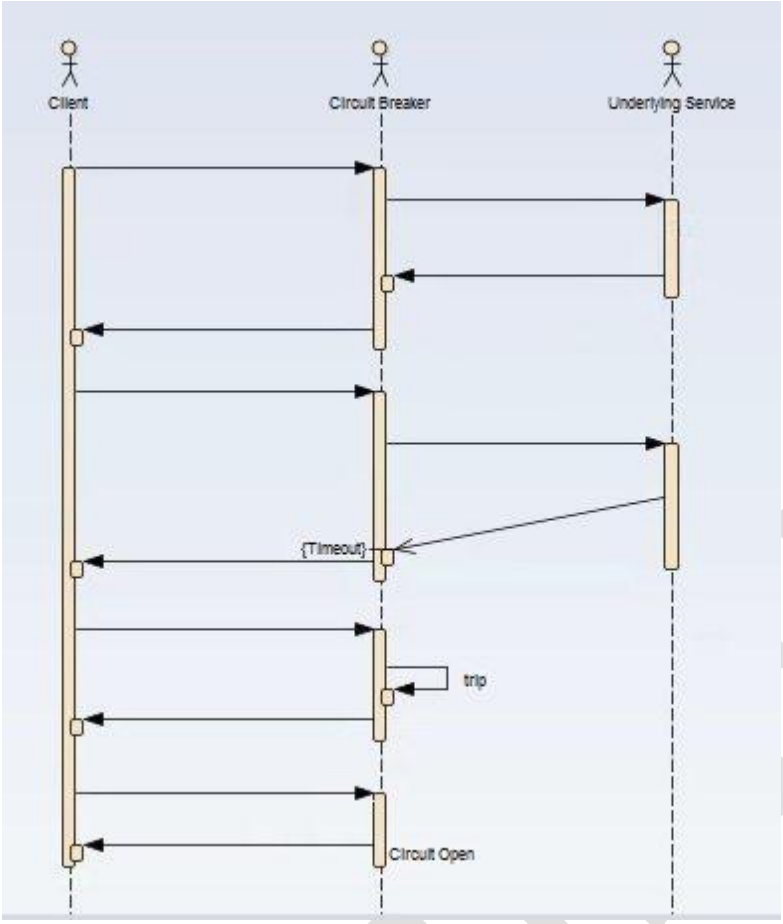
ARISEK

```
    @RequestMapping(method = RequestMethod.GET, value =
"/hello")Hello iFailSometimes();
}
staticclassHystrixClientFallback implementsHystrixClient {
    @OverridepublicHello iFailSometimes() {
        returnnewHello("fallback");
    }
}
```

Why is Circuit Breaker Pattern?

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.
Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.
So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back !! That's cool right?



- Hystrix – Enable Circuit Breaker

```
@SpringBootApplication
@EnableCircuitBreaker
public class SpringHystrixSchoolServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringHystrixSchoolServiceApplication.class, args);
    }
}


 @HystrixCommand(fallbackMethod = "callStudentServiceAndGetData_Fallback")
    public String callStudentServiceAndGetData(String schoolname) {
}
```

This time it will return the fall back method response. Here Hystrix comes into picture, it monitors Student service in frequent interval and as it is down, Hystrix component has opened the Circuit and fallback path enabled.

How do we implement asynchronous and synchronous communication between Micro services.

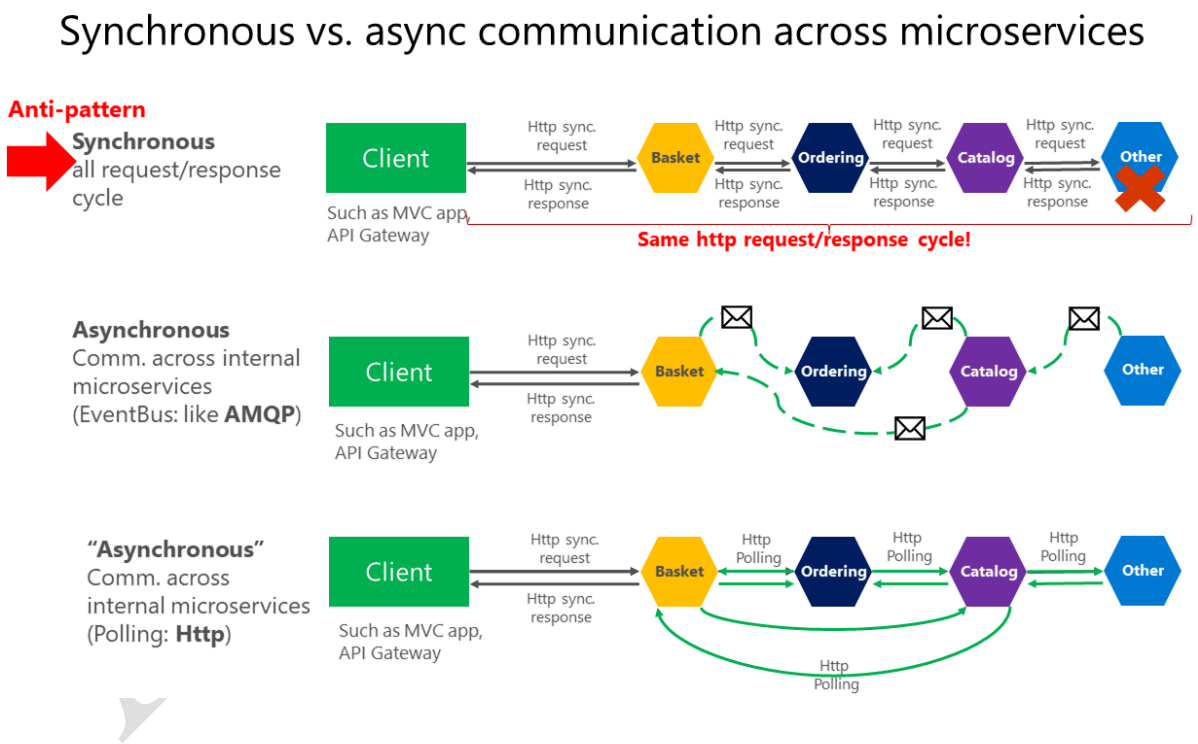
ABHISHEK

Communication types
Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.

The first axis defines if the protocol is synchronous or asynchronous:

Synchronous protocol. HTTP is a synchronous protocol. The client sends a request and waits for a response from the service. That's independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn't blocked, and the response will reach a callback eventually). The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.

Asynchronous protocol. Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code or message sender usually doesn't wait for a response. It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.



https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

Data Base Design in Microservices

Eureka Server

28 September 2022
15:28

Feign Client

03 December 2021
09:06

Declarative REST Client: Feign

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations.

```
@SpringBootApplication@EnableFeignClientspublicclassApplication {
publicstaticvoidmain(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

ABHISEK

```
}
```

StoreClient.java.

```
@FeignClient("stores")publicinterfaceStoreClient {
    @RequestMapping(method = RequestMethod.GET, value =
"/stores")List<Store> getStores();
@RequestMapping(method = RequestMethod.POST, value =
"/stores/{storeId}", consumes = "application/json")Store
update(@PathVariable("storeId")Long storeId, Store store);
}
```

API Gateway

17 March 2022
07:59

Design Patterns

16 November 2021
13:38

1. **What is Design Pattern ?**

   Design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

2. **Types Of Design Patterns?**

   Design patterns had originally been categorized into 3 sub-classifications based on what kind of problem they solve.

S.N.	Pattern	Description
1	**Creational Patterns**	These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

ABHISEK

2	**Structural Patterns**	These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioural Patterns**	These design patterns are specifically concerned with communication between objects.

- **Creational patterns**

Name	Description
Abstract factory	Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.
Factory method	Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

- **Structural patterns**

Name	Description
Adapter, Wrapper, or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
Extension object	Adding functionality to a hierarchy without changing the hierarchy.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Front controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.

Marker	Empty interface to associate metadata with a class.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.

- **Behavioural patterns**

Name	Description
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.
Specification	Recompilable business logic in a Boolean fashion.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on instances of a set of classes. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.

1. **Singleton Design Pattern with Example**

Singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system.

Implementations of the singleton pattern must:

Ensure that only one instance of the singleton class ever exists and Provide global access to that instance.
Typically, this is done by:

Declaring all constructors of the class to be private.
Providing a static method that returns a reference to the instance.
The instance is usually stored as a private static variable;
The instance is created when the variable is initialized, at some point before the static method is first called.

```
public final class Singleton {

  private static volatile Singleton instance = null;

  private Singleton() {}

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

How to prevent Singleton Pattern from Reflection, Serialization and Cloning?

https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/

ABHISEK

we will see that what are various concepts which can break singleton property of a class and how to avoid them. There are mainly 3 concepts which can break singleton property of a class. Let's discuss them one by one.

1. Reflection

Overcome reflection issue: To overcome issue raised by reflection, enums are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

2. Serialization

Overcome serialization issue:- To overcome this issue, we have to implement method readResolve() method.

3. Cloning

Overcome Cloning issue:- To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton. If you don't want to throw exception you can also return the same instance from clone method.

1. Reflection: Reflection can be caused to destroy singleton property of singleton class, as shown in following example

```
// Java code to explain effect of Reflection on Singleton property

import java.lang.reflect.Constructor;

// Singleton class
class Singleton
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }
}

public class GFG
{

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

ABHIEK

```
        Singleton instance1 = Singleton.instance;
        Singleton instance2 = null;
        try
        {
            Constructor[] constructors
= Singleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors)
            {
                // Below code will destroy the singleton pattern
                constructor.setAccessible(true);
                instance2 = (Singleton) constructor.newInstance();
                break;
            }
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

    System.out.println("instance1.hashCode():- "  +
instance1.hashCode());
    System.out.println("instance2.hashCode():- "  +
instance2.hashCode());
    }
}
```

```
Output:-
instance1.hashCode():- 366712642
instance2.hashCode():- 1829164700
```

After running this class, you will see that hashCodes are different that means, 2 objects of same class are created and singleton pattern has been destroyed.

Overcome reflection issue: To overcome issue raised by reflection, enums are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

```
//Java program for Enum type singleton
public enum Singleton
{
  INSTANCE;
}
```

As enums don't have any constructor so it is not possible for Reflection to utilize it. Enums have their by-default constructor, we can't invoke them by ourself. **JVM handles the creation and invocation of enum constructors internally.** As enums don't give their constructor definition

ARISEK

to the program, it is not possible for us to access them by Reflection also. Hence, reflection can't break singleton property in case of enums.

Serialization:- Serialization can also cause breakage of singleton property of singleton classes. Serialization is used to convert an object of byte stream and save in a file or send over a network. Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

```
// Java code to explain effect of Serialization on singleton classes
import java.io.*;

class Singleton implements Serializable
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }
}


public class GFG
{

    public static void main(String[] args)
    {
        try
        {
            Singleton instance1 = Singleton.instance;
            ObjectOutput out  = new ObjectOutputStream(new
FileOutputStream("file.text"));
            out.writeObject(instance1);
            out.close();

            // deserialize from file to object
            ObjectInput in = new ObjectInputStream(new
FileInputStream("file.text"));

            Singleton instance2 = (Singleton) in.readObject();
            in.close();

            System.out.println("instance1 hashCode:- " +
instance1.hashCode());
            System.out.println("instance2 hashCode:- " +
instance2.hashCode());
        }

        catch (Exception e)
```

ABHISEK

```
        {
            e.printStackTrace();
        }
    }
}
```

Output:-

```
instance1 hashCode:- 1550089733
instance2 hashCode:- 865113938
```

As you can see, hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

Overcome serialization issue:- To overcome this issue, we have to implement method readResolve() method.

```
// Java code to remove the effect of Serialization on singleton
classes
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Singleton implements Serializable
{
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }

    // implement readResolve method
    protected Object readResolve()
    {
        return instance;
    }
}

public class GFG
{

    public static void main(String[] args)
    {
        try
        {
            Singleton instance1 = Singleton.instance;
            ObjectOutput out = new ObjectOutputStream(new
FileOutputStream("file.text"));
```

ABHISEK

```
            out.writeObject(instance1);
            out.close();

            // deserialize from file to object
            ObjectInput in = new ObjectInputStream(new
FileInputStream("file.text"));
            Singleton instance2 = (Singleton) in.readObject();
            in.close();

            System.out.println("instance1 hashCode:- " +
instance1.hashCode());
            System.out.println("instance2 hashCode:- " +
instance2.hashCode());
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:-

```
instance1 hashCode:- 1550089733
instance2 hashCode:- 1550089733
```

Above both hashcodes are same hence no other instance is created.

Cloning: Cloning is a concept to create duplicate objects. Using clone we can create copy of object. Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

```
// Java code to explain cloning issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
```

ABUSEK

```
    // private constructor
  }
}

public class GFG
{
  public static void main(String[] args) throws
CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "   +
instance1.hashCode());
    System.out.println("instance2 hashCode:- "   +
instance2.hashCode());
  }
}
```

```
Output :-
instance1 hashCode:- 366712642
instance2 hashCode:- 1829164700
```

Two different hashCode means there are 2 different objects of singleton class.

Overcome Cloning issue:- To overcome this issue, override clone() method and throw an exception from clone method that is CloneNotSupportedException. Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

```
// Java code to explain overcome cloning issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
    // private constructor
  }
```

ABHISEK

```
  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    throw new CloneNotSupportedException();
  }
}

public class GFG
{
  public static void main(String[] args) throws
CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- "  +
instance1.hashCode());
    System.out.println("instance2 hashCode:- "  +
instance2.hashCode());
  }
}
```

```
Output:-
Exception in thread "main" java.lang.CloneNotSupportedException
    at GFG.Singleton.clone(GFG.java:29)
    at GFG.GFG.main(GFG.java:38)
```

Now we have stopped user to create clone of singleton class. If you dont want to throw exception you can also return the same instance from clone method.

```
// Java code to explain overcome cloning issue with singleton
class SuperClass implements Cloneable
{
  int i = 10;

  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }
}

// Singleton class
class Singleton extends SuperClass
{
  // public instance initialized when loading the class
  public static Singleton instance = new Singleton();

  private Singleton()
  {
    // private constructor
  }
```

ABHISEK

```
  @Override
  protected Object clone() throws CloneNotSupportedException
  {
    return instance;
  }
}

public class GFG
{
  public static void main(String[] args) throws
CloneNotSupportedException
  {
    Singleton instance1 = Singleton.instance;
    Singleton instance2 = (Singleton) instance1.clone();
    System.out.println("instance1 hashCode:- " +
instance1.hashCode());
    System.out.println("instance2 hashCode:- " +
instance2.hashCode());
  }
}
```

```
Output:-
instance1 hashCode:- 366712642
instance2 hashCode:- 366712642
```

Now, as hashcode of both the instances is same that means they represent a single instance.

BHISEK

1. **Builder Design pattern with Example**

- The builder pattern is a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

- The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object it is to create.

- The Builder design pattern is designed to provide a flexible solution to various object creation problems in object-oriented programming.

- The Builder design pattern provides a way to separate the construction of a complex object from its representation.
- The Builder pattern constructs a complex object by using simple objects and step by step approach.
- The pattern provides one of the best ways to create a complex object.
- This pattern is useful to build different immutable objects using same object building process.

Factory Method Pattern

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

Usage of Factory Design Pattern

- When a class doesn't know what sub-classes will be required to create
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.

Calculate Electricity Bill : A Real World Example of Factory Method

Step 1: Create a Plan abstract class.

```
import java.io.*;

abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}
```

Step 2: Create the concrete classes that extends Plan abstract class.

```
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
           rate=3.50;
```

ABHISEK

```
    }
  }

class  CommercialPlan extends Plan{
  //@override
   public void getRate(){
      rate=7.50;
  }
 }

class  InstitutionalPlan extends Plan{
  //@override
   public void getRate(){
      rate=5.50;
  }
 }
```

Step 3: Create a GetPlanFactory to generate object of concrete classes based on given information..

```
class GetPlanFactory{

  //use getPlan method to get object of type Plan
     public Plan getPlan(String planType){
         if(planType == null){
          return null;
         }
        if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
             return new DomesticPlan();
           }
         else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
            return new CommercialPlan();
         }
        else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
        }
     return null;
  }
}
```

Step 4: Generate Bill by using the GetPlanFactory to get the object of concrete classes by passing an information such as type of plan DOMESTICPLAN or COMMERCIALPLAN or INSTITUTIONALPLAN.

```
import java.io.*;
class GenerateBill{

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

ABHISEK

```
    GetPlanFactory planFactory = new GetPlanFactory();


    Plan p = planFactory.getPlan("INSTITUTIONALPLAN");
    //call getRate() method and calculateBill()method of DomesticPaln.

     System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
        p.getRate();
        p.calculateBill(units);
         }
}
```

Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

Advantage of Abstract Factory Pattern

- Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
- It eases the exchanging of object families.
- It promotes consistency among objects.

Usage of Abstract Factory Pattern

- When the system needs to be independent of how its object are created, composed, and represented.
- When the family of related objects has to be used together, then this constraint needs to be enforced.
- When you want to provide a library of objects that does not show implementations and only reveals interfaces.
- When the system needs to be configured with one of a multiple family of objects.

Example of Abstract Factory Pattern

Here, we are calculating the loan payment for different banks like HDFC, ICICI, SBI etc.

Step 1: Create a Bank interface

```
import java.io.*;
interface Bank{
      String getBankName();
```

```
}
```

Step 2: Create concrete classes that implement the Bank interface.

```
class HDFC implements Bank{
        private final String BNAME;
        public HDFC(){
              BNAME="HDFC BANK";
       }
       public String getBankName() {
               return BNAME;
       }
}

class ICICI implements Bank{
      private final String BNAME;
      ICICI(){
              BNAME="ICICI BANK";
       }
       public String getBankName() {
               return BNAME;
      }
}

class SBI implements Bank{
     private final String BNAME;
     public SBI(){
              BNAME="SBI BANK";
       }
      public String getBankName(){
               return BNAME;
      }
}
```

Step 3: Create the Loan abstract class.

```
abstract class Loan{

  protected double rate;
  abstract void getInterestRate(double rate);
  public void calculateLoanPayment(double loanamount, int years)
  {
      double EMI;
       int n;

       n=years*12;
       rate=rate/1200;
```

ABHISEK

```
        EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount
;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount
+" you have borrowed");
 }
}
```

Step 4: Create concrete classes that extend the Loan abstract class..

```
class HomeLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
   }
}

class BussinessLoan extends Loan{
   public void getInterestRate(double r){
        rate=r;
    }

}

class EducationLoan extends Loan{
    public void getInterestRate(double r){
      rate=r;
 }
}
```

Step 5: Create an abstract class (i.e AbstractFactory) to get the factories for Bank and Loan Objects.

```
abstract class AbstractFactory{

  public abstract Bank getBank(String bank);
  public abstract Loan getLoan(String loan);

}
```

Step 6: Create the factory classes that inherit AbstractFactory class to generate the object of concrete class based on given information.

```
class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
     if(bank == null){
       return null;
```

ABHISEK

```
    }
    if(bank.equalsIgnoreCase("HDFC")){
      return new HDFC();
    } else if(bank.equalsIgnoreCase("ICICI")){
      return new ICICI();
    } else if(bank.equalsIgnoreCase("SBI")){
      return new SBI();
    }
    return null;
  }
  public Loan getLoan(String loan) {
    return null;
  }
}

class LoanFactory extends AbstractFactory{
        public Bank getBank(String bank){
            return null;
        }

    public Loan getLoan(String loan){
    if(loan == null){
      return null;
    }
    if(loan.equalsIgnoreCase("Home")){
      return new HomeLoan();
    } else if(loan.equalsIgnoreCase("Business")){
      return new BussinessLoan();
    } else if(loan.equalsIgnoreCase("Education")){
      return new EducationLoan();
    }
    return null;
  }

}
```

Step 7: Create a FactoryCreator class to get the factories by passing an information such as Bank or Loan.

```
class FactoryCreator {

    public static AbstractFactory getFactory(String choice){

    if(choice.equalsIgnoreCase("Bank")){
      return new BankFactory();
    } else if(choice.equalsIgnoreCase("Loan")){
      return new LoanFactory();
    }
```

ABHISEK

```
    return null;
  }
}
```

Step 8: Use the FactoryCreator to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

```
import java.io.*;

class AbstractFactoryPatternExample {

    public static void main(String args[])throws IOException {

    BufferedReader br=new BufferedReader(new InputStreamReader(System.in
));

        System.out.print("Enter the name of Bank from where you want to take
 loan amount: ");
        String bankName="HDFC"  ;
       String loanName="Home";

      AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
      Bank b=bankFactory.getBank(bankName);

      System.out.print("\n");
      System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

      double rate=Double.parseDouble(br.readLine());
      System.out.print("\n");
      System.out.print("Enter the loan amount you want to take: ");

      double loanAmount=Double.parseDouble(br.readLine());
      System.out.print("\n");
      System.out.print("Enter the number of years to pay your entire loan amoun
      t: ");
      int years=Integer.parseInt(br.readLine());

      System.out.print("\n");
      System.out.println("you are taking the loan from "+ b.getBankName());

      AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");

        Loan l= loanFactory.getLoan(loanName);
        l.getInterestRate(rate);
        l.calculateLoanPayment(loanAmount,years);
```

ABHISEK

```
  }
}
```

Proxy Design pattern :

Proxy means 'in place of', representing' or 'in place of' or 'on behalf of' are literal meanings of proxy and that directly explains Proxy Design Pattern.
Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.

A real world example can be a cheque or credit card is a proxy for what is in our bank account.
It can be used in place of cash, and provides a means of accessing that cash when required.
And that's exactly what the Proxy pattern does – "Controls and manage access to the object they are protecting".

```
//Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

interface Internet {
	public void connectTo(String serverhost) throws Exception;
}

class RealInternet implements Internet {
	@Override
	public void connectTo(String serverhost) {
		System.out.println("Connecting to " + serverhost);
	}
}

class ProxyInternet implements Internet {
	private Internet internet = new RealInternet();
	private static List<String> bannedSites;

	static {
		bannedSites = new ArrayList<String>();
		bannedSites.add("abc.com");
		bannedSites.add("def.com");
		bannedSites.add("ijk.com");
		bannedSites.add("lnm.com");
	}

	@Override
	public void connectTo(String serverhost) throws Exception {
		if (bannedSites.contains(serverhost.toLowerCase())) {
			throw new Exception("Access Denied");
		}
```

ABHISEK

```
            internet.connectTo(serverhost);
        }
}

public class ProxyPatternDemo {

        public static void main(String[] args) {
            Internet internet = new ProxyInternet();
            try {
                internet.connectTo("geeksforgeeks.org");
                internet.connectTo("abc.com");
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

}
```

SAGA Design Pattern

ABHISEK

SOLID Principles

23 November 2021
12:24

SOLID principles of object-oriented design.

1. Single Responsibility Principle (SRP)	**Every Java class must perform a single functionality**
1. Open-Closed Principle (OCP)	**The module should be open for extension but closed for modification.**
1. Liskov Substitution Principle (LSP)	**Derived classes must be completely substitutable for their base classes**.
1. Interface Segregation Principle (ISP)	**Larger interfaces split into smaller ones.**
1. Dependency Inversion Principle (DIP)	**High-level modules should not depend on low-level modules. Both should depend upon abstractions.**

ASHISEK

Immutable Classes

25 November 2021
19:12

Immutable class in java 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. Prior to going ahead do go through characteristics of immutability in order to have a good understanding while implementing the same. Following are the requirements:

How do we modify data of immutable class?

- The class must be declared as final so that child classes can't be created.
- Data members in the class must be declared 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 an object reference.
- Deep Copy of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)

```
// Java Program to Create An Immutable Class

// Class  An immutable class
final class Student {

    // Member attributes of final class
    private final String name;
    private final int regNo;
    private final Map<String, String> metadata;

    // Constructor of immutable class Parameterized constructor
    public Student(String name, int regNo, Map<String, String> metadata)
    {
```

ABHISEK

```
            // This keyword refers to current instance itself
            this.name = name;
            this.regNo = regNo;

            // Creating Map object with reference to HashMap Declaring object of string type
            Map<String, String> tempMap = new HashMap<>();

            // Iterating using for-each loop
            for (Map.Entry<String, String> entry :
                    metadata.entrySet()) {
                    tempMap.put(entry.getKey(), entry.getValue());
            }

            this.metadata = tempMap;
        }

        // Method 1
        public String getName() { return name; }

        // Method 2
        public int getRegNo() { return regNo; }

        // Note that there should not be any setters
        // Method 3 User -defined type To get meta data
        public Map<String, String> getMetadata()
        {

            // Creating Map with HashMap reference
            Map<String, String> tempMap = new HashMap<>();

            for (Map.Entry<String, String> entry :
                    this.metadata.entrySet()) {
                    tempMap.put(entry.getKey(), entry.getValue());
            }
            return tempMap;
        }
}

// Class 2
// Main class
class GFG {

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

            // Creating Map object with reference to HashMap
            Map<String, String> map = new HashMap<>();

            // Adding elements to Map object
```

ABHISEK

```
            // using put() method
            map.put("1", "first");
            map.put("2", "second");

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

            // Calling the above methods 1,2,3 of class1 inside main() method in class2 and
            executing the print statement over them
            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");
            // Remains unchanged due to deep copy in constructor
            System.out.println(s.getMetadata());
            s.getMetadata().put("4", "fourth");
            // Remains unchanged due to deep copy in getter
            System.out.println(s.getMetadata());
        }
}
```

Employee contains Address :

Employee is immutable but Address is not immutable ? How do we achieve immutability?

ABHISEK

In this case, #2 probably means you can't return a reference to Address like you have with `getAddress()`. **And** you have to make a defensive copy in the constructor. I.e., make a copy of any mutable parameter, and store the copy in Employee. If you can't make a defensive copy, there's really no way to make Employee immutable.

```
public final class Employee{
    private final int id;
    private final Address address;
    public Employee(int id, Address address)
    {
        this.id = id;
        this.address=new Address();  // defensive copy
        this.address.setStreet( address.getStreet() );
    }
    public int getId(){
        return id;
    }
    public Address getAddress() {
        Address nuAdd = new Address(); // must copy here too
        nuAdd.setStreet( address.getStreet() );
        return nuAdd;
}
```

Implementing `clone()` or something similar (a copy ctor) would make creating defensive objects easier for complicated classes. However, the best recommendation I think would be to make `Address` immutable. Once you do that you can freely pass around its reference without any thread-safety issues.

In this example, notice I do *NOT* have to copy the value of `street`. `Street` is a String, and strings are immutable. If `street` consisted of mutable fields (integer street number for example) then I *would* have to make a copy of `street` also, and so on ad infinitum. This is why immutable objects are so valuable, they break the "infinite copy" chain.

Immutable with Date
Immutable with Mutable

Database

07 December 2021

08:18

1. **2nd Highest Salary**

   Select salary from (Select rownum row_num, salary (Select Salary from employees order by salary desc)) where row_num=2;

2. **Salary between Range :** Display the name, age and address of customers whose salary is between 9000 and 10000 (inclusive)

   select * from HR.employees where salary between 9000 and 10000;

3. **Display the name, address,salary of lowest salary person**

   select employee_id,email,salary from HR.employees where salary = (select min(salary) from HR.employees);

4. **Display the name, salary and dept ID of the employee who has highest salary of each dept**

```
SELECT  d.department_id, e.first_name, e.salary FROM  hr.departments d
  INNER JOIN (SELECT department_id, MAX(salary) salary FROM hr.employees GROUP
BY department_id) t ON t.department_id = d.department_id
  INNER JOIN hr.employees e ON e.department_id = t.department_id  AND e.salary =
t.salary;
```

https://www.edureka.co/blog/interview-questions/sql-query-interview-questions

SQL Notes

07 December 2021
19:45

- A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.

- `(INNER) JOIN`: Returns records that have matching values in both tables
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table

ABHISEK

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Indexing

What is an Index in Oracle?

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

The syntax for creating an index in Oracle/PLSQL is:

CREATE [UNIQUE] INDEX index_name ON table_name (column1, column2, ... column_n) [COMPUTE STATISTICS];

UNIQUE	It indicates that the combination of values in the indexed columns must be unique.
index_name	The name to assign to the index.
table_name	The name of the table in which to create the index.
column1, column2, ... column_n	The columns to use in the index.
COMPUTE STATISTICS	It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

For example:

CREATE INDEX supplier_idx ON supplier (supplier_name);

In this example, we've created an index on the supplier table called supplier_idx. It consists of only one field - the supplier_name field.

We could also create an index with more than one field as in the example below:

CREATE INDEX supplier_idx ON supplier (supplier_name, city);

We could also choose to collect statistics upon creation of the index as follows:

CREATE INDEX supplier_idx ON supplier (supplier_name, city) COMPUTE STATISTICS;

https://www.edureka.co/blog/interview-questions/sql-query-interview-questions

REST API

25 April 2022
18:18

Idempotent in REST API.

REST API Consuming

12:15

WebClient:

Spring WebClient is a non-blocking and reactive web client to perform HTTP requests. WebClient has been added in *Spring 5 (spring-webflux module)* and provides *fluent functional style API*.

```
WebClient webClient = WebClient.create("http://localhost:3000");
Employee createdEmployee = webClient.post() .uri("/employees")
.header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
.body(Mono.just(empl), Employee.class) .retrieve()
.bodyToMono(Employee.class);
```

```
@Service
publicclassMyService {
privatefinalWebClient webClient;
publicMyService(WebClient.Builder webClientBuilder) {
                this.webClient =
webClientBuilder.baseUrl("https://example.org").build();
        }
publicMono<Details> someRestCall(String name) {
                returnthis.webClient.get().uri("/{name}/details", name)
                                                .retrieve().bodyToMono(
Details.class);
        }
}
```

ARHISEK

HTTP Status Codes

14 July 2022
15:19

What Are HTTP Status Codes?

Simply put, an HTTP Status Code refers to a 3-digit code that is part of a server's HTTP Response. The first digit of the code describes the category in which the response falls. This already gives a hint to determine whether the request was successful or not.

Below are the different categories:

1. **Informational (1xx)**: Indicates that the request was received and the process is continuing. It alerts the sender to wait for a final response.
2. **Successful (2xx)**: Indicates that the request was successfully received, understood, and accepted.
3. **Redirection (3xx)**: Indicates that further action must be taken to complete the request.
4. **Client Errors (4xx)**: Indicates that an error occurred during the request processing and it is the client who caused the error.
5. **Server Errors (5xx)**: Indicates that an error occurred during request processing but that it was by the server.

While the list is hardly exhaustive, here are some of the most common HTTP codes you'll be running into:

Code	Status	Description
200	OK	The request was successfully completed.
201	Created	A new resource was successfully created.
400	Bad Request	The request was invalid.
401	Unauthorized	The request did not include an authentication token or the authentication token was expired.
403	Forbidden	The client did not have permission to access the requested resource.

ABHISEK

404	Not Found	The requested resource was not found.
405	Method Not Allowed	The HTTP method in the request was not supported by the resource. For example, the DELETE method cannot be used with the Agent API.
500	Internal Server Error	The request was not completed due to an internal error on the server side.
503	Service Unavailable	The server was unavailable.

ABHISEK

Interview Programs

21 September 2022
16:10

Rotate An Array :

Array : 1,2,3,4,5

n = 5 : length of array

k time ?
k =1 : 51234
k=2 : 45123

Rotate Single Linked List

External Server Configuration

07 December 2021
08:00

First, we need to package a WAR application instead of a JAR. For this, we change *pom.xml* with the following content:

```
<packaging>war</packaging>
```

Now, let's modify the final *WAR* file name to avoid including version numbers:

```
<build>
```

```
    <finalName>${artifactId}</finalName>
    ...
</build>
```

Then, we're going to add the Tomcat dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```

Finally, we initialize the Servlet context required by Tomcat by implementing the *SpringBootServletInitializer* interface:

```
@SpringBootApplication
Public class SpringBootTomcatApplication extends
SpringBootServletInitializer{

}
```

ABHISEK