Basics of JAVA

1. Compare JVM, JRE & JDK

**JAVA DEVELOPMENT KIT** : The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

**JAVA RUNTIME ENVIRONMENT**: JRE stands for Java Runtime Environment and may also be written as Java RTE. The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

**JAVA VIRTUAL MACHINE**:

* A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
* An implementation is a computer program that meets the requirements of the JVM specification
* Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.



* DK: Java Development Kit (in short JDK) is Kit which provides the environment to develop and execute(run) the Java program. JDK is a kit(or package) which includes two things.
  + Development Tools(to provide an environment to develop your java programs).
  + JRE (to execute your java program).

Note : JDK is only used by Java Developers.

* JRE: Java Runtime Environment (to say JRE) is an installation package which provides environment to only run(not develop) the java program(or application)onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* JVM: Java Virtual machine(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line hence it is also known as interpreter.

1. What does JRE consists of?

JRE consists of the following components:

* Deployment technologies, including deployment, Java Web Start and Java Plug-in.
* User interface toolkits, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD) and input methods.
* Integration libraries, including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) and scripting.
* Other base libraries, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization and Java for XML Processing (XML JAXP).
* Lang and util base libraries, including lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects and Regular Expressions.
* Java Virtual Machine (JVM), including Java HotSpot Client and Server Virtual Machines.

1. How does JRE works?

To understand how the JRE works let us consider a Java source file saved as Example.java. The file is compiled into a set of Byte Code that is stored in a .class file. Here it will be Example.class.



The following diagram depicts what is done at compile time.

The following actions occur at runtime.

* **Class Loader**: The Class Loader loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.
* **Byte Code Verifier**: The JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.



* **Intrepreter**: At runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:
  + Execute the Byte Code
  + Make appropriate calls to the underlying hardware

Both operations can be shown as:



To understand the interactions between JDK and JRE consider the following diagram.



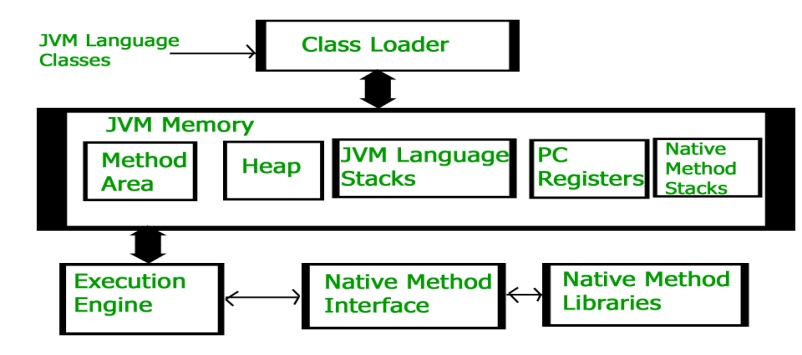
The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.

1. How does JVM works?

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.



**Class Loader Subsystem**

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading** : The Class loader reads the .class file, generate the corresponding binary data and save it in method area. For each .class file, JVM stores following information in method area.

* Fully qualified name of the loaded class and its immediate parent class.
* Whether .class file is related to Class or Interface or Enum
* Modifier, Variables and Method information etc.

After loading .class file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use getClass() method of Object class.

|  |
| --- |
| // A Java program to demonstrate working of a Class type  // object created by JVM to represent .class file in  // memory.  import java.lang.reflect.Field;  import java.lang.reflect.Method;    // Java code to demonstrate use of Class object  // created by JVM  public class Test  {      public static void main(String[] args)      {          Student s1 = new Student();            // Getting hold of Class object created          // by JVM.          Class c1 = s1.getClass();            // Printing type of object using c1.          System.out.println(c1.getName());            // getting all methods in an array          Method m[] = c1.getDeclaredMethods();          for (Method method : m)              System.out.println(method.getName());            // getting all fields in an array          Field f[] = c1.getDeclaredFields();          for (Field field : f)              System.out.println(field.getName());      }  }    // A sample class whose information is fetched above using  // its Class object.  class Student  {      private String name;      private int roll\_No;        public String getName()  {  return name;   }      public void setName(String name) { this.name = name; }      public int getRoll\_no()  { return roll\_No;  }      public void setRoll\_no(int roll\_no) {          this.roll\_No = roll\_no;      }  } |

Output:

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

**Note :** For every loaded *.class* file, only **one** object of Class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking** : Performs verification, preparation, and (optionally) resolution.

* Verification : It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError.
* Preparation : JVM allocates memory for class variables and initializing the memory to default values.
* Resolution : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization** : In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in class hierarchy.

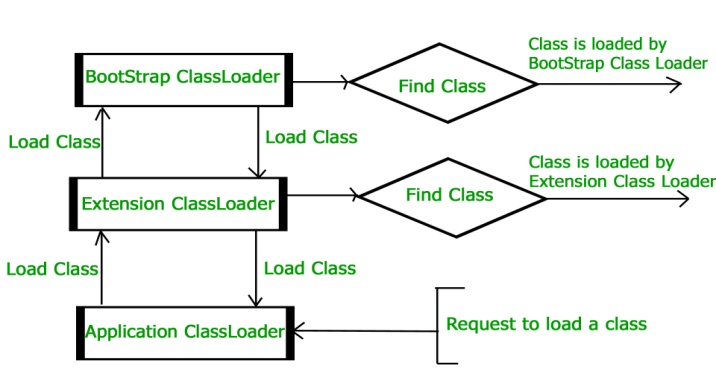
In general, there are three class loaders :

* **Bootstrap class loader** : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in JAVA\_HOME/jre/lib directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
* **Extension class loader** : It is child of bootstrap class loader. It loads the classes present in the extensions directories JAVA\_HOME/jre/lib/ext(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.
* **System/Application class loader** : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

|  |
| --- |
| // Java code to demonstrate Class Loader subsystem  public class Test  {      public static void main(String[] args)      {          // String class is loaded by bootstrap loader, and          // bootstrap loader is not Java object, hence null          System.out.println(String.class.getClassLoader());            // Test class is loaded by Application loader          System.out.println(Test.class.getClassLoader());      }  } |

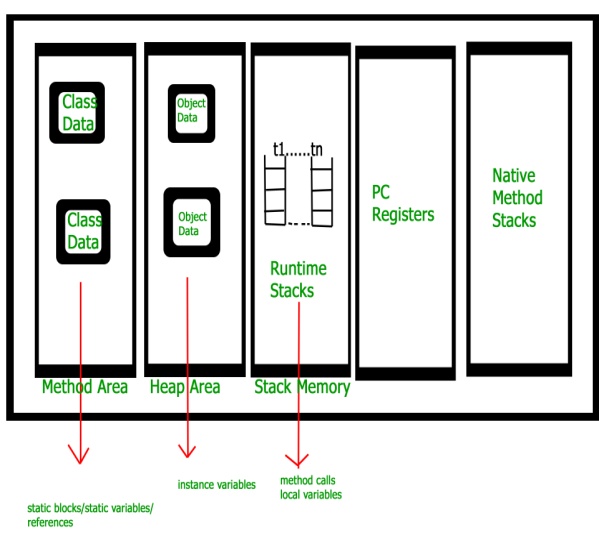
* Output:
* null
* sun.misc.Launcher$AppClassLoader@73d16e93

**Note** : JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.



**JVM Memory**

* **Method area** :In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
* **Heap area** :Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.
* **Stack area** :For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.
* **PC Registers** :Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.
* **Native method stacks** :For every thread, separate native stack is created. It stores native method information.



**Execution Engine**

Execution engine execute the .class (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

* **Interpreter** : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
* **Just-In-Time Compiler(JIT)** : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
* **Garbage Collector** : It destroy un-referenced objects.For more on Garbage Collector,refer Garbage Collector.
* **Java Native Interface (JNI)** : It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries** :

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

1. Can i have a multiple main method in Java

Yes, Its possible but have to maintain the different argument list.

1. What is the default class in Java

Object

1. What are the methods available in Object class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

* toString()
* **hashCode()**:For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm. The hashCode() method is native because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.

Use of hashCode() method : Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.

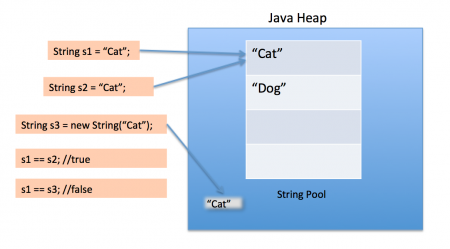
**Note** : Override of hashCode() method needs to be done such that for every object we generate a unique number. For example,for a Student class we can return roll no. of student from hashCode() method as it is unique.

* **equals(Object obj):** Compares the given object to this object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override equals(Object obj) method to get our own equality condition on Objects
* **getClass()**: Returns the class object of this object and used to get actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don’t override it.
* **finalize()** method : This method is called just before an object is garbage collected. It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.
* **clone()** : It returns a new object that is exactly the same as this object. For clone() method refer Clone()
* The remaining three methods **wait(), notify() notifyAll()** are related to Concurrency. Refer Inter-thread Communication in Java for details.

1. What is string pool

As the name suggests, String Pool in java is a pool of Strings stored in Java Heap Memory. We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

Here is a diagram which clearly explains how String Pool is maintained in java heap space and what happens when we use different ways to create Strings.



String Pool is possible only because String is immutable in Java and it’s implementation of String interning concept. String pool is also example of Flyweight design pattern.

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

1. Difference between String, StringBuffer & StringBuilder

**String in Java**

* String class represents character strings, we can instantiate String by two ways.

String str = "abc"; or String str = new String ("abc");

* String is immutable in java, so its easy to share it across different threads or functions.
* When we create a String using double quotes, it first looks for the String with same value in the JVM string pool, if found it returns the reference else it creates the String object and then place it in the String pool. This way JVM saves a lot of space by using same String in different threads. But if new operator is used, it explicitly creates a new String in the heap memory.
* + operator is overloaded for String and used to concatenate two Strings. Although internally it uses StringBuffer to perform this action.
* String overrides equals() and hashCode() methods, two Strings are equal only if they have same characters in same order. Note that equals() method is case sensitive, so if you are not looking for case sensitive checks, you should use equalsIgnoreCase() method.
* A String represents a string in the UTF-16 format
* String is a final class with all the fields as final except “private int hash”. This field contains the hashCode() function value and created only when hashCode() method is called and then cached in this field. Furthermore, hash is generated using final fields of String class with some calculations, so every time hashCode() method is called, it will result in same output. For caller, its like calculations are happening every time but internally it’s cached in hash field.

**String vs StringBuffer**

Since String is immutable in java, whenever we do String manipulation like concat, substring etc, it generates a new String and discard the older String for garbage collection.

These are heavy operations and generate a lot of garbage in heap. So Java has provided StringBuffer and StringBuilder class that should be used for String manipulation.

StringBuffer and StringBuilder are mutable objects in java and provide append(), insert(), delete() and substring() methods for String manipulation.

**StringBuffer vs StringBuilder**

StringBuffer was the only choice for String manipulation till Java 1.4 but it has one disadvantage that all of its public methods are synchronized. StringBuffer provides Thread safety but on a performance cost.

In most of the scenarios, we don’t use String in multithreaded environment, so Java 1.5 introduced a new class StringBuilder that is similar with StringBuffer except thread safety and synchronization.

So if you are in a single threaded environment or don’t care about thread safety, you should use StringBuilder else use StringBuffer. See this post for performance benchmarking between StringBuffer and StringBuilder.

**String vs StringBuffer vs StringBuilder**

* String is immutable whereas StringBuffer and StringBuider are mutable classes.
* StringBuffer is thread safe and synchronized whereas StringBuilder is not, thats why StringBuilder is more faster than StringBuffer.
* String concat + operator internally uses StringBuffer or StringBuilder class.
* For String manipulations in non-multi threaded environment, we should use StringBuilder else use StringBuffer class.

1. Types of modifier

| **Access Modifiers** | **Visibility** |
| --- | --- |
| Public | Visible to All classes. |
| Protected | Visible to classes with in the package and the subclasses of other package. |
| No Access Modifier (Default) | Visible to the classes with the package |
| private | Visible with in the class. It is not accessible outside the class. |

1. What is immutable and how do create the immutable class

An object is immutable if its state cannot change after construction. Immutable objects don’t expose any way for other objects to modify their state; the object’s fields are initialized only once inside the constructor and never change again.

In this article, we'll define the typical steps for creating an immutable class in Java and also shed light on the common mistakes which are made by developers while creating immutable classes.

* **Usage of Immutable Classes**

Nowadays, the “must-have” specification for every software application is to be distributed and multi-threaded—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the Synchronized blocks whenever they modify the state of an object.

With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn’t worry about concurrent modifications.

* **Some Popular Immutable Classes**

String is the most popular immutable class in Java. Once initialized its value cannot be modified. Operations like trim(), substring(), replace() always return a new instance and don’t affect the current instance, that’s why we usually call trim() as the following:

String alex = "Alex";

alex = alex.trim();

Another example from JDK is the wrapper classes like: Integer, Float, Boolean … these classes don’t modify their state, however they create a new instance each time you try to modify them.

Integer a =3;

a += 3;

After calling a += 3, a new instance is created holding the value: 6 and the first instance is lost.

* **How Do We Create an Immutable Class**

In order to create an immutable class, you should follow the below steps:

1. Make your class final, so that no other classes can extend it.
2. Make all your fields final, so that they’re initialized only once inside the constructor and never modified afterward.
3. Don’t expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.
5. If the class holds a mutable object:

Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.

Make sure to always return a clone copy of the field and never return the real object instance.

* **Simple Immutable Class**

Let’s follow the above steps and create our own immutable class (ImmutableStudent.java).

package com.programmer.gate.beans;

public final class ImmutableStudent {

private final int id;

private final String name;

public ImmutableStudent(int id, String name) {

this.name = name;

this.id = id;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

}

The above class is a very simple immutable class which doesn’t hold any mutable object and never expose its fields in any way; these type of classes are normally used for caching purposes.

* **Passing Mutable Objects to Immutable Class**

package com.programmer.gate.beans;

public class Age {

private int day;

private int month;

private int year;

public int getDay() {

return day;

}

public void setDay(int day) {

this.day = day;

}

public int getMonth() {

return month;

}

public void setMonth(int month) {

this.month = month;

}

public int getYear() {

return year;

}

public void setYear(int year) {

this.year = year;

}

}

package com.programmer.gate.beans;

public final class ImmutableStudent {

private final int id;

private final String name;

private final Age age;

public ImmutableStudent(int id, String name, Age age) {

this.name = name;

this.id = id;

this.age = age;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public Age getAge() {

return age;

}

}

So, we added a new mutable field of type Age to our immutable class and assign it as normal inside the constructor.

Let’s create a simple test class and verify that ImmutableStudent is no more immutable:

public static void main(String[] args) {

Age age = new Age();

age.setDay(1);

age.setMonth(1);

age.setYear(1992);

ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

System.out.println("Alex age year before modification = " + student.getAge().getYear());

age.setYear(1993);

System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

After running the above test, we get the following output:

*Alex age year before modification = 1992*

*Alex age year after modification = 1993*

We claim that ImmutableStudent is an immutable class whose state is never modified after construction, however in the above example we are able to modify the age of Alex even after constructing Alex object. If we go back to the implementation of ImmutableStudent constructor, we find that age field is being assigned to the instance of the Age argument, so whenever the referenced Age is modified outside the class, the change is reflected directly on the state of Alex. Check out my Pass by value OR pass by reference article to more deeply understand this concept.

In order to fix this and make our class again immutable, we follow step #5 from the steps that we mention above for creating an immutable class. So we modify the constructor in order to clone the passed argument of Age and use a clone instance of it.

public ImmutableStudent(int id, String name, Age age) {

this.name = name;

this.id = id;

Age cloneAge = new Age();

cloneAge.setDay(age.getDay());

cloneAge.setMonth(age.getMonth());

cloneAge.setYear(age.getYear());

this.age = cloneAge;

}

Now, if we run our test, we get the following output:

Alex age year before modification = 1992

Alex age year after modification = 1992

### Returning Mutable Objects From Immutable Class

However, our class still has a leak and is not fully immutable, let’s take the following test scenario:

public static void main(String[] args) {

Age age = new Age();

age.setDay(1);

age.setMonth(1);

age.setYear(1992);

ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

System.out.println("Alex age year before modification = " + student.getAge().getYear());

student.getAge().setYear(1993);

System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

Output:

Alex age year before modification = 1992

Alex age year after modification = 1993

Again according to step **#4**, when returning mutable fields from immutable object, you should return a clone instance of them and not the real instance of the field.

So we modify ***getAge()***in order to return a clone of the object’s age:

public Age getAge() {

Age cloneAge = new Age();

cloneAge.setDay(this.age.getDay());

cloneAge.setMonth(this.age.getMonth());

cloneAge.setYear(this.age.getYear());

return cloneAge;

}

Now the class becomes fully immutable and provides no way or method for other objects to modify its state.

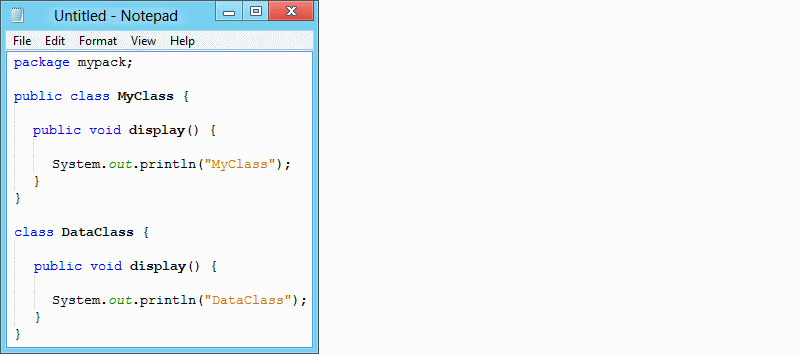
Alex age year before modification = 1992

Alex age year after modification = 1992

1. How do hide the class with in the package (Class should not visible outside the package)

When we import a package within a program, only the classes declared as public in that package will be made accessible within this program. In other words, the classes not declared as public in that package will not be accessible within this program.

We shall profitably make use of the above fact. Sometimes, we may wish that certain classes in a package should not be made accessible to the importing program. In such cases, we need not declare those classes as public. When we do so, those classes will be hidden from being accessed by the importing class. Here is an example :



Here, the class DataClass which is not declared public is hidden from outside of the package mypack. This class can be seen and used only by other classes in the same package. Note that a Java source file should contain only one public class and may include any number of non-public classes.

1. Types of exception in java

**Checked Exception**

What is Checked Exception in Java Programming language. In simple language: Exception which are checked at Compile time called Checked Exception. Some these are mentioned below. If in your code if some of method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

* IOException
* SQLException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException
* MalformedURLException

**Unchecked Exception**

Unchecked Exception in Java is those Exceptions whose handling is NOT verified during Compile time. These exceptions occurs because of bad programming. The program won’t give a compilation error.

All Unchecked exceptions are direct sub classes of RuntimeException class.

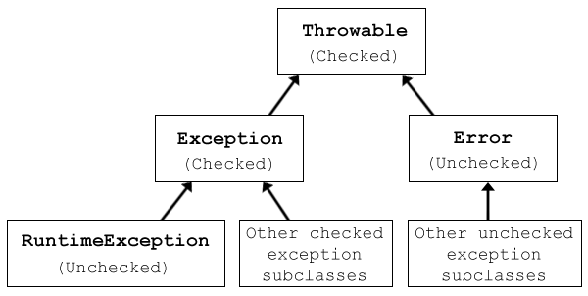
* NullPointerException
* ArrayIndexOutOfBound
* IllegalArgumentException
* IllegalStateException

1. Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |
| --- | --- |
| Throw | Throws |
| Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| Throw is followed by an instance. | Throws is followed by class. |
| Throw is used within the method. | Throws is used with the method |
| You cannot throw multiple exceptions. | You can declare multiple exceptions |

1. Tell me the exception hierarchy



1. Compare Finally, Finalize & Final

**Final**: Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

**Finally**: Finally is used to place important code, it will be executed whether exception is handled or not.

**Finalize**: Finalize is used to perform clean up processing just before object is garbage collected.

Final is a keyword. Finally is a block. Finalize is a method.

1. Compare ClasssNotFound & NoClassDefFound exceptions

ClassNotFoundException and NoClassDefFoundError occur when a particular class is not found at runtime. However, they occur at different scenarios.

ClassNotFoundException is an exception that occurs when you try to load a class at run time using Class.forName() or loadClass() methods and mentioned classes are not found in the classpath.

NoClassDefFoundError is an error that occurs when a particular class is present at compile time, but was missing at run time.

**ClassNotFoundException**

ClassNotFoundException is a runtime exception that is thrown when an application tries to load a class at runtime using the Class.forName() or loadClass() or findSystemClass() methods ,and the class with specified name are not found in the classpath. For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. Most of the time, this exception occurs when you try to run an application without updating the classpath with required JAR files.

For example, the below program will throw ClassNotFoundException if the mentioned class “oracle.jdbc.driver.OracleDriver” is not found in the classpath.

public class MainClass

{

public static void main(String[] args)

{

try

{

Class.forName("oracle.jdbc.driver.OracleDriver");

}catch (ClassNotFoundException e)

{

e.printStackTrace();

}

}

}

**NoClassDefFoundError**

NoClassDefFoundError is an error that is thrown when the Java Runtime System tries to load the definition of a class, and that class definition is no longer available. The required class definition was present at compile time, but it was missing at runtime. For example, compile the program below.

class A

{

// some code

}

public class B

{

public static void main(String[] args)

{

A a = new A();

}

}

When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you remove the A.class file and run the B.class file, Java Runtime System will throw NoClassDefFoundError like below:

1. Is system allows multiple catch
2. Is it possible to write try with out catch block

It is possible to have try and finally blocks without catch.

1. How do create a static class

Java allows us to define a class within another class. Such a class is called a nested class. The class which enclosed nested class is known as Outer class. In java, we can’t make Top level class static. Only nested classes can be static.

1. What are the differences between static and non-static nested classes?

Following are major differences between static nested class and non-static nested class. Non-static nested class is also called Inner Class.

1) Nested static class doesn’t need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.

2) Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

3) An instance of Inner class cannot be created without an instance of outer class and an Inner class can reference data and methods defined in Outer class in which it nests, so we don’t need to pass reference of an object to the constructor of the Inner class. For this reason Inner classes can make program simple and concise.

/\* Java program to demonstrate how to implement static and non-static classes in a java program. \*/

class OuterClass

{

private static String msg = "GeeksForGeeks";

// Static nested class

public static class NestedStaticClass{

// Only static members of Outer class is directly accessible in nested

// static class

public void printMessage()

{

// Try making 'message' a non-static variable, there will be

// compiler error

System.out.println("Message from nested static class: " + msg);

}

}

// non-static nested class - also called Inner class

public class InnerClass

{

// Both static and non-static members of Outer class are accessible in

// this Inner class

public void display(){

System.out.println("Message from non-static nested class: "+ msg);

}

}

}

class Main

{

// How to create instance of static and non static nested class?

public static void main(String args[]){

// create instance of nested Static class

OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();

// call non static method of nested static class

printer.printMessage();

// In order to create instance of Inner class we need an Outer class

// instance. Let us create Outer class instance for creating

// non-static nested class

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

// calling non-static method of Inner class

inner.display();

// we can also combine above steps in one step to create instance of

// Inner class

OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();

// similarly we can now call Inner class method

innerObject.display();

}

}

1. When you go to static method

Instance method are methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in within which it defined.

**Memory allocation**: These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack. They can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depend on the access type provided to the desired instance method.

**Important Points**:

* Instance method(s) belong to the Object of the class not to the class i.e. they can be called after creating the Object of the class.
* Every individual Object created from the class has its own copy of the instance method(s) of that class.
* They can be overridden since they are resolved using dynamic binding at run time.
* filter\_none

1. Explain Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the java.io.Serializable interface.

The ObjectOutputStream class contains writeObject() method for serializing an Object.

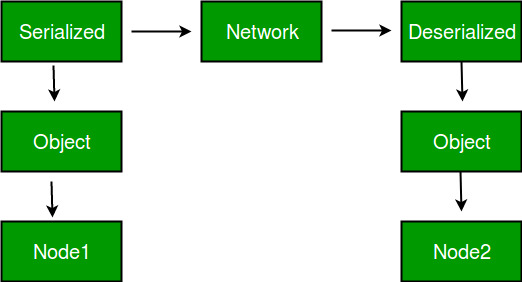
public final void writeObject(Object obj) throws IOException

public final Object readObject()throws IOException, ClassNotFoundException

Advantages of Serialization

1. To save/persist state of an object.

2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing java.io.Serializable interface.

Serializable is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember

1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.

4. Constructor of object is never called when an object is deserialized.

5. Associated objects must be implementing Serializable interface.

**SerialVersionUID:** The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an InvalidClassException. A Serializable class can declare its own UID explicitly by declaring a field name.

It must be static, final and of type long.

i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

**serialver**

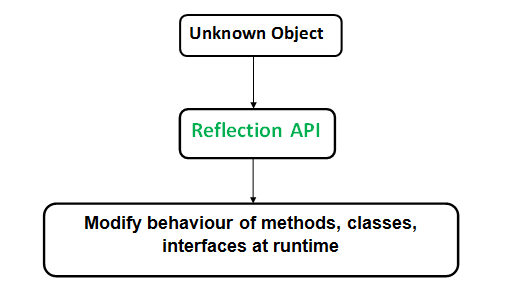
The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

You can run the following command to get serialVersionUID

1. What is reflection

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

* The required classes for reflection are provided under java.lang.reflect package.
* Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
* Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.



Reflection can be used to get information about

* Class The getClass() method is used to get the name of the class to which an object belongs.
* Constructors The getConstructors() method is used to get the public constructors of the class to which an object belongs.
* Methods The getMethods() method is used to get the public methods of the class to which an objects belongs.

1. Can we reflect the private member values

Possible from Java 9 onwards

This only works when running the code as a standalone Java application, like you do with unit tests and regular applications. If you try to do this inside a Java Applet, you will need to fiddle around with the SecurityManager. But, since that is not something you need to do very often, it is left out of this text so far.

Note: There has been a lot of talk about disabling the ability to access private fields via reflection from Java 9. From my experiments it seems to still be possible in Java 9, but be aware that this might change in a future Java version..

Accessing Private Fields

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Field privateStringField = PrivateObject.class.

getDeclaredField("privateString");

**privateStringField.setAccessible(true);**

String fieldValue = (String) privateStringField.get(privateObject);

System.out.println("fieldValue = " + fieldValue);

**Accessing Private Methods**

public class PrivateObject {

private String privateString = null;

public PrivateObject(String privateString) {

this.privateString = privateString;

}

private String getPrivateString(){

return this.privateString;

}

}

PrivateObject privateObject = new PrivateObject("The Private Value");

Method privateStringMethod = PrivateObject.class.

getDeclaredMethod("getPrivateString", null);

**privateStringMethod.setAccessible(true);**

String returnValue = (String)

privateStringMethod.invoke(privateObject, null);

System.out.println("returnValue = " + returnValue);

1. Types of memory in Java

* Class Memory
* Method Memory
* Heap Memory
* Stack Memory

**Java Heap Space**

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.

Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.

**Java Stack Memory**

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method.

As soon as method ends, the block becomes unused and become available for next method.

Stack memory size is very less compared to Heap memory.

**Heap and Stack Memory in Java Program**

Let’s understand the Heap and Stack memory usage with a simple program.

package com.journaldev.test;

public class Memory {

public static void main(String[] args) { // Line 1

int i=1; // Line 2

Object obj = new Object(); // Line 3

Memory mem = new Memory(); // Line 4

mem.foo(obj); // Line 5

} // Line 9

private void foo(Object param) { // Line 6

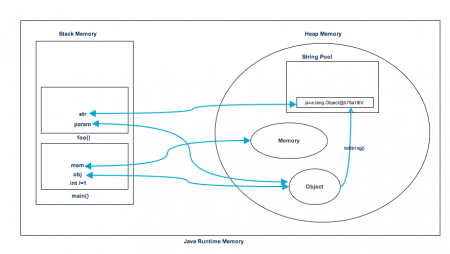
String str = param.toString(); //// Line 7

System.out.println(str);

} // Line 8

}

Below image shows the Stack and Heap memory with reference to above program and how they are being used to store primitive, Objects and reference variables.



Let’s go through the steps of execution of the program.

* As soon as we run the program, it loads all the Runtime classes into the Heap space. When main() method is found at line 1, Java Runtime creates stack memory to be used by main() method thread.
* We are creating primitive local variable at line 2, so it’s created and stored in the stack memory of main() method.
* Since we are creating an Object in line 3, it’s created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
* Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
* A string is created in line 7, it goes in the String Pool in the heap space and a reference is created in the foo() stack space for it.
* foo() method is terminated in line 8, at this time memory block allocated for foo() in stack becomes free.
* In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also the program ends at this line, hence Java Runtime frees all the memory and end the execution of the program.

**Difference between Java Heap Space and Stack Memory**

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

* Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
* Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
* Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
* Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at Java Garbage Collection.
* Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
* We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.
* When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
* Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

1. How do find the memory leakage in java application before and after implementation.
2. How GC is working

Java Memory Management, with its built-in garbage collection, is one of the language's finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. To wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

How Garbage Collection Really Works

Many people think garbage collection collects and discards dead objects. In reality, Java garbage collection is doing the opposite! Live objects are tracked and everything else designated garbage. As you'll see, this fundamental misunderstanding can lead to many performance problems.

Let's start with the heap, which is the area of memory used for dynamic allocation. In most configurations the operating system allocates the heap in advance to be managed by the JVM while the program is running. This has a couple of important ramifications:

* Object creation is faster because global synchronization with the operating system is not needed for every single object. An allocation simply claims some portion of a memory array and moves the offset pointer forward (see Figure 2.1). The next allocation starts at this offset and claims the next portion of the array.
* When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion and no memory is given back to the operating system.

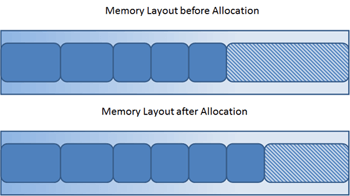


Figure 2.1: New objects are simply allocated at the end of the used heap.

All objects are allocated on the heap area managed by the JVM. Every item that the developer uses is treated this way, including class objects, static variables, and even the code itself. As long as an object is being referenced, the JVM considers it alive. Once an object is no longer referenced and therefore is not reachable by the application code, the garbage collector removes it and reclaims the unused memory. As simple as this sounds, it raises a question: what is the first reference in the tree?

**Garbage-Collection Roots—The Source of All Object Trees**

Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable. But when are those root objects considered reachable? Special objects called garbage-collection roots (GC roots; see Figure 2.2) are always reachable and so is any object that has a garbage-collection root at its own root.

There are four kinds of GC roots in Java:

* **Local variables** are kept alive by the stack of a thread. This is not a real object virtual reference and thus is not visible. For all intents and purposes, local variables are GC roots.
* **Active Java threads** are always considered live objects and are therefore GC roots. This is especially important for thread local variables.
* **Static variables** are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables. This is of special importance when we use application servers, OSGi containers or class loaders in general. We will discuss the related problems in the Problem Patterns section.
* **JNI References** are Java objects that the native code has created as part of a JNI call. Objects thus created are treated specially because the JVM does not know if it is being referenced by the native code or not. Such objects represent a very special form of GC root, which we will examine in more detail in the Problem Patterns section below.

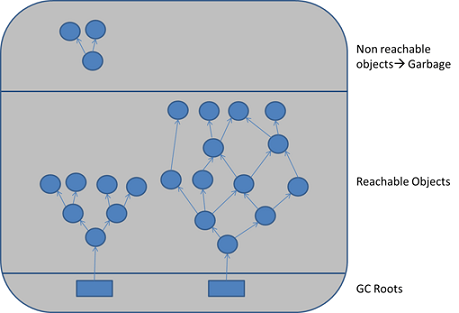


Figure 2.2: GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected.

Therefore, a simple Java application has the following GC roots:

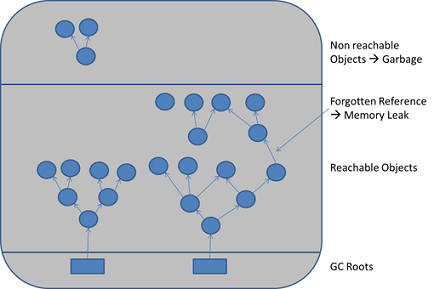
* Local variables in the main method
* The main thread
* Static variables of the main class

**Marking and Sweeping Away Garbage**

To determine which objects are no longer in use, the JVM intermittently runs what is very aptly called a mark-and-sweep algorithm. As you might intuit, it's a straightforward, two-step process:

* The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.
* All of the heap memory that is not occupied by marked objects is reclaimed. It is simply marked as free, essentially swept free of unused objects.

Garbage collection is intended to remove the cause for classic memory leaks: unreachable-but-not-deleted objects in memory. However, this works only for memory leaks in the original sense. It's possible to have unused objects that are still reachable by an application because the developer simply forgot to dereference them. Such objects cannot be garbage-collected. Even worse, such a logical memory leak cannot be detected by any software (see Figure 2.3). Even the best analysis software can only highlight suspicious objects. We will examine memory leak analysis in the Analyzing the Performance Impact of Memory Utilization and Garbage Collection section, below.



When objects are no longer referenced directly or indirectly by a GC root, they will be removed. There are no classic memory leaks. Analysis cannot really identify memory leaks; it can only point out suspicious objects.

**A Definition of Java Garbage Collection**

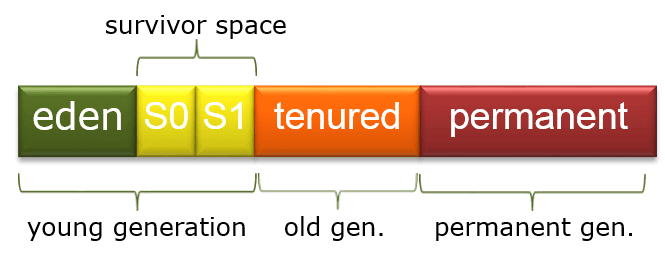
Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

**How Java Garbage Collection Works**

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted. The garbage collection implementation lives in the JVM. Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification. Although there are many JVMs, Oracle’s HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process. In the first step, unreferenced objects are identified and marked as ready for garbage collection. In the second step, marked objects are deleted. Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap. The compaction process makes it easier to allocate memory to new objects sequentially after the block of memory allocated to existing objects.

All of HotSpot’s garbage collectors implement a generational garbage collection strategy that categorizes objects by age. The rationale behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.



The heap is divided into three sections:

* **Young Generation**: Newly created objects start in the Young Generation. The Young Generation is further subdivided into an Eden space, where all new objects start, and two Survivor spaces, where objects are moved from Eden after surviving one garbage collection cycle. When objects are garbage collected from the Young Generation, it is a minor garbage collection event.
* **Old Generation**: Objects that are long-lived are eventually moved from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.
* **Permanent Generation**: Metadata such as classes and methods are stored in the Permanent Generation. Classes that are no longer in use may be garbage collected from the Permanent Generation.

During a full garbage collection event, unused objects in all generations are garbage collected.

HotSpot has four garbage collectors:

* **Serial**: All garbage collection events are conducted serially in one thread. Compaction is executed after each garbage collection.
* **Parallel**: Multiple threads are used for minor garbage collection. A single thread is used for major garbage collection and Old Generation compaction. Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.
* **CMS (Concurrent Mark Sweep):** Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old, but CMS runs concurrently alongside application processes to minimize “stop the world” events (i.e. when the garbage collector running stops the application). No compaction is performed.
* **G1 (Garbage First):** The newest garbage collector is intended as a replacement for CMS. It is parallel and concurrent like CMS, but it works quite differently under the hood compared to the older garbage collectors.

**Benefits of Java Garbage Collection**

The biggest benefit of Java garbage collection is that it automatically handles deletion of unused objects or objects that are out of reach to free up vital memory resources. Programmers working in languages without garbage collection (like C and C++) must implement manual memory management in their code.

Despite the extra work required, some programmers argue in favor of manual memory management over garbage collection, primarily for reasons of control and performance. While the debate over memory management approaches continues to rage on, garbage collection is now a standard component of many popular programming languages. For scenarios in which the garbage collector is negatively impacting performance, Java offers many options for tuning the garbage collector to improve its efficiency.

**Java Garbage Collection Best Practices**

For many simple applications, Java garbage collection is not something that a programmer needs to consciously consider. However, for programmers who want to advance their Java skills, it is important to understand how Java garbage collection works and the ways in which it can be tuned.

Besides the basic mechanisms of garbage collection, one of the most important points to understand about garbage collection in Java is that it is non-deterministic, and there is no way to predict when garbage collection will occur at run time. It is possible to include a hint in the code to run the garbage collector with the System.gc() or Runtime.gc() methods, but they provide no guarantee that the garbage collector will actually run.

The best approach to tuning Java garbage collection is setting flags on the JVM. Flags can adjust the garbage collector to be used (e.g. Serial, G1, etc.), the initial and maximum size of the heap, the size of the heap sections (e.g. Young Generation, Old Generation), and more. The nature of the application being tuned is a good initial guide to settings. For example, the Parallel garbage collector is efficient but will frequently cause “stop the world” events, making it better suited for backend processing where long pauses for garbage collection are acceptable.

On the other hand, the CMS garbage collector is designed to minimize pauses, making it ideal for GUI applications where responsiveness is important. Additional fine-tuning can be accomplished by changing the size of the heap or its sections and measuring garbage collection efficiency using a tool like jstat.

1. How do destroy the object and the values in java

Use Finalize method.

1. What is use of volatile variable

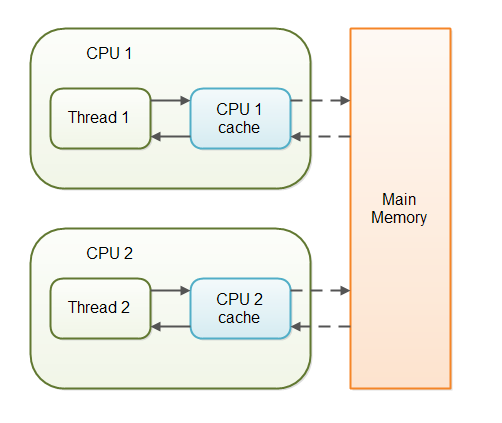
The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Actually, since Java 5 the volatile keyword guarantees more than just that volatile variables are written to and read from main memory. I will explain that in the following sections.

**Variable Visibility Problems**

The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

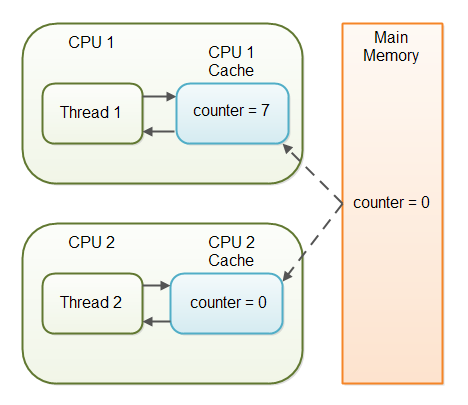
public class SharedObject {

public int counter = 0;

}

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

**The Java volatile Visibility Guarantee**

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

Here is how the volatile declaration of the counter variable looks:

public class SharedObject {

public **volatile** int counter = 0;

}

Declaring a variable volatile thus guarantees the visibility for other threads of writes to that variable.

In the scenario given above, where one thread (T1) modifies the counter, and another thread (T2) reads the counter (but never modifies it), declaring the counter variable volatile is enough to guarantee visibility for T2 of writes to the counter variable.

If, however, both T1 and T2 were incrementing the counter variable, then declaring the counter variable volatile would not have been enough. More on that later.

**Full volatile Visibility Guarantee**

Actually, the visibility guarantee of Java volatile goes beyond the volatile variable itself. The visibility guarantee is as follows:

* If Thread A writes to a volatile variable and Thread B subsequently reads the same volatile variable, then all variables visible to Thread A before writing the volatile variable, will also be visible to Thread B after it has read the volatile variable.
* If Thread A reads a volatile variable, then all all variables visible to Thread A when reading the volatile variable will also be re-read from main memory.

Let me illustrate that with a code example:

public class MyClass {

private int years;

private int months

private volatile int days;

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

The udpate() method writes three variables, of which only days is volatile.

The full volatile visibility guarantee means, that when a value is written to days, then all variables visible to the thread are also written to main memory. That means, that when a value is written to days, the values of years and months are also written to main memory.

When reading the values of years, months and days you could do it like this:

public class MyClass {

private int years;

private int months

private volatile int days;

**public int totalDays() {**

**int total = this.days;**

**total += months \* 30;**

**total += years \* 365;**

**return total;**

**}**

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

Notice the totalDays() method starts by reading the value of days into the total variable. When reading the value of days, the values of months and years are also read into main memory. Therefore you are guaranteed to see the latest values of days, months and years with the above read sequence.

**The Java volatile Happens-Before Guarantee**

To address the instruction reordering challenge, the Java volatile keyword gives a "happens-before" guarantee, in addition to the visibility guarantee. The happens-before guarantee guarantees that:

Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable.

The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.

Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.

The above happens-before guarantee assures that the visibility guarantee of the volatile keyword are being enforced.

**volatile is Not Always Enough**

Even if the volatile keyword guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory, there are still situations where it is not enough to declare a variable volatile.

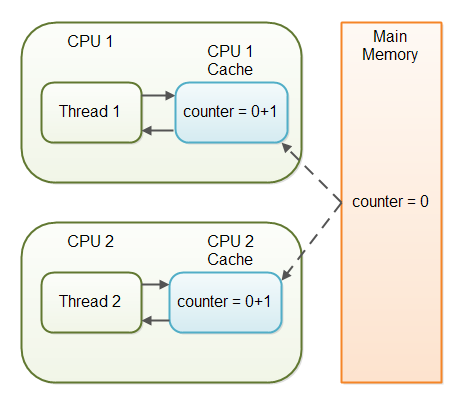
In the situation explained earlier where only Thread 1 writes to the shared counter variable, declaring the counter variable volatile is enough to make sure that Thread 2 always sees the latest written value.

In fact, multiple threads could even be writing to a shared volatile variable, and still have the correct value stored in main memory, if the new value written to the variable does not depend on its previous value. In other words, if a thread writing a value to the shared volatile variable does not first need to read its value to figure out its next value.

As soon as a thread needs to first read the value of a volatile variable, and based on that value generate a new value for the shared volatile variable, a volatile variable is no longer enough to guarantee correct visibility. The short time gap in between the reading of the volatile variable and the writing of its new value, creates an race condition where multiple threads might read the same value of the volatile variable, generate a new value for the variable, and when writing the value back to main memory - overwrite each other's values.

The situation where multiple threads are incrementing the same counter is exactly such a situation where a volatile variable is not enough. The following sections explain this case in more detail.

Imagine if Thread 1 reads a shared counter variable with the value 0 into its CPU cache, increment it to 1 and not write the changed value back into main memory. Thread 2 could then read the same counter variable from main memory where the value of the variable is still 0, into its own CPU cache. Thread 2 could then also increment the counter to 1, and also not write it back to main memory. This situation is illustrated in the diagram below:



Thread 1 and Thread 2 are now practically out of sync. The real value of the shared counter variable should have been 2, but each of the threads has the value 1 for the variable in their CPU caches, and in main memory the value is still 0. It is a mess! Even if the threads eventually write their value for the shared counter variable back to main memory, the value will be wrong.

**When is volatile Enough?**

As I have mentioned earlier, if two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use a synchronized in that case to guarantee that the reading and writing of the variable is atomic. Reading or writing a volatile variable does not block threads reading or writing. For this to happen you must use the synchronized keyword around critical sections.

As an alternative to a synchronized block you could also use one of the many atomic data types found in the java.util.concurrent package. For instance, the AtomicLong or AtomicReference or one of the others.

In case only one thread reads and writes the value of a volatile variable and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

The volatile keyword is guaranteed to work on 32 bit and 64 variables.

**Performance Considerations of volatile**

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use volatile variables when you really need to enforce visibility of variables.

1. Types of itration is available in Java

**Internal vs. External Iteration**

**External iteration**

Till Java 7, the collections framework relied on the concept of external iteration, where a Collection provides, by implementing Iterable, a means to enumerate its elements i.e. Iterator, and clients use this to step sequentially through the elements of a collection. For example, if we wanted to get all strings in uppercase, we would write:

public class IterationExamples {

    public static void main(String[] args)

{

       List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});

        for(String letter: alphabets){

            System.out.println(letter.toUpperCase());

        }

    }

}

OR we can write like this:

public class IterationExamples {

    public static void main(String[] args){

       List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});

        Iterator<String> iterator = alphabets.listIterator();

        while(iterator.hasNext()){

            System.out.println(iterator.next().toUpperCase());

        }

    }

}

Above both code snippets are for external iteration. External iteration is straightforward enough, but it has several problems:

1) Java’s for-each loop/iterator is inherently sequential, and must process the elements in the order specified by the collection.

2) It limits the opportunity to manage the control flow, which might be able to provide better performance by exploiting reordering of the data, parallelism, short-circuiting, or laziness.

**Internal iteration**

Sometimes the strong guarantees of the for-each loop (sequential, in-order) are desirable, but often are just an disadvantage to performance. The alternative to external iteration is internal iteration, where instead of controlling the iteration, client let it handle by library and only provide the code which must be executed for all/some of data elements.

The internal-iteration equivalent of the previous example is:

public class IterationExamples {

    public static void main(String[] args){

       List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});

        alphabets.forEach(l -> l.toUpperCase());

    }

}

1. How do lock a object in java

**Object level and Class level locks in Java**

**Synchronization**: Synchronization is a modifier which is used for method and block only. With the help of synchronized modifier we can restrict a shared resource to be accessed only by one thread. When two or more threads need access to shared resources, there is some loss of data i.e. data inconsistency. The process by which we can achieve data consistency between multiple threads it is called Synchronization.

**Why do you need Synchronization?**

Let us assume if you have two threads that are reading and writing to the same ‘resource’. Suppose there is a variable named as geek, and you want that at one time only one thread should access the variable(atomic way). But Without the synchronized keyword, your thread 1 may not see the changes thread 2 made to geek, or worse, it may only be half changed that cause the data inconsistency problem. This would not be what you logically expect. The tool needed to prevent these errors is synchronization.

**In synchronization, there are two types of locks on threads:**

1. **Object level lock** : Every object in java has a unique lock. Whenever we are using synchronized keyword, then only lock concept will come in the picture. If a thread wants to execute synchronized method on the given object. First, it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock. Acquiring and release lock internally is taken care by JVM and programmer is not responsible for these activities. Lets have a look on the below program to understand the object level lock:

|  |
| --- |
| class Geek implements Runnable {      public void run()      {          Lock();      }      public void Lock()      {          System.out.println(Thread.currentThread().getName());          synchronized(this)          {              System.out.println("in block "                  + Thread.currentThread().getName());              System.out.println("in block " +                  Thread.currentThread().getName() + " end");          }      }        public static void main(String[] args)      {          Geek g = new Geek();          Thread t1 = new Thread(g);          Thread t2 = new Thread(g);          Geek g1 = new Geek();          Thread t3 = new Thread(g1);          t1.setName("t1");          t2.setName("t2");          t3.setName("t3");          t1.start();          t2.start();          t3.start();      }  } |

Output:

t1

in block t1

in block t1 end

t2

in block t2

in block t2 end

t3

in block t3

in block t3 end

1. **Class level lock** : Every class in java has a unique lock which is nothing but class level lock. If a thread wants to execute a static synchronized method, then thread requires class level lock. Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock. Lets look on the below program for better understanding:

|  |
| --- |
| class Geek implements Runnable {      public void run()      {          Lock();      }        public void Lock()      {          System.out.println(Thread.currentThread().getName());          synchronized(Geek.class)          {              System.out.println("in block "                  + Thread.currentThread().getName());              System.out.println("in block "                  + Thread.currentThread().getName() + " end");          }      }        public static void main(String[] args)      {          Geek g1 = new Geek();          Thread t1 = new Thread(g1);          Thread t2 = new Thread(g1);          Geek g2 = new Geek();          Thread t3 = new Thread(g2);          t1.setName("t1");          t2.setName("t2");          t3.setName("t3");          t1.start();          t2.start();          t3.start();      }  } |

Output:

t1

in block t1

in block t1 end

t2

in block t2

in block t2 end

t3

in block t3

in block t3 end

1. Difference between synchronized & non synchronized class

Non synchronized -It is not-thread safe and can't be shared between many threads without proper synchronization code. While, Synchronized- It is thread-safe and can be shared with many threads.

public synchronized void send(String msg)

class Sender

{

public void send(String msg)

{

synchronized(this)

{

System.out.println("Sending\t" + msg );

try

{

Thread.sleep(1000);

}

catch (Exception e)

{

System.out.println("Thread interrupted.");

}

System.out.println("\n" + msg + "Sent");

}

}

}

1. Difference between prepared statement & statement

The Prepared Statement is a slightly more powerful version of a Statement, and should always be at least as quick and easy to handle as a Statement.

The Prepared Statement may be parametrized

Most relational databases handles a JDBC / SQL query in four steps:

* Parse the incoming SQL query
* Compile the SQL query
* Plan/optimize the data acquisition path
* Execute the optimized query / acquire and return data

A Statement will always proceed through the four steps above for each SQL query sent to the database. A Prepared Statement pre-executes steps (1) - (3) in the execution process above. Thus, when creating a Prepared Statement some pre-optimization is performed immediately. The effect is to lessen the load on the database engine at execution time.

Some of the benefits of PreparedStatement over Statement are:

* PreparedStatement helps us in preventing SQL injection attacks because it automatically escapes the special characters.
* PreparedStatement allows us to execute dynamic queries with parameter inputs.
* PreparedStatement provides different types of setter methods to set the input parameters for the query.
* PreparedStatement is faster than Statement. It becomes more visible when we reuse the PreparedStatement or use it’s batch processing methods for executing multiple queries.
* PreparedStatement helps us in writing object Oriented code with setter methods whereas with Statement we have to use String Concatenation to create the query. If there are multiple parameters to set, writing Query using String concatenation looks very ugly and error prone.

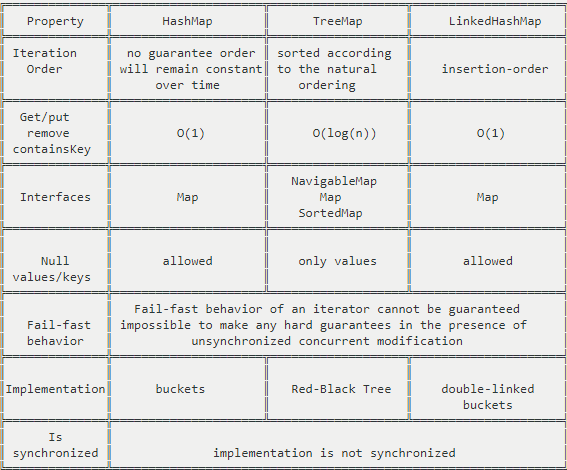
OOPS

1. Compare Association, Aggregation & Composition
2. What is Encaptulation
3. When you go for Abstract class
4. Compare Abstract & Interface
5. Why java is not allowing multiple inheritance
6. How do achieve multiple inheritance in java
7. What is default extended class
8. Can I override the static methods

Collections

1. Compare HashTable, HashMap, LinkedHashMap & TreeMap

* HashTable
  + Doesn’t allow key value as NULL
* HashMap
  + Allow single NULL as a key
  + Order not maintains
  + Faster than LinkedHashMap
  + Used for store heap of objects
* LinkedHashMap
  + LinkedHashMap insertion order will be maintained
  + Slower than HashMap and faster than TreeMap
  + If you want to maintain an insertion order use this.
* TreeMap
  + TreeMap is a tree-based mapping
  + TreeMap will follow the natural ordering of key
  + Slower than HashMap and LinkedHashMap
  + Use TreeMap when you need to maintain natural(default) ordering



1. Compare HashSet, TreeSet

HashSet is much faster than TreeSet (constant-time versus log-time for most operations like add, remove and contains) but offers no ordering guarantees like TreeSet.

**HashSet**

* the class offers constant time performance for the basic operations (add, remove, contains and size).
* it does not guarantee that the order of elements will remain constant over time
* iteration performance depends on the initial capacity and the load factor of the HashSet.
* It's quite safe to accept default load factor but you may want to specify an initial capacity that's about twice the size to which you expect the set to grow.

**TreeSet**

* guarantees log(n) time cost for the basic operations (add, remove and contains)
* guarantees that elements of set will be sorted (ascending, natural, or the one specified by you via its constructor) (implements SortedSet)
* doesn't offer any tuning parameters for iteration performance
* offers a few handy methods to deal with the ordered set like first(), last(), headSet(), and tailSet() etc

**Important points:**

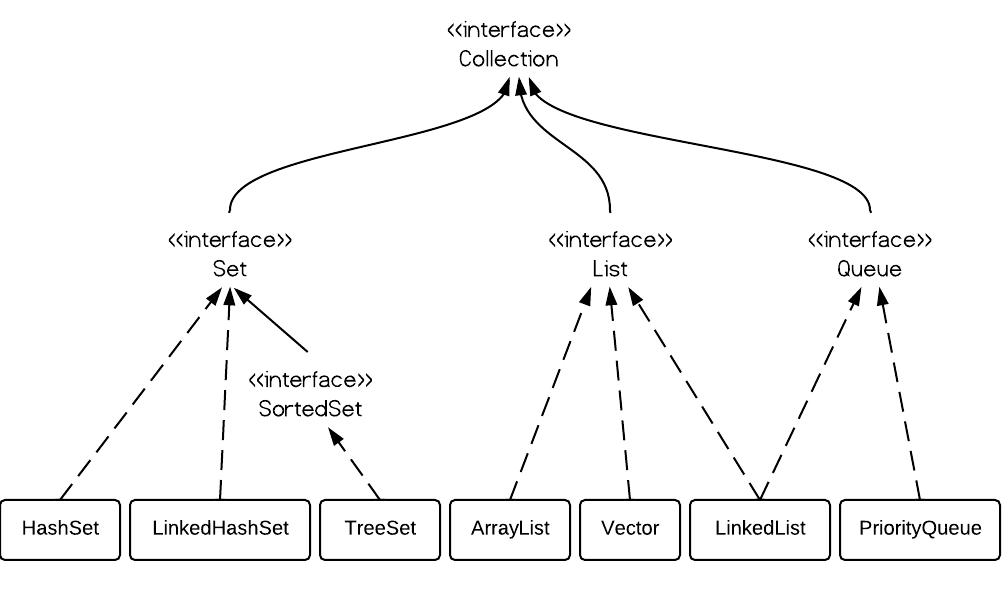
* Both guarantee duplicate-free collection of elements
* It is generally faster to add elements to the HashSet and then convert the collection to a TreeSet for a duplicate-free sorted traversal.
* None of these implementations are synchronized. That is if multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
* LinkedHashSet is in some sense intermediate between HashSet and TreeSet. Implemented as a hash table with a linked list running through it, however,it provides insertion-ordered iteration which is not same as sorted traversal guaranteed by TreeSet.

So a choice of usage depends entirely on your needs but I feel that even if you need an ordered collection then you should still prefer HashSet to create the Set and then convert it into TreeSet.

1. Compare ArrayList, List & LinkedList

**List Overview**

List, as its name indicates, is an ordered sequence of elements. When we talk about List, it is a good idea to compare it with Set which is a set of unique and unordered elements. The following is the class hierarchy diagram of Collection. From the hierarchy diagram you can get a general idea of Java Collections.



**ArrayList vs. LinkedList vs. Vector**

From the hierarchy diagram, they all implement List interface. They are very similar to use. Their main difference is their implementation which causes different performance for different operations.

**ArrayList** is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array.

**LinkedList** is implemented as a double linked list. Its performance on add and remove is better than Arraylist, but worse on get and set methods.

**Vector** is similar with ArrayList, but it is synchronized.

**ArrayList** is a better choice if your program is thread-safe. Vector and ArrayList require more space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.

Note: The default initial capacity of an ArrayList is pretty small. It is a good habit to construct the ArrayList with a higher initial capacity. This can avoid the resizing cost.

1. Is HasTable allows duplicate key name, What will happen if it allowed

NO

1. How do convert HashSet to TreeSet

Set<String> tset = new TreeSet<String>(hashSet);

1. What is HashMap collation and how do avoid it.
2. How do sort the values from Array
3. Difference between Comparator & Comparable

Java provides two interfaces to sort objects using data members of the class:

* Comparable
* Comparator

**Using Comparable Interface**

A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

**Using Comparator**

Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

* Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
* Make an instance of the Comparator class.
* Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

import java.io.\*;

import java.util.\*;

class Movie implements Comparable<Movie>

{

    private double rating;

    private String name;

    private int year;

    // Used to sort movies by year

    public int compareTo(Movie m)

    {

        return this.year - m.year;

    }

    public Movie(String nm, double rt, int yr)

    {

        this.name = nm;

        this.rating = rt;

        this.year = yr;

    }

    public double getRating() { return rating; }

    public String getName()   {  return name; }

    public int getYear()      {  return year;  }

}

class RatingCompare implements Comparator<Movie>

{

    public int compare(Movie m1, Movie m2)

    {

        if (m1.getRating() < m2.getRating()) return -1;

        if (m1.getRating() > m2.getRating()) return 1;

        else return 0;

    }

}

class NameCompare implements Comparator<Movie>

{

    public int compare(Movie m1, Movie m2)

    {

        return m1.getName().compareTo(m2.getName());

    }

}

class Main

{

    public static void main(String[] args)

    {

        ArrayList<Movie> list = new ArrayList<Movie>();

        list.add(new Movie("Force Awakens", 8.3, 2015));

        list.add(new Movie("Star Wars", 8.7, 1977));

        list.add(new Movie("Empire Strikes Back", 8.8, 1980));

        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        // Sort by rating : (1) Create an object of ratingCompare

        //                  (2) Call Collections.sort

        //                  (3) Print Sorted list

        System.out.println("Sorted by rating");

        RatingCompare ratingCompare = new RatingCompare();

        Collections.sort(list, ratingCompare);

        for (Movie movie: list)

            System.out.println(movie.getRating() + " " +

                               movie.getName() + " " +

                               movie.getYear());

        // Call overloaded sort method with RatingCompare

        // (Same three steps as above)

        System.out.println("\nSorted by name");

        NameCompare nameCompare = new NameCompare();

        Collections.sort(list, nameCompare);

        for (Movie movie: list)

            System.out.println(movie.getName() + " " +

                               movie.getRating() + " " +

                               movie.getYear());

        // Uses Comparable to sort by year

        System.out.println("\nSorted by year");

        Collections.sort(list);

        for (Movie movie: list)

            System.out.println(movie.getYear() + " " +

                               movie.getRating() + " " +

                               movie.getName()+" ");

    }

}

1. What are the methods available in Collections.
2. Difference between HashMap & Concurrent HashMap

Concurrent hashmap is not accept key as NULL

Is used to avoid the ConcurrentModificationException while iterating the values.

HashMap is the Class which is under Traditional Collection and ConcurrentHashMap is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying ConcurrentModificationException.Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.

static ConcurrentHashMap<Integer,String> l = new ConcurrentHashMap<Integer,String>();

1. Difference between Arrays.sort & Collections.Sort

**Arrays.sort()**:

Arrays.sort() is a method residing in Arrays class. It is used to sort the Array passed to it. It can be integer array, float array, String array, Array of objects etc.

he time complexity for this method is O(nlogn) as it runs quicksort in background

sort() method is best optimized, so if you use this method instead of writing your own, you'll get best results.

import java.util.Arrays;

public class JavaTest{

public static void main(String[] args){

String[] names = {"apples", "zen", "delhi"};

Arrays.sort(names);

for(String name: names)

System.out.print(name+" ");

}

}

//Output:

//apple delhi zen

Arrays.sort() uses Dual-Pivot Quicksort algorithm for primitive types (int, String, double, ...) and Mergesort for array of objects.

**Collections.sort()**:

Collections.sort() is used to sort an object which extends List interface. ArrayList and LinkedList extend List interface, so we can sort them using Collections.sort.

Collections.sort() has a time complexity of O(nlogn) as it run merge sort in background

import java.util.\*;

public class JavaTest{

public static void main(String[] args){

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

l.add(15);

l.add(2);

l.add(43);

l.add(0);

Collections.sort(l);

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

System.out.print(l.get(i)+" ");

}

}

//Output

//0 2 15 43

1. Is HashTable allows null key --> NO

Java 8

1. List Java 8 features

* forEach() method in Iterable interface
* default and static methods in Interfaces
* Functional Interfaces and Lambda Expressions
* Java Stream API for Bulk Data Operations on Collections
* Java Time API
* Collection API improvements
* Concurrency API improvements
* Java IO improvements
* Miscellaneous Core API improvements

1. What is lambda

Reffer: <https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood>

1. What is the use of streaming

Refer: <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

In functional programming, a monad is a structure that represents computations defined as sequences of steps.

A type with a monad structure defines what it means to chain operations, or nest functions of that type together.

Stream operations are either intermediate or terminal.

Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons

**How streams work**

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream().filter(s -> s.startsWith("c")).map(String::toUpperCase).sorted().forEach(System.out::println);

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example filter, map and sorted are intermediate operations whereas forEach is a terminal operation. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as operation pipeline.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify myList by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

**Different kind of streams**

Streams can be created from various data sources, especially collections. Lists and Sets support new methods stream() and parallelStream() to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads and will be covered in a later section of this tutorial. We focus on sequential streams for now:

Arrays.asList("a1", "a2", "a3").stream().findFirst().ifPresent(System.out::println);

// a1

Calling the method stream() on a list of objects returns a regular object stream. But we don't have to create collections in order to work with streams as we see in the next code sample:

Stream.of("a1", "a2", "a3").findFirst().ifPresent(System.out::println);

// a1

Just use Stream.of() to create a stream from a bunch of object references.

Besides regular object streams Java 8 ships with special kinds of streams for working with the primitive data types int, long and double. As you might have guessed it's IntStream, LongStream and DoubleStream.

IntStreams can replace the regular for-loop utilizing IntStream.range():

IntStream.range(1, 4).forEach(System.out::println);

// 1

// 2

// 3

All those primitive streams work just like regular object streams with the following differences: Primitive streams use specialized lambda expressions, e.g. IntFunction instead of Function or IntPredicate instead of Predicate. And primitive streams support the additional terminal aggregate operations sum() and average():

Arrays.stream(new int[] {1, 2, 3}).map(n -> 2 \* n + 1).average().ifPresent(System.out::println); // 5.0

Sometimes it's useful to transform a regular object stream to a primitive stream or vice versa. For that purpose object streams support the special mapping operations mapToInt(), mapToLong() and mapToDouble:

Stream.of("a1", "a2", "a3").map(s -> s.substring(1)).mapToInt(Integer::parseInt).max().ifPresent(System.out::println); // 3

Primitive streams can be transformed to object streams via mapToObj():

IntStream.range(1, 4).mapToObj(i -> "a" + i).forEach(System.out::println);

// a1

// a2

// a3

Here's a combined example: the stream of doubles is first mapped to an int stream and than mapped to an object stream of strings:

Stream.of(1.0, 2.0, 3.0).mapToInt(Double::intValue).mapToObj(i -> "a" + i).forEach(System.out::println);

// a1

// a2

// a3

**Processing Order**

Now that we've learned how to create and work with different kinds of streams, let's dive deeper into how stream operations are processed under the hood.

An important characteristic of intermediate operations is laziness. Look at this sample where a terminal operation is missing:

Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {System.out.println("filter: " + s);return true;});

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

Let's extend the above example by the terminal operation forEach:

Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {System.out.println("filter: " + s);return true;}).forEach(s -> System.out.println("forEach: " + s));

Executing this code snippet results in the desired output on the console:

filter: d2

forEach: d2

filter: a2

forEach: a2

filter: b1

forEach: b1

filter: b3

forEach: b3

filter: c

forEach: c

The order of the result might be surprising. A naive approach would be to execute the operations horizontally one after another on all elements of the stream. But instead each element moves along the chain vertically. The first string "d2" passes filter then forEach, only then the second string "a2" is processed.

This behavior can reduce the actual number of operations performed on each element, as we see in the next example:

Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {System.out.println("map: " + s);return s.toUpperCase();}).anyMatch(s -> {System.out.println("anyMatch: " + s);return s.startsWith("A");});

// map: d2

// anyMatch: D2

// map: a2

// anyMatch: A2

The operation anyMatch returns true as soon as the predicate applies to the given input element. This is true for the second element passed "A2". Due to the vertical execution of the stream chain, map has only to be executed twice in this case. So instead of mapping all elements of the stream, map will be called as few as possible.

**Why order matters**

The next example consists of two intermediate operations map and filter and the terminal operation forEach. Let's once again inspect how those operations are being executed:

Stream.of("d2", "a2", "b1", "b3", "c").map(s -> { System.out.println("map: " + s); return s.toUpperCase();}).filter(s -> { System.out.println("filter: " + s); return s.startsWith("A");}).forEach(s -> System.out.println("forEach: " + s));

// map: d2

// filter: D2

// map: a2

// filter: A2

// forEach: A2

// map: b1

// filter: B1

// map: b3

// filter: B3

// map: c

// filter: C

As you might have guessed both map and filter are called five times for every string in the underlying collection whereas forEach is only called once.

We can greatly reduce the actual number of executions if we change the order of the operations, moving filter to the beginning of the chain:

Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> { System.out.println("filter: " + s); return s.startsWith("a");}).map(s -> { System.out.println("map: " + s); return s.toUpperCase();}).forEach(s -> System.out.println("forEach: " + s));

// filter: d2

// filter: a2

// map: a2

// forEach: A2

// filter: b1

// filter: b3

// filter: c

Now, map is only called once so the operation pipeline performs much faster for larger numbers of input elements. Keep that in mind when composing complex method chains.

Let's extend the above example by an additional operation, sorted:

Stream.of("d2", "a2", "b1", "b3", "c").sorted((s1, s2) -> { System.out.printf("sort: %s; %s\n", s1, s2); return s1.compareTo(s2);}).filter(s -> { System.out.println("filter: " + s); return s.startsWith("a");}).map(s -> { System.out.println("map: " + s); return s.toUpperCase();}).forEach(s -> System.out.println("forEach: " + s));

Sorting is a special kind of intermediate operation. It's a so called stateful operation since in order to sort a collection of elements you have to maintain state during ordering.

Executing this example results in the following console output:

sort: a2; d2

sort: b1; a2

sort: b1; d2

sort: b1; a2

sort: b3; b1

sort: b3; d2

sort: c; b3

sort: c; d2

filter: a2

map: a2

forEach: A2

filter: b1

filter: b3

filter: c

filter: d2

First, the sort operation is executed on the entire input collection. In other words sorted is executed horizontally. So in this case sorted is called eight times for multiple combinations on every element in the input collection.

Once again we can optimize the performance by reordering the chain:

Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> { System.out.println("filter: " + s); return s.startsWith("a");}).sorted((s1, s2) -> { System.out.printf("sort: %s; %s\n", s1, s2); return s1.compareTo(s2);}).map(s -> { System.out.println("map: " + s); return s.toUpperCase();}).forEach(s -> System.out.println("forEach: " + s));

// filter: d2

// filter: a2

// filter: b1

// filter: b3

// filter: c

// map: a2

// forEach: A2

In this example sorted is never been called because filter reduces the input collection to just one element. So the performance is greatly increased for larger input collections.

**Reusing Streams**

Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

Stream<String> stream =Stream.of("d2", "a2", "b1", "b3", "c") .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true); // ok

stream.noneMatch(s -> true); // exception

Calling noneMatch after anyMatch on the same stream results in the following exception:

java.lang.IllegalStateException: stream has already been operated upon or closedat java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)at com.winterbe.java8.Streams5.test7(Streams5.java:38)at com.winterbe.java8.Streams5.main(Streams5.java:28)

To overcome this limitation we have to to create a new stream chain for every terminal operation we want to execute, e.g. we could create a stream supplier to construct a new stream with all intermediate operations already set up:

Supplier<Stream<String>> streamSupplier =() -> Stream.of("d2", "a2", "b1", "b3", "c") .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true); // ok

streamSupplier.get().noneMatch(s -> true); // ok

Each call to get() constructs a new stream on which we are save to call the desired terminal operation.

**Advanced Operations**

Streams support plenty of different operations. We've already learned about the most important operations like filter or map. I leave it up to you to discover all other available operations (see Stream Javadoc). Instead let's dive deeper into the more complex operations collect, flatMap and reduce.

Most code samples from this section use the following list of persons for demonstration purposes:

class Person

{

String name;

int age;

Person(String name, int age)

{

this.name = name; this.age = age;

}

@Overridepublic String toString()

{

return name;

}

}

List<Person> persons =Arrays.asList( new Person("Max", 18), new Person("Peter", 23), new Person("Pamela", 23), new Person("David", 12));

**Collect**

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map. Collect accepts a Collector which consists of four different operations: a supplier, an accumulator, a combiner and a finisher. This sounds super complicated at first, but the good part is Java 8 supports various built-in collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

Let's start with a very common usecase:

List<Person> filtered =persons .stream() .filter(p -> p.name.startsWith("P")) .collect(Collectors.toList());

System.out.println(filtered); // [Peter, Pamela]

As you can see it's very simple to construct a list from the elements of a stream. Need a set instead of list - just use Collectors.toSet().

The next example groups all persons by age:

Map<Integer, List<Person>> personsByAge = persons.stream().collect(Collectors.groupingBy(p -> p.age));

personsByAge.forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]

// age 23: [Peter, Pamela]

// age 12: [David]

Collectors are extremely versatile. You can also create aggregations on the elements of the stream, e.g. determining the average age of all persons:

Double averageAge = persons.stream().collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge); // 19.0

If you're interested in more comprehensive statistics, the summarizing collectors return a special built-in summary statistics object. So we can simply determine min, max and arithmetic average age of the persons as well as the sum and count.

IntSummaryStatistics ageSummary =persons .stream() .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);

// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}

The next example joins all persons into a single string:

String phrase = persons.stream().filter(p -> p.age >= 18).map(p -> p.name).collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);

// In Germany Max and Peter and Pamela are of legal age.

The join collector accepts a delimiter as well as an optional prefix and suffix.

In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped. Keep in mind that the mapped keys must be unique, otherwise an IllegalStateException is thrown. You can optionally pass a merge function as an additional parameter to bypass the exception:

Map<Integer, String> map = persons.stream().collect(Collectors.toMap( p -> p.age, p -> p.name, (name1, name2) -> name1 + ";" + name2));

System.out.println(map);

// {18=Max, 23=Peter;Pamela, 12=David}

Now that we know some of the most powerful built-in collectors, let's try to build our own special collector. We want to transform all persons of the stream into a single string consisting of all names in upper letters separated by the | pipe character. In order to achieve this we create a new collector via Collector.of(). We have to pass the four ingredients of a collector: a supplier, an accumulator, a combiner and a finisher.

Collector<Person, StringJoiner, String> personNameCollector =Collector.of( () -> new StringJoiner(" | "), // supplier (j, p) -> j.add(p.name.toUpperCase()), // accumulator (j1, j2) -> j1.merge(j2), // combiner StringJoiner::toString); // finisher

String names = persons.stream().collect(personNameCollector);

System.out.println(names); // MAX | PETER | PAMELA | DAVID

Since strings in Java are immutable, we need a helper class like StringJoiner to let the collector construct our string. The supplier initially constructs such a StringJoiner with the appropriate delimiter. The accumulator is used to add each persons upper-cased name to the StringJoiner. The combiner knows how to merge two StringJoiners into one. In the last step the finisher constructs the desired String from the StringJoiner.

**FlatMap**

We've already learned how to transform the objects of a stream into another type of objects by utilizing the map operation. Map is kinda limited because every object can only be mapped to exactly one other object. But what if we want to transform one object into multiple others or none at all? This is where flatMap comes to the rescue.

FlatMap transforms each element of the stream into a stream of other objects. So each object will be transformed into zero, one or multiple other objects backed by streams. The contents of those streams will then be placed into the returned stream of the flatMap operation.

Before we see flatMap in action we need an appropriate type hierarchy:

class Foo

{

String name;

List<Bar> bars = new ArrayList<>();

Foo(String name) { this.name = name;}

}

class Bar {

String name;

Bar(String name) { this.name = name;}

}

Next, we utilize our knowledge about streams to instantiate a couple of objects:

List<Foo> foos = new ArrayList<>();

// create foos

IntStream.range(1, 4).forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars

foos.forEach(f ->IntStream .range(1, 4) .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));

Now we have a list of three foos each consisting of three bars.

FlatMap accepts a function which has to return a stream of objects. So in order to resolve the bar objects of each foo, we just pass the appropriate function:

foos.stream().flatMap(f -> f.bars.stream()).forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1

// Bar2 <- Foo1

// Bar3 <- Foo1

// Bar1 <- Foo2

// Bar2 <- Foo2

// Bar3 <- Foo2

// Bar1 <- Foo3

// Bar2 <- Foo3

// Bar3 <- Foo3

As you can see, we've successfully transformed the stream of three foo objects into a stream of nine bar objects.

Finally, the above code example can be simplified into a single pipeline of stream operations:

IntStream.range(1, 4).mapToObj(i -> new Foo("Foo" + i)).peek(f -> IntStream.range(1, 4) .mapToObj(i -> new Bar("Bar" + i + " <- " f.name)) .forEach(f.bars::add)).flatMap(f -> f.bars.stream()).forEach(b -> System.out.println(b.name));

FlatMap is also available for the Optional class introduced in Java 8. Optionals flatMap operation returns an optional object of another type. So it can be utilized to prevent nasty null checks.

Think of a highly hierarchical structure like this:

class Outer {

Nested nested;

}

class Nested {

Inner inner;

}

class Inner {

String foo;

}

In order to resolve the inner string foo of an outer instance you have to add multiple null checks to prevent possible NullPointerExceptions:

Outer outer = new Outer();

if (outer != null && outer.nested != null && outer.nested.inner != null) {

System.out.println(outer.nested.inner.foo);

}

The same behavior can be obtained by utilizing optionals flatMap operation:

Optional.of(new Outer()).flatMap(o -> Optional.ofNullable(o.nested)).flatMap(n -> Optional.ofNullable(n.inner)).flatMap(i -> Optional.ofNullable(i.foo)).ifPresent(System.out::println);

Each call to flatMap returns an Optional wrapping the desired object if present or null if absent.

**Reduce**

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of reduce methods. The first one reduces a stream of elements to exactly one element of the stream. Let's see how we can use this method to determine the oldest person:

persons.stream().reduce((p1, p2) -> p1.age > p2.age ? p1 : p2).ifPresent(System.out::println); // Pamela

The reduce method accepts a BinaryOperator accumulator function. That's actually a BiFunction where both operands share the same type, in that case Person. BiFunctions are like Function but accept two arguments. The example function compares both persons ages in order to return the person with the maximum age.

The second reduce method accepts both an identity value and a BinaryOperator accumulator. This method can be utilized to construct a new Person with the aggregated names and ages from all other persons in the stream:

Person result =persons .stream() .reduce(new Person("", 0), (p1, p2) -> { p1.age += p2.age; p1.name += p2.name; return p1; });

System.out.format("name=%s; age=%s", result.name, result.age);

// name=MaxPeterPamelaDavid; age=76

The third reduce method accepts three parameters: an identity value, a BiFunction accumulator and a combiner function of type BinaryOperator. Since the identity values type is not restricted to the Person type, we can utilize this reduction to determine the sum of ages from all persons:

Integer ageSum = persons.stream().reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum); // 76

As you can see the result is 76, but what's happening exactly under the hood? Let's extend the above code by some debug output:

Integer ageSum = persons.stream().reduce(0, (sum, p) -> { System.out.format("accumulator: sum=%s; person=%s\n", sum, p); return sum += p.age; }, (sum1, sum2) -> { System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2); return sum1 + sum2; });

// accumulator: sum=0; person=Max

// accumulator: sum=18; person=Peter

// accumulator: sum=41; person=Pamela

// accumulator: sum=64; person=David

As you can see the accumulator function does all the work. It first get called with the initial identity value 0 and the first person Max. In the next three steps sum continually increases by the age of the last steps person up to a total age of 76.

Wait wat? The combiner never gets called? Executing the same stream in parallel will lift the secret:

Integer ageSum = persons.parallelStream().reduce(0, (sum, p) -> { System.out.format("accumulator: sum=%s; person=%s\n", sum, p); return sum += p.age; }, (sum1, sum2) -> { System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2); return sum1 + sum2; });

// accumulator: sum=0; person=Pamela

// accumulator: sum=0; person=David

// accumulator: sum=0; person=Max

// accumulator: sum=0; person=Peter

// combiner: sum1=18; sum2=23

// combiner: sum1=23; sum2=12

// combiner: sum1=41; sum2=35

Executing this stream in parallel results in an entirely different execution behavior. Now the combiner is actually called. Since the accumulator is called in parallel, the combiner is needed to sum up the separate accumulated values.

Let's dive deeper into parallel streams in the next chapter.

**Parallel Streams**

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common ForkJoinPool available via the static ForkJoinPool.commonPool() method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

ForkJoinPool commonPool = ForkJoinPool.commonPool();

System.out.println(commonPool.getParallelism()); // 3

On my machine the common pool is initialized with a parallelism of 3 per default. This value can be decreased or increased by setting the following JVM parameter:

-Djava.util.concurrent.ForkJoinPool.common.parallelism=5

Collections support the method parallelStream() to create a parallel stream of elements. Alternatively you can call the intermediate method parallel() on a given stream to convert a sequential stream to a parallel counterpart.

In order to understate the parallel execution behavior of a parallel stream the next example prints information about the current thread to sout:

Arrays.asList("a1", "a2", "b1", "c2", "c1").parallelStream().filter(s -> { System.out.format("filter: %s [%s]\n", s, Thread.currentThread().getName()); return true;}).map(s -> { System.out.format("map: %s [%s]\n", s, Thread.currentThread().getName()); return s.toUpperCase();}).forEach(s -> System.out.format("forEach: %s [%s]\n", s, Thread.currentThread().getName()));

By investigating the debug output we should get a better understanding which threads are actually used to execute the stream operations:

filter: b1 [main]

filter: a2 [ForkJoinPool.commonPool-worker-1]

map: a2 [ForkJoinPool.commonPool-worker-1]

filter: c2 [ForkJoinPool.commonPool-worker-3]

map: c2 [ForkJoinPool.commonPool-worker-3]

filter: c1 [ForkJoinPool.commonPool-worker-2]

map: c1 [ForkJoinPool.commonPool-worker-2]

forEach: C2 [ForkJoinPool.commonPool-worker-3]

forEach: A2 [ForkJoinPool.commonPool-worker-1]

map: b1 [main]

forEach: B1 [main]

filter: a1 [ForkJoinPool.commonPool-worker-3]

map: a1 [ForkJoinPool.commonPool-worker-3]

forEach: A1 [ForkJoinPool.commonPool-worker-3]

forEach: C1 [ForkJoinPool.commonPool-worker-2]

As you can see the parallel stream utilizes all available threads from the common ForkJoinPool for executing the stream operations. The output may differ in consecutive runs because the behavior which particular thread is actually used is non-deterministic.

Let's extend the example by an additional stream operation, sort:

Arrays.asList("a1", "a2", "b1", "c2", "c1").parallelStream().filter(s -> { System.out.format("filter: %s [%s]\n", s, Thread.currentThread().getName()); return true;}).map(s -> { System.out.format("map: %s [%s]\n", s, Thread.currentThread().getName()); return s.toUpperCase();}).sorted((s1, s2) -> { System.out.format("sort: %s <> %s [%s]\n", s1, s2, Thread.currentThread().getName()); return s1.compareTo(s2);}).forEach(s -> System.out.format("forEach: %s [%s]\n", s, Thread.currentThread().getName()));

The result may look strange at first:

filter: c2 [ForkJoinPool.commonPool-worker-3]

filter: c1 [ForkJoinPool.commonPool-worker-2]

map: c1 [ForkJoinPool.commonPool-worker-2]

filter: a2 [ForkJoinPool.commonPool-worker-1]

map: a2 [ForkJoinPool.commonPool-worker-1]

filter: b1 [main]

map: b1 [main]

filter: a1 [ForkJoinPool.commonPool-worker-2]

map: a1 [ForkJoinPool.commonPool-worker-2]

map: c2 [ForkJoinPool.commonPool-worker-3]

sort: A2 <> A1 [main]

sort: B1 <> A2 [main]

sort: C2 <> B1 [main]

sort: C1 <> C2 [main]

sort: C1 <> B1 [main]

sort: C1 <> C2 [main]

forEach: A1 [ForkJoinPool.commonPool-worker-1]

forEach: C2 [ForkJoinPool.commonPool-worker-3]

forEach: B1 [main]

forEach: A2 [ForkJoinPool.commonPool-worker-2]

forEach: C1 [ForkJoinPool.commonPool-worker-1]

It seems that sort is executed sequentially on the main thread only. Actually, sort on a parallel stream uses the new Java 8 method Arrays.parallelSort() under the hood. As stated in Javadoc this method decides on the length of the array if sorting will be performed sequentially or in parallel:

If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.

Coming back to the reduce example from the last section. We already found out that the combiner function is only called in parallel but not in sequential streams. Let's see which threads are actually involved:

List<Person> persons = Arrays.asList(new Person("Max", 18),new Person("Peter", 23),new Person("Pamela", 23),new Person("David", 12));

persons.parallelStream().reduce(0, (sum, p) -> { System.out.format("accumulator: sum=%s; person=%s [%s]\n", sum, p, Thread.currentThread().getName()); return sum += p.age; }, (sum1, sum2) -> { System.out.format("combiner: sum1=%s; sum2=%s [%s]\n", sum1, sum2, Thread.currentThread().getName()); return sum1 + sum2; });

The console output reveals that both the accumulator and the combiner functions are executed in parallel on all available threads:

accumulator: sum=0; person=Pamela; [main]

accumulator: sum=0; person=Max; [ForkJoinPool.commonPool-worker-3]

accumulator: sum=0; person=David; [ForkJoinPool.commonPool-worker-2]

accumulator: sum=0; person=Peter; [ForkJoinPool.commonPool-worker-1]

combiner: sum1=18; sum2=23; [ForkJoinPool.commonPool-worker-1]

combiner: sum1=23; sum2=12; [ForkJoinPool.commonPool-worker-2]

combiner: sum1=41; sum2=35; [ForkJoinPool.commonPool-worker-2]

In summary, it can be stated that parallel streams can bring be a nice performance boost to streams with a large amount of input elements. But keep in mind that some parallel stream operations like reduce and collect need additional computations (combine operations) which isn't needed when executed sequentially.

Furthermore we've learned that all parallel stream operations share the same JVM-wide common ForkJoinPool. So you probably want to avoid implementing slow blocking stream operations since that could potentially slow down other parts of your application which rely heavily on parallel streams.

1. Is parallel stream is creating the multithread

Yes

1. What is use of FunctionalInterface

@FunctionalInterface annotation is useful for compilation time checking of your code. You cannot have more than one method besides static, default and abstract methods that override methods in Object in your @FunctionalInterface or any other interface used as a functional interface.

But you can use lambdas without this annotation as well as you can override methods without @Override annotation.

1. What is difference between Lambda Expression and Anonymous class?

The key difference between Anonymous class and Lambda expression is the usage of 'this' keyword. In the anonymous classes, ‘this’ keyword resolves to anonymous class itself, whereas for lambda expression ‘this’ keyword resolves to enclosing class where lambda expression is written.

Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into private method of the class. It uses invokedynamic instruction that was added in Java 7 to bind this method dynamically.

1. What is the difference between Closure and Lambda in Java 8?

Lambdas are a language construct (anonymous functions), closures are an implementation technique to implement first-class functions (whether anonymous or not).

A **lambda** is just an anonymous function - a function defined with no name. In some languages, they are equivalent to named functions. In fact, the function definition is re-written as binding a lambda to a variable internally. In other languages, like Python, there are some (rather needless) distinctions between them, but they behave the same way otherwise.

**Closures** are nothing but stateful functions! A few definitions from the web include:

* A closure is a combination of a function bundled together (enclosed) with references to its surrounding state
* A closure gives you access to an outer function’s scope from an inner function

**interface** IProcess

{

**void** process(**int** i);

}

**public** **static** **void** main(String[] args)

{

**int** a=10;

**int** b = 30; //this is closure

//JDK 7

*doProcess*(a, **new** IProcess() {

@Override

**public** **void** process(**int** i) {

System.***out***.println(i +b);

}

});

//JDK 8

*doProcess*(a, i -> System.***out***.println(i+b));

}

**private** **static** **void** doProcess(**int** i, IProcess process)

{

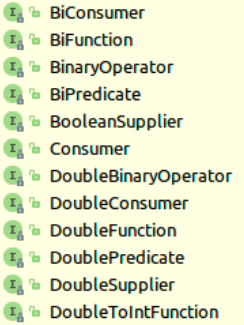
process.process(i);

}

1. Does Java 8 Lambda supports recursive call?

In general, Lambda implementations are mostly anonymous functions. In recursion, a method calls itself. Since anonymous function doesnot have a name, it cannot be called by itself. That means an anonymous Lambda can not be called by itself. But if we have a Lambda function declaration as a member variable or class variable, Java 8 supports recursion with Lambda functions. Java 8 does not support Lambda function declaration with local variable.

1. Can u give some FunctionalInterface name & the uses



**BiConsumer**

import java.util.function.BiConsumer;

public class UsingBiConsumer {

public static void main(String[] args) {

BiConsumer printer = (s1, s2) -> System.out.println(s1 + " " + s2);

printer.accept("Hello!","Functional Interface API");

}

}

This code is as crisp as one could write using the existing functional interfaces.

**BiFunction** is another interface similar to **BiConsumer** with the only difference that it returns.

**Consumer**, as is apparent, takes only one parameter and performs some operations without returning anything.

**Function** is similar to Consumer but returns a value of some generic type.

Another important functional interface is **Supplier**. As the name suggests, it accepts no parameter but returns a value of some type.

Below is an example of its implementation:

import java.util.function.Supplier;

public class SupplierImplementation {

public static void main(String[] args) {

Supplier ageSupplier = ()-> 20;

System.out.println("Minimum age for applying for this job is:"+ageSupplier.get());

}

}

**Predicate** is an interface that takes an input of some type and always outputs either true or false. Hence, the name.

Remember to be DRY (Don’t Repeat Yourself) and check the java.util.function package for any existing functional interface. Create a new one only when it is not available.

Hope it helps. My recommendation is that you go through java.util.function package once to see all available functional interfaces.

1. Why java provides/accepts to create the default/static methods in interface

ava 8 introduced default and static methods in interfaces. This feature enables us to add new functionality in the interfaces without breaking the existing contract of the implementing classes.

**How to Define Default and Static Methods?**

Default methods have default, and static methods have thestatic keyword in the method signature.

public interface Parser {

List<String> parse(File file);

// default implementation valid for all the file parser viz. JsonFileParser, CsvFileParser, TextFileParser

default boolean canParse(File file) {

return Objects.nonNull(file) && file.getName().endsWith(getFileType().getExtension());

}

FileType getFileType();

static void someStaticHelperMethod() {

//helper method implementation

}

}

**What Is the Need for Default Methods?**

Default methods in interfaces help us to introduce new functionality without breaking the contract of the implementing classes.

Suppose we have an Expression interface that has ConstantExpression, BinaryExpression, DivisionExpression etc. as existing implementations. Now we get a requirement to add new functionality.

* Return the signum of the evaluated result.
* Return signum after evaluating the expression.

This can be done with default and static methods without breaking any functionality as follows.

public interface Expression {

double evaluate();

default double signum() {

return signum(evaluate());

}

static double signum(double value) {

return Math.signum(value);

}

}

**Default Methods and Multiple Inheritance Ambiguity Problems**

Java supports multiple inheritance of interfaces. Consider having two interfaces, InterfaceA and InterfaceB , having default methods with the same signature. Your class ConcreteC is implementing both the interfaces.

interface InterfaceA {

void performA();

default boolean canPerform() {

// return true if I can perform the action

}

}

interface InterfaceB {

void performB();

default boolean canPerform() {

//return true if I can perform the action

}

}

class ConcreteC implements InterfaceA, InterfaceB {

}

The above code will fail to compile with "error: unrelated defaults for canPerform()from InterfaceA and InterfaceB."

To overcome this problem, you need to override the default method.

class ConcreteC implements InterfaceA, InterfaceB {

override

public boolean canPerform() {

}

}

But say you don't want to provide the implementation of the overridden default method but instead want to reuse the existing one. That is also possible with the following syntax.

class ConcreteC implements InterfaceA, InterfaceB {

override

public boolean canPerform() {

return InterfaceA.super.canPerform();

}

}

A Few Important Points for Default Methods

* You can inherit the default method.
* You can redeclare the default method essentially making it abstract.
* You can redefine the default method (equivalent to overriding).

1. I have a common method in two interface and these are implemented in the class how can i access the method from derived class.

Refer the above question last samples

1. What is the use of optional class
2. Remove duplicate values from list

**class** StudentT

{

**int** id;

String name;

String dept;

**public** StudentT(**int** id, String name, String dept) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.dept = dept;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getDept() {

**return** dept;

}

**public** **void** setDept(String dept) {

**this**.dept = dept;

}

}

**public** **static** **void** main(String[] args)

{

List<StudentT> list = Arrays.*asList*(

**new** StudentT(100, "Vadvivelan", "CSE"),

**new** StudentT(102, "Muthuramalinga", "CSE"),

**new** StudentT(100, "Vadvivelan", "CSE"),

**new** StudentT(103, "Vadvivelan", "CSE"),

**new** StudentT(104, "Senthil", "CSE"),

**new** StudentT(105, "Vadvivelan", "CSE"),

**new** StudentT(104, "Karthik", "CSE"));

System.***out***.println("--------------Remove Duplicated baased on ID 1 -----------");

Set<StudentT> set=

list.stream().collect(

Collectors.*toCollection*(

() -> **new** TreeSet<StudentT>(Comparator.*comparing*(StudentT::getId))

));

System.***out***.println("-------Remove Duplicated baased on ID ------");

list.stream().distinct().collect(Collectors.*toList*())

.forEach(x -> System.***out***.println("ID : " + x.getId() + " Name : " + x.getName() + " Dept : " + x.getDept()));

1. How do filter the values from list
2. Convert List to Set

List<String> list = **new** ArrayList<>();

list.add("Vadivelan");

list.add("Muthuramalingam");

list.add("Vadivelan");

System.***out***.print("Print values what is available in the list : ");

System.***out***.print(list);

Set<String> set = **new** HashSet<>();

set.add("Vadivelan 123");

set.add("Vadivelan 123");

set.add("Vadivelan 123");

set.add("Vadivelan 123");

set = list.stream().distinct().collect(Collectors.*toSet*());

System.***out***.println(set);

System.***out***.print("Print values and remove duplicate : ");

list.stream().distinct().forEach(p -> System.***out***.println(p));

System.***out***.print("Print values and remove duplicate Another Approach : ");

list.stream().distinct().collect(Collectors.*toList*());

System.***out***.println(list);

list.add("Vadivelan");

System.***out***.print("Print values and remove duplicate Another Approach by using map: ");

list = list.stream().map(x -> x).distinct().collect(Collectors.*toList*());

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

List<Integer> listWithoutDuplicates = listWithDuplicates.stream().distinct()

.collect(Collectors.*toList*());

1. Convert HashMap to List

List<Integer> result = map.keySet().stream().collect(Collectors.*toList*());

List<String> result2 = map.values().stream().collect(Collectors.*toList*());

List<String> result3 = map.values().stream().filter(

x -> !"banana".equalsIgnoreCase(x)).collect(Collectors.toList());

//Convert Map to List of Object

List<Person> list = map.entrySet().stream().sorted(

Comparator.comparing(e -> e.getKey()))

.map(e -> new Person(e.getKey(), e.getValue()))

.collect(Collectors.toList());

1. Convert List to HashMap

//Convert List to Map and eliminate duplicate values

List<Store> list = **new** ArrayList<>();

list.add(**new** Store(1, "Vadivelan"));

list.add(**new** Store(2, "Vadivelan"));

list.add(**new** Store(2, "Muthu"));

Map<Integer, String> result =

list.stream()

.collect(

Collectors.*toMap*(Store::getNo,Store::getName,(oldValue,newValue)-> newValue)

);

System.***out***.println("Result 1 : " + result);

List<Person> persons = Arrays.*asList*(**new** Person("Max", 18),

**new** Person("Peter", 23),

**new** Person("Pamela", 23),

**new** Person("David", 12));

Map<Integer, String> map = persons

.stream()

.collect(Collectors.*toMap*(

p -> p.age,

p -> p.name,

(name1, name2) -> name1 + ";" + name2));

System.***out***.println(map);

1. How convert multiple HashMap to List
2. How do merger the more than one HashMap to single HashMap

new\_hash\_map.forEach((k,v) -> hash\_map.merge(k, v, (oldVlaue,newValue) -> newValue));

1. How do achieve multicore programming in java 8

Refer Stream and ForkJoin

1. Differences between Collection API and Stream API?
2. What is Spliterator in Java SE 8?
3. Differences between Iterator and Spliterator in Java SE 8?
4. What are the major advantages of Internal Iteration over External Iteration

**External Iterators Definition(or Active Iterators)** With external iterators responsibility of iterating over the elements, and making sure that this iteration takes into account the total number of records, whether more records exist to be iterated and so on lies with the programmer.

**A brief look at the evolution of external iterators in java**

Lets look into some external iterators which we have been using as java language evolved over the years.

Starting with Enumerations, iterations then moved on to Iterators(remember iterator(), next() or hasNext() methods for iterators). Then came Java 5 and along with came the enhanced for-loop which made use of generics to make iteration a lot easier. Lets see an example of enhanced for-loop introduced in Java 5 which uses the Iterable interface (you might already be familiar with this one)

**Example 1:**

import java.util.\*;

public class ExternalIterator {

public static void main(String args[]){

List<String> namesList=Arrays.asList("Tom", "Dick", "Harry");

for(String name:namesList){

System.out.println(name);

}

}

}

**Example 2:**

int count = 0;

Iterator<SomeStaff> iterator = allTheStaffs.iterator();

while(iterator.hasNext()) {

SomeStaff staff = iterator.next();

if(staff.getSalary() > 25) {

count++;

}

}

However, though we are explicitly not invoking hasNext() or next() methods while iterating over the list above, still the underlying code which makes this iteration work uses these methods. This implies that the complexity behind these operations is hidden from the programmer but it still exists. And it still is an active iterator.

**Internal Iterators(or Passive Iterators)** Internal Iterators manage the iterations in the background. This leaves the programmer to just declaratively code what is meant to be done with the elements of the Collection, rather than managing the iteration and making sure that all the elements are processed one-by-one.

Lets see how simple it is to say print all elements in an ArrayList in Java 8 using an example of internal iterator based forEach loop

import java.util.\*;

public class InternalIterator {

public static void main(String args[]){

List<String> namesList=Arrays.asList("Tom", "Dick", "Harry");

namesList.forEach(name -> System.out.println(name));//Internal Iteration

}

}

**Advantage of internal iterators over external iterators**

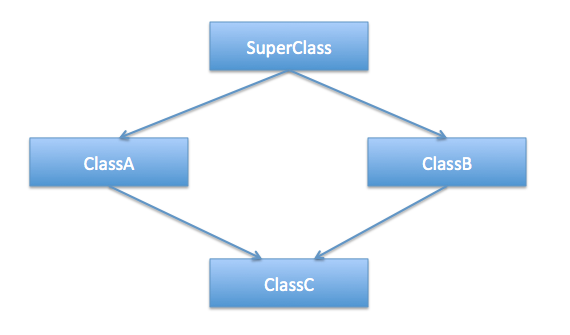
* Improved code readability as its declarative in nature
* Concise code as multiple lines of code for external iterators is reduced to just one or two lines of code in case of internal iterators
* Simplified implementation/less defects as code written by programmer is very less, chances of bugs creeping into the iteration logic are not there.

**Point in favor of external iterators**

If you want more control over the iteration, and want to perform some check or operation for a particular index then external iterators are preferred over internal ones.

1. What is Diamond Problem in Inheritance? How Java 8 Solves this problem

Refer the below flow chart to understand the diamond problem.



SuperClass is a abstract class.

ClassA is a concrete class which is inherited from SuperClass

ClassB is a concrete class which is inherited from SuperClass

ClassB is a concrete class which is extends from ClassA & ClassB (Its not accept in java).

**How are above problems handled for Default Methods and Interfaces ?**

Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

// A simple Java program to demonstrate multiple

// inheritance through default methods.

interface PI1

{

    // default method

    default void show()

    {

        System.out.println("Default PI1");

    }

}

interface PI2

{

    // Default method

    default void show()

    {

        System.out.println("Default PI2");

    }

}

// Implementation class code

class TestClass implements PI1, PI2

{

    // Overriding default show method

    public void show()

    {

        // use super keyword to call the show

        // method of PI1 interface

        PI1.super.show();

        // use super keyword to call the show

        // method of PI2 interface

        PI2.super.show();

    }

    public static void main(String args[])

    {

        TestClass d = new TestClass();

        d.show();

    }

}

Spring & Spring Boot

1. What is Spring

Basically Spring is a framework for dependency-injection which is a pattern that allows to build very decoupled systems.

Spring is a good framework for web development. Spring MVC is one of the many parts of Spring, and is a web framework making use of the general features of Spring, like dependency injection. It is a pretty generic framework in that it is very configurable: you can use different DB layers (Hibernate, iBatis, plain JDBC), different view layers (JSP, Velocity, Freemarker...)

1. What is Dependency Injection

Any application is composed of many objects that collaborate with each other to perform some useful stuff. Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.

For example, consider a Car object. A Car depends on Wheels, Engine, Fuel, Battery, etc to run. Traditionally we define the brand of such dependent objects along with the definition of the Car object.

class Car{

private Wheel wh= new NepaliRubberWheel();

private Battery bt= new ExcideBattery();

//rest

}

Here, the Car object is responsible for creating the dependent objects.

What if we want to change the type of its dependent object - say Wheel - after the initial NepaliRubberWheel() punctures? We need to recreate the Car object with its new dependency say ChineseRubberWheel(), but only the Car manufacturer can do that.

Then what the Dependency Injection do us for ...

When using Dependency Injection, objects are given their dependencies at run time rather than compile time (car manufacturing time). So that we can now change the Wheel whenever we want. Here, the Dependency (Wheel) can be injected into Car at run time.

Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and Dependency Injection is merely one concrete example of Inversion of Control.

This concept says that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file. A container is then responsible for hooking it all up.

1. What is the use of IOC

Inversion of Control (IoC) is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* To instantiate the application class.
* To configure the object.
* To assemble the dependencies between the objects.

There are two types of IoC containers. They are:

* BeanFactory
* ApplicationContext

1. What is Bean

Bean is an object, which is created, managed and destroyed in Spring Container. We can inject an object into the Spring Container through the metadata(either xml or annotation), which is called inversion of control.

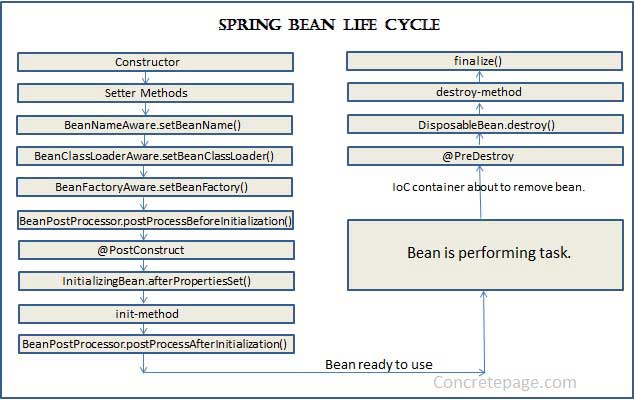
1. Differences between BeanFactory and the ApplicationContext in Spring framework.

**Spring BeanFactory Container**: This is the simplest container providing the basic support for DI (dependency injection) and is defined by the org.springframework.beans.factory.BeanFactory interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

**Spring ApplicationContext Container**: This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the org.springframework.context.ApplicationContext interface.

The ApplicationContext container includes all functionality of the BeanFactorycontainer, so it is generally recommended over BeanFactory. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

1. Explain Bean Life Cycle



A bean life cycle includes the following steps.   
**1.** Within IoC container, a spring bean is created using class **constructor**.   
**2.** Now the dependency injection is performed using setter method.   
**3.** Once the dependency injection is completed, BeanNameAware.setBeanName() is called. It sets the name of bean in the bean factory that created this bean.   
**4.** Now < code>BeanClassLoaderAware.setBeanClassLoader() is called that supplies the bean class loader to a bean instance.   
**5.** Now < code>BeanFactoryAware.setBeanFactory() is called that provides the owning factory to a bean instance.   
**6.** Now the IoC container calls BeanPostProcessor.postProcessBeforeInitialization on the bean. Using this method a wrapper can be applied on original bean.   
**7.** Now the method annotated with @PostConstruct is called.   
**8.** After @PostConstruct, the method InitializingBean.afterPropertiesSet() is called.   
**9.** Now the method specified by init-method attribute of bean in XML configuration is called.   
**10.** And then BeanPostProcessor.postProcessAfterInitialization() is called. It can also be used to apply wrapper on original bean.   
**11.** Now the bean instance is ready to be used. Perform the task using the bean.   
**12.** Now when the ApplicationContext shuts down such as by using registerShutdownHook() then the method annotated with @PreDestroy is called.   
**13.** After that DisposableBean.destroy() method is called on the bean.   
**14.** Now the method specified by destroy-method attribute of bean in XML configuration is called.   
**15.**Before garbage collection, finalize() method of Object is called.

1. Benefits of DI

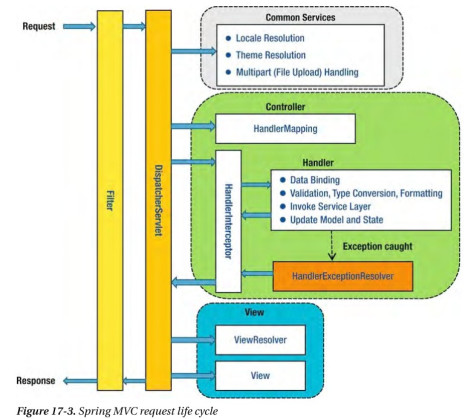
* Key benefit is loose coupling between dependent objects.If an object operates on their dependencies by their interface not by implementation then compile time dependency can be swapped out with Dependency Injection.
* Very useful when we have objects (Wheel) whose implementations change often ( if the type punctures often we can change it easily) ?
* Very useful for large projects where there is issue of maintainability, simplicity and many others ty

1. What are the different types of dependency injections in spring?

Spring supports 2 types of dependency injection, they are:

* **Constructor-based dependency injection**: It is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
* **Setter-based dependency injection**: It is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

1. Spring MVC LifeCycle



The main components and their purposes are as follows:

* **Filter**: The filter applies to every request. Several commonly used filters and their purposes are described in the next section.
* **Dispatcher servlet**: The servlet analyzes the requests and dispatches them to the appropriate controller for processing.
* **Common services**: The common services will apply to every request to provide supports including i18n, theme, file upload, and so on. Their configuration is defined in the DispatcherServlet’s WebApplicationContext.
* **Handler mapping**: This maps the request to the handler (a method within a Spring MVC controller class). Since Spring 2.5, in most situations the configuration is not required because Spring MVC will automatically register the org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping class that maps handlers based on HTTP paths expressed through the @RequestMapping annotation at the type or method level within controller classes.
* **Handler interceptor**: In Spring MVC, you can register interceptors for the handlers for implementing common checking or logic. For example, a handler interceptor can check and ensure that only the handlers can be invoked during office hours.
* **Handler exception resolver**: In Spring MVC, the HandlerExceptionResolver interface (under the packageorg.springframework.web.servlet) is designed to deal with unexpected exceptions thrown during request processing by handlers.

By default, the DispatcherServlet registers the DefaultHandlerExceptionResolver class (under the packageorg.springframework.web.servlet.mvc.support). This resolver handles certain standard Spring MVC exceptions by setting a specific response status code. You can also implement your own exception handler by annotating a controller method with the @ExceptionHandler annotation and passing in the exception type as the attribute.

* **View Resolver**: Spring MVC’s ViewResolver interface (under the package org.springframework.web.servlet) supports view resolution based on a logical name returned by the controller. There are many implementation classes to support various view resolving mechanisms. For example, the UrlBasedViewResolver class supports direct resolution of logical names to URLs. The ContentNegotiatingViewResolver class supports dynamic resolving of views depending on the media type supported by the client (such as XML, PDF, JSON, and so on). There also exists a number of implementations to integrate with different view technologies, such as FreeMarker (FreeMarkerViewResolver), Velocity (VelocityViewResolver), and JasperReports (JasperReportsViewResolver).

These descriptions cover only a few commonly used handlers and resolvers. For a full description, please refer to the Spring Framework reference documentation and its Javadoc.

1. What is the use of filter
2. What is the use of DispatcherServlet

DispatcherServlet as the Heart of Spring MVC

What we really want to do as developers of a web application is to abstract away the following tedious and boilerplate tasks and focus on useful business logic:

* mapping an HTTP request to a certain processing method
* parsing of HTTP request data and headers into data transfer objects (DTOs) or domain objects
* model-view-controller interaction
* generation of responses from DTOs, domain objects, etc.

The Spring DispatcherServlet provides exactly that. It is the heart of the Spring Web MVC framework; this core component receives all requests to your application.

As you’ll see, DispatcherServlet is very extensible. For example, it allows you to plug in different existing or new adapters for a lot of tasks:

* map a request to a class or method that should handle it (implementations of the HandlerMappinginterface)
* handle a request using a specific pattern, like a regular servlet, a more complex MVC workflow, or just a method in a POJO bean (implementations of the HandlerAdapter interface)
* resolve views by name, allowing you to use different templating engines, XML, XSLT or any other view technology (implementations of the ViewResolver interface)
* parse multipart requests by using the default Apache Commons file uploading implementation or writing your own MultipartResolver
* resolve locale with any LocaleResolver implementation, including cookie, session, Accept HTTP header, or any other way of determining the locale expected by the user

1. What are the limitations and disadvantages of spring autowiring?

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions. Consider the limitations and disadvantages of autowiring:

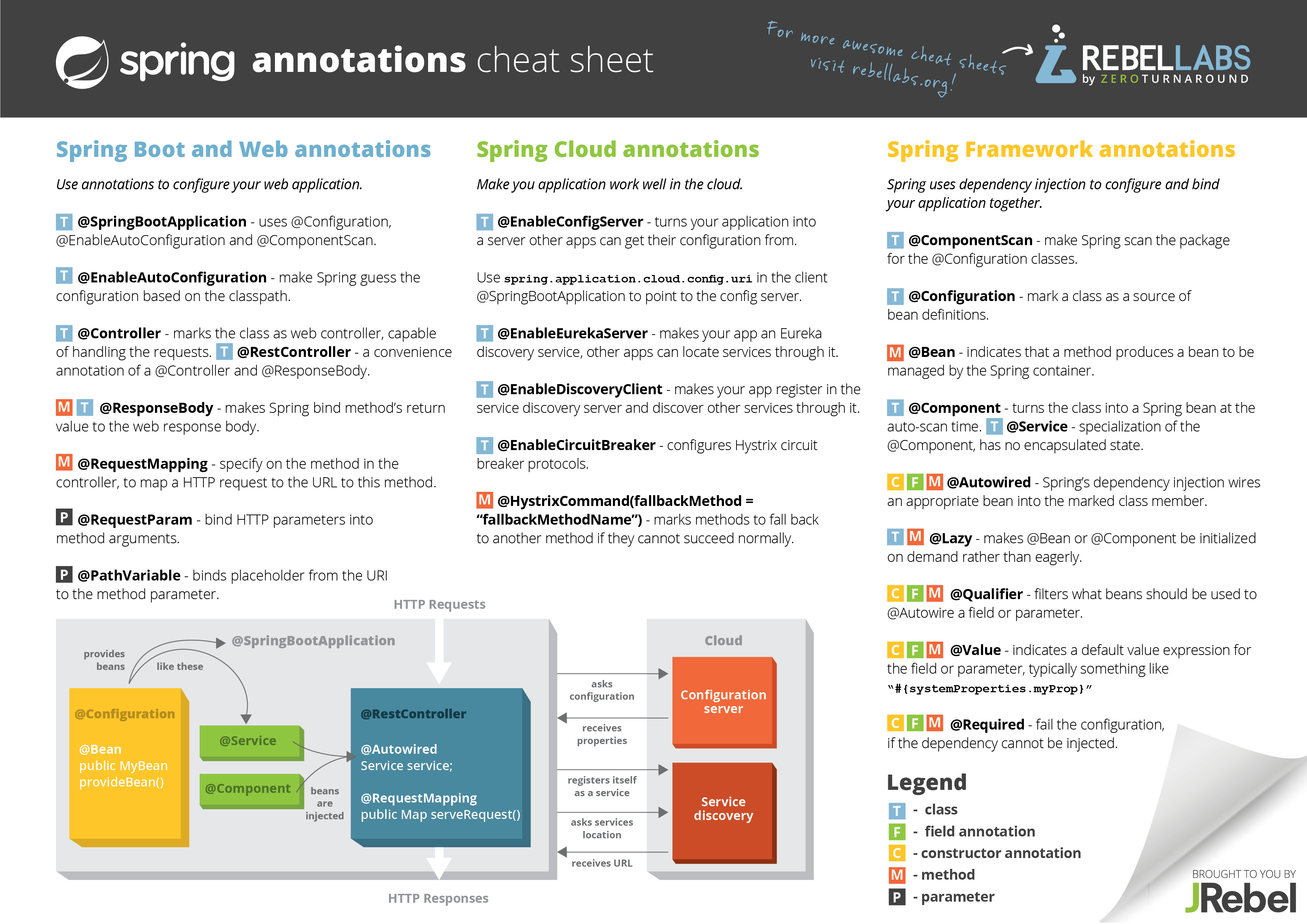
Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.

Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.

Wiring information may not be available to tools that may generate documentation from a Spring container.

Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

1. Types of annotation



1. What is the scope of default bean

SingletoN

1. Types of scope in Spring bean

* **singleton** only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
* **prototype** A new instance will be created every time the bean is requested from the spring container.
* **request** This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
* **session** A new bean will be created for each HTTP session by the container.
* **global-session** This is used to create global session beans for Portlet applications.

1. I have a SingleToN object class internally i have Prototype scope Class. Is Second class will work as prototype scope or it will work as a SingleToN pattern
2. Compare @Component & @Service

* @Component: generic stereotype for any Spring-managed component
* @Repository : stereotype for persistence layer
* @Service : stereotype for service layer
* @Controller : stereotype for presentation layer (spring-mvc)

1. Types of HTTP methods in Rest Service

| **HTTP Verb** | **CRUD** | **Entire Collection (e.g. /customers)** | **Specific Item (e.g. /customers/{id})** |
| --- | --- | --- | --- |
| POST | Create | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| GET | Read | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| PATCH | Update/Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

1. What is the difference between @Requestparam & @PathVariable
2. How do chose the MediaType
3. Difference between @Controller & @RestController
4. What is use of @ControllerAdvice
5. How do handle the Exception in Spring MVC/REST
6. How do create a Bean in XML
7. How do achieve role based authentication in Spring
8. How do restrict the URI/Controller access in Spring
9. What is HTTP basic authentication?

In the context of a HTTP transaction, basic access authentication is a method for an HTTP user agent to provide a user name and password when making a request.

HTTP Basic authentication implementation is the simplest technique for enforcing access controls to web resources because it doesn't require cookies, session identifier and login pages. Rather, HTTP Basic authentication uses static, standard HTTP headers which means that no handshakes have to be done in anticipation.

When the user agent wants to send the server authentication credentials it may use the Authorization header. The Authorization header is constructed as follows:

1) Username and password are combined into a string "username:password"

2) The resulting string is then encoded using Base64 encoding

3) The authorization method and a space i.e. "Basic " is then put before the encoded string.

1. What is the difference between HTTP methods GET and POST?

HTTP works as a request-response protocol between a client and server. A web browser may be the client, and an application on a computer that hosts a web site may be the server. Two commonly used HTTP methods to make a request to the server are GET and POST.

When you use GET method, the data will be sent to the server as a query parameters. These are appended to the URL as a key value pair. In the below URL, you can see how data is passed to the server as key value pair. These values will be visible at the address bar. URL character length is limited, so you can not use it if you are sending large data. GET is recommended to use for querying information from server, kind of search operations. GET requests should never be used when dealing with sensitive data.

http://java2novice.com/history?name=madhu&language=java

POST method sends data as part of HTTP message body, data sent to the server, will not be visible to the user. POST requests cannot be cached. It does not have any character length restrictions. POST is recommended to submits data to be processed to a specified resource.

1. What is MVC pattern?

MVC is a design pattern called Model-View-Controller. It decouples data access logic from business logic.

**Model**:

The Model contains the core of the application's functionality. It encapsulates the state of the application. Sometimes the only functionality it contains is state. It knows nothing about the view or controller.

**View**:

The view provides the presentation of the model. It is the look and feel of the application. The view can access the model getters, but it has no knowledge of the setters. In addition, it knows nothing about the controller. The view should be notified when changes to the model occur.

**Controller**:

The controller reacts to the user input. It creates and sets the model and helps to identify which view should be part of response.

1. What is the difference between Servlet and Filter?

A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

1) Query the request and act accordingly.

2) Block the request-and-response pair from passing any further.

3) Modify the request headers and data. You do this by providing a customized version of the request.

4) Modify the response headers and data. You do this by providing a customized version of the response.

5) Interact with external resources.

Servlet is used for performing the action which needs to be taken for particular request like user login, get the response based on user role, interacts with database for getting the data, business logic execution, etc.

1. What is the difference between application server and web server?

Web Server is designed to serve HTTP Content. Application Server can also serve HTTP Content but is not limited to just HTTP. It can be provided other protocol support such as RMI/RPC

Web Server is mostly designed to serve static content, though most Web Servers have plugins to support scripting languages like Perl, PHP, ASP, JSP etc. through which these servers can generate dynamic HTTP content.

Most of the application servers have Web Server as integral part of them, that means App Server can do whatever Web Server is capable of. Additionally Application Server have components and features to support Application level services such as Connection Pooling, Object Pooling, Transaction Support, Messaging services etc.

As web servers are well suited for static content and app servers for dynamic content, most of the production environments have web server acting as reverse proxy to app server. That means while service a page request, static contents such as images/Static html is served by web server that interprets the request. Using some kind of filtering technique (mostly extension of requested resource) web server identifies dynamic content request and transparently forwards to app server

Example of such configuration is Apache HTTP Server and BEA WebLogic Server. Apache HTTP Server is Web Server and BEA WebLogic is Application Server.

1. What is the difference between JPA and Hibernate?

JPA is just a specification which needs concrete implementation. The default implementation provided by oracle is "Eclipselink" now. Toplink is donated by Oracle to Eclipse foundation to merge with eclipselink.

Using Eclipselink, one can be sure that the code is portable to any implementation if need arises. Hibernate is also a full JPA implementation + MORE. Hibernate is super set of JPA with some extra Hibernate specific functionality. So application developed in Hibernate may not be compatible when switched to other implementation. Still hibernate is choice of majority of developers as JPA implementation and widely used.

Another JPA implementation is OpenJPA, which is an extension of Kodo implementation.

1. What is difference between the Value Object and JDO?

Java Data Objects (JDO) is really a technology of persistence used to keep objects of Java in databases which provides the advantage to its developers by manipulating all details at applications level and helps developers to concentrate on coding that is not database specific.

But, the Value Objects represents an abstracted design blueprint, which provides a generic holder of data known as Data transfer Object which can hold data so that it can be transferred between the customer and databases. JDO provides the advantage of persisting data, while the Value Object is in charge of keeping data provisionally during the period of data transfer only. In other words, the Value Object does not provide persistence.

1. Add quest

Spring Boot

1. REST based Services/Architecture VC RESTFUL Services/Architecture

REST (REpresentational State Transfer) is basically an architectural style of development having some principles...

* It should be stateless
* It should access all the resources from the server using only URI
* It does not have inbuilt encryption
* It does not have session
* It uses one and only one protocol that is HTTP
* For performing CRUD operations, it should use HTTP verbs such as get, post, put and delete
* It should return the result only in the form of JSON or XML, atom, OData etc. (lightweight data )

REST based services follow some of the above principles and not all

RESTFUL services means it follows all the above principles.

* MVC only supports the following from the REST API
* We can access the resource using URI
* It supports the HTTP verb to access the resource from server
* It can return the results in the form of JSON, XML, that is the HTTPResponse.

However, at the same time in MVC

* We can use the session
* We can make it stateful
* We can return video or image from the controller action method which basically violates the REST principles

That is why MVC is REST-Based whereas WEB API supports all the above principles and is RESTFul.

1. What is Spring Boot

Spring Boot as a tool which can do these initial tasks for us automatically.Spring Boot works on an opinionated view of the Spring platform being used by us and ensures that team can quickly start working on solving the actual business problem rather than spending time on the initial configurations and setup.

Spring Boot provides the following feature out of the box

* It simplifies Spring dependencies by taking the opinionated view ( we will discuss it in more details).
* Spring Boot provides a preconfigured set of technologies/framework to reduces error-prone configuration so we as a developer focused on building our business logic rather than thinking of project setup.
* You really don’t need those big XML configurations for your project.
* Embed Tomcat, Jetty or Undertow directly.
* Provide opinionated Maven POM to simplify your configuration

**Better Dependency Management**: Just check configuration closely and you won’t be finding any entry for all those Spring dependencies (like web MVC, core, AOP, ORM, Validation API etc.), you might have noticed similar entries spring-boot-starter-\*, this is Spring Boot dependency management process. We have added spring-boot-starter-web to our pom.xml and Spring Boot will pull all required dependencies for Spring MVC application (no more manual configurations).

**Auto Configurations**: Auto Configuration is another interesting feature of Spring Boot this is why Spring Boot team say’s that it has opinions.These are some of work Spring Boot will do for you

* It will add all dependencies as highlighted in point
* Auto Configurations indicates that Spring Boot has some reasonable defaults i.e based on the configurations Spring Boot will guess the type of application and will supply default implementations required to run your application in case we have not defined those in our application. in case you define these, Spring Boot will ensure that these defaults will be taken out of the context and let your custom configurations will take charge of application.
* To give a more clear picture, let’s say you have defined dependency for JPA and have not defined any database configurations, Spring Boot will automatically create required configurations for us.

1. What are the advantages of Spring Boot?

* It simplifies Spring dependencies by taking the opinionated view.
* Spring Boot provides a pre-configured set of technologies/framework to reduces error-prone configuration so we as a developer focused on building our business logic and not thinking of project setup.
* It reduces development code by avoiding a lot of boilerplate code.
* Easier to integrate Spring Boot Application with Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security, etc.
* You really don’t need those big XML configurations for your project.
* Embed Tomcat, Jetty or Undertow directly.
* Provide opinionated Maven POM to simplify your configuration.

1. What are the different Spring Boot Components?

* Boot Initializer
* Spring Boot Starter
* Auto Configurator.
* Spring Boot CLI.
* Actuator.

1. What is Spring Boot Starters?

Spring Boot Starters are the set of convenient dependency descriptors which can be easily included in any level of application. These starters work as a bootstrapping process for the Spring related technologies, we no longer need to worry about the dependencies and they will be automatically managed by Spring Boot Starters.

The starters contain a lot of the dependencies you need to get a project up and running quickly and with a consistent, supported a set of managed transitive dependencies. To summarize, Spring Boot Starters are just JAR files used by Spring Boot for auto-dependency.

1. Name some starter provided by Spring Boot?

* spring-boot-starter-web: Web and RESTful applications
* spring-boot-starter-security: Spring Security
* spring-boot-starter-data-jpa: Spring Data JPA
* spring-boot-starter-test: Unit testing
* spring-boot-starter-hateoas: Add HATEOAS features
* spring-boot-starter-data-jpa: Spring Data JPA with Hibernate

1. How can I reload my Spring Boot changes without restarting the server?

<dependencies>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

1. What are the embedded containers supported by Spring Boot?

* Tomcat
* Jetty
* Undertow.

1. What is the Spring Boot Actuator?

The actuator provides production-ready features for Spring Boot application. It will help us check and manage our application in the production environment. We need none code to get these features since they are available once the actuator dependency is in the class-path. The actuator provides features like auditing, health, metrics, environment information, thread dump etc. using HTTP endpoints. Read Spring Boot Actuator for more detail.

1. How can we create a custom endpoint in Spring Boot Actuator?

To create a custom endpoint using Spring Boot 1.x, we should expose the instance of the custom endpoint class as a bean. We need to implement Endpoint<T> interface.

@Component

public class CustomEndpoint implements Endpoint {

//method implimentation

}

Spring Boot 2.x changed it by introducing @Endpoint annotation. Spring Boot expose endpoints with @Endpoint@WebEndpointor @WebEndpointExtension over HTTP using Jersey, Spring MVC, or Spring WebFlux

1. How to run Spring Boot application to custom port?

server.port=9001

1. How to implement security for Spring boot application?

Use the spring-boot-starter-security starter to enable the Spring security support in your Spring Boot application.

<dependency>

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

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

1. W
2. Q
3. W
4. W
5. W
6. W
7. W
8. W
9. W
10. W
11. W
12. w

*Features of Spring boot-*

* **Auto-Configuration** - No need to manually configure dispatcher servlet, static resource mappings, property source loader, message converters etc.
* **Dependency Management** - The different versions of commonly used libraries are pre-selected and grouped in different starter POMs that we can include in your project. By selecting one Spring Boot version we are implicitly selecting dozens of dependencies that we would have to otherwise select and harmonize ourself. Example-
* **Advanced Externalized Configuration** - There is a large list of bean properties that can be configured through application.properties file without touching java or xml config.
* **Production support**- We get health checking, application and jvm metrics, jmx via http and a few more things for free.
* **Runnable Jars** - We can package your application as a runnable jar with embedded tomcat included so it presents a self-contained deployment unit

//Enables functionala pgm

//readable and concise code (elimiate unnecesary code)

//easier to use API and librares

//enable parallel processing

//Funcation programing - better code, readable code, and maintanable code

Streams

[*https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/*](https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/)

*In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together.*

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons.

Streams can be created from various data sources, especially collections. Lists and Sets support new methods stream() and parallelStream()

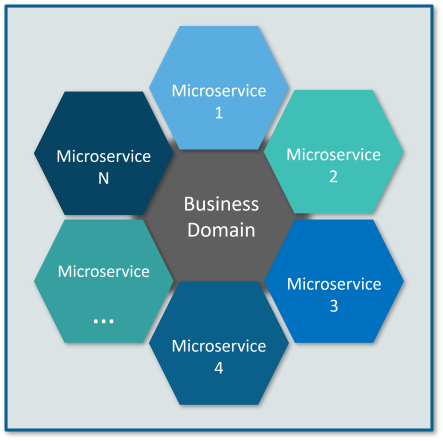
Microservice

### ****Q1. List down the advantages of Microservices Architecture.****

|  |  |
| --- | --- |
| **Advantage** | **Description** |
| **Independent Development** | All microservices can be easily developed based on their individual functionality |
| **Independent Deployment** | Based on their services, they can be individually deployed in any application |
| **Fault Isolation** | Even if one service of the application does not work, the system still continues to function |
| **Mixed Technology Stack** | Different languages and technologies can be used to build different services of the same application |
| **Granular Scaling** | Individual components can scale as per need, there is no need to scale all components together |

### Q2. What do you know about Microservices?

* **Microservices**, aka ***Microservice Architecture***, is an architectural style that structures an application as a collection of small autonomous services, modeled around a **business domain.**
* In layman terms, you must have seen how bees build their honeycomb by aligning hexagonal wax cells.
* They initially start with a small section using various materials and continue to build a large beehive out of it.
* These cells form a pattern resulting in a strong structure which holds together a particular section of the beehive.
* Here, each cell is independent of the other but it is also correlated with the other cells.
* This means that damage to one cell does not damage the other cells, so, bees can reconstruct these cells without impacting the complete beehive.



**Fig 1:** Beehive Representation of Microservices – Microservices Interview Questions

Refer to the above diagram. Here, each hexagonal shape represents an individual service component. Similar to the working of bees, each agile team builds an individual service component with the available frameworks and the chosen technology stack. Just as in a beehive, each service component forms a strong microservice architecture to provide better scalability. Also, issues with each service component can be handled individually by the agile team with no or minimal impact on the entire application.

### ****Q3. What are the features of Microservices?****

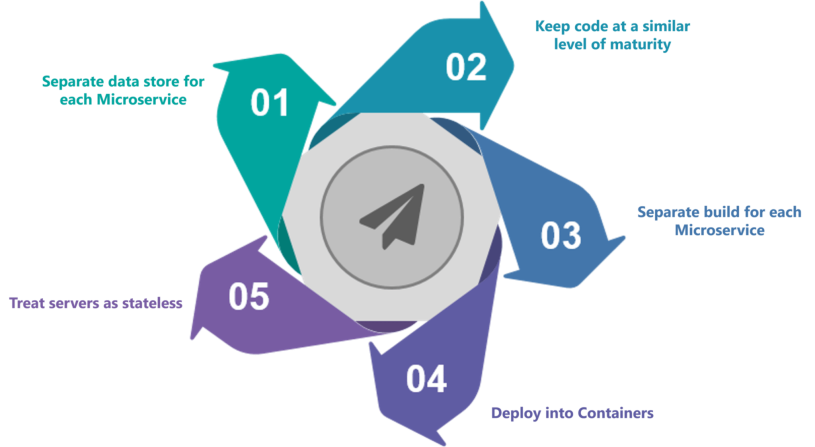


**Fig 3:**Features of Microservices – Microservices Interview Questions

* **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
* **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
* **Business Capabilities** – Microservices are very simple and focus on a single capability
* **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
* **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
* **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
* **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
* **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

### ****Q4. What are the best practices to design Microservices?****

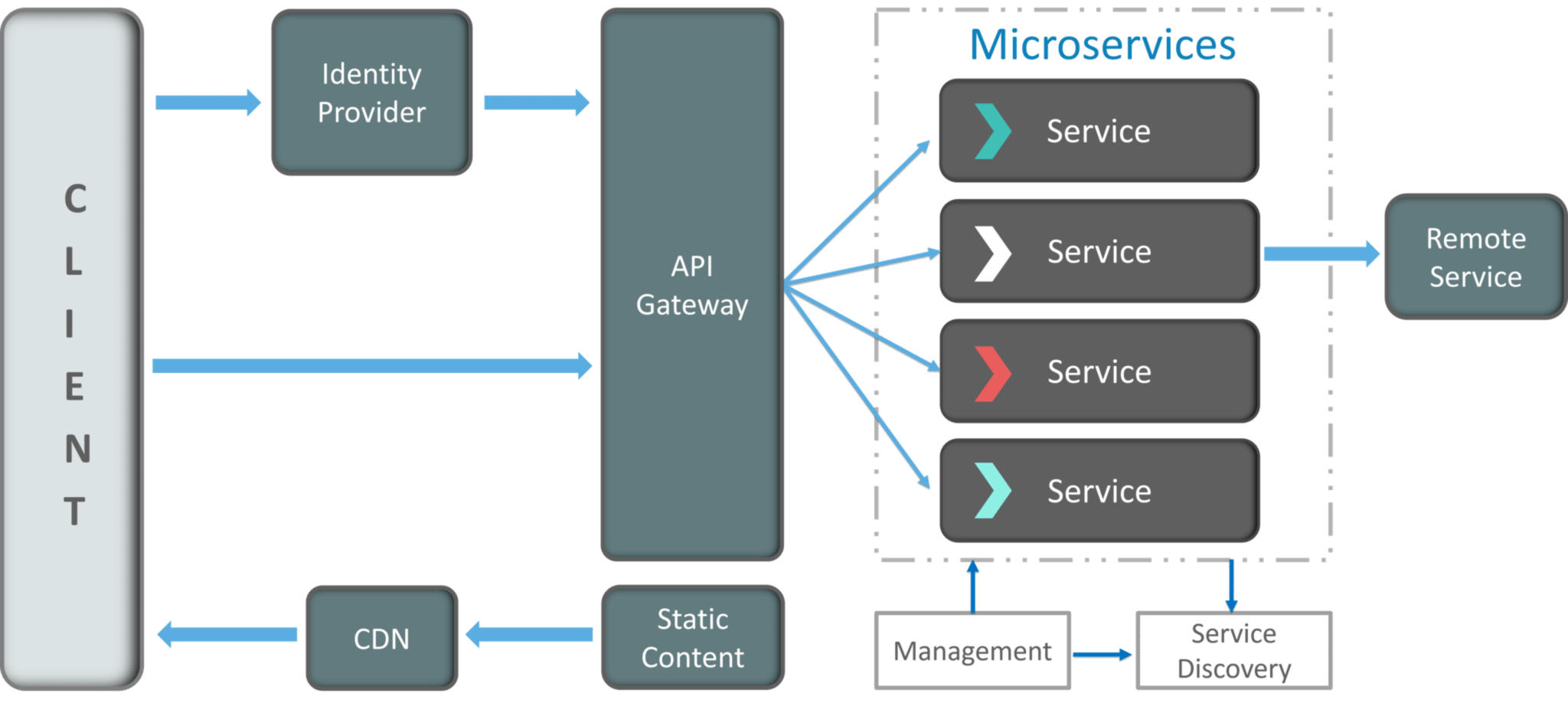
The following are the best practices to design microservices:



**Fig 4:**Best Practices to Design Microservices – Microservices Interview Questions

### ****Q5. How does Microservice Architecture work?****

A microservice architecture has the following components:



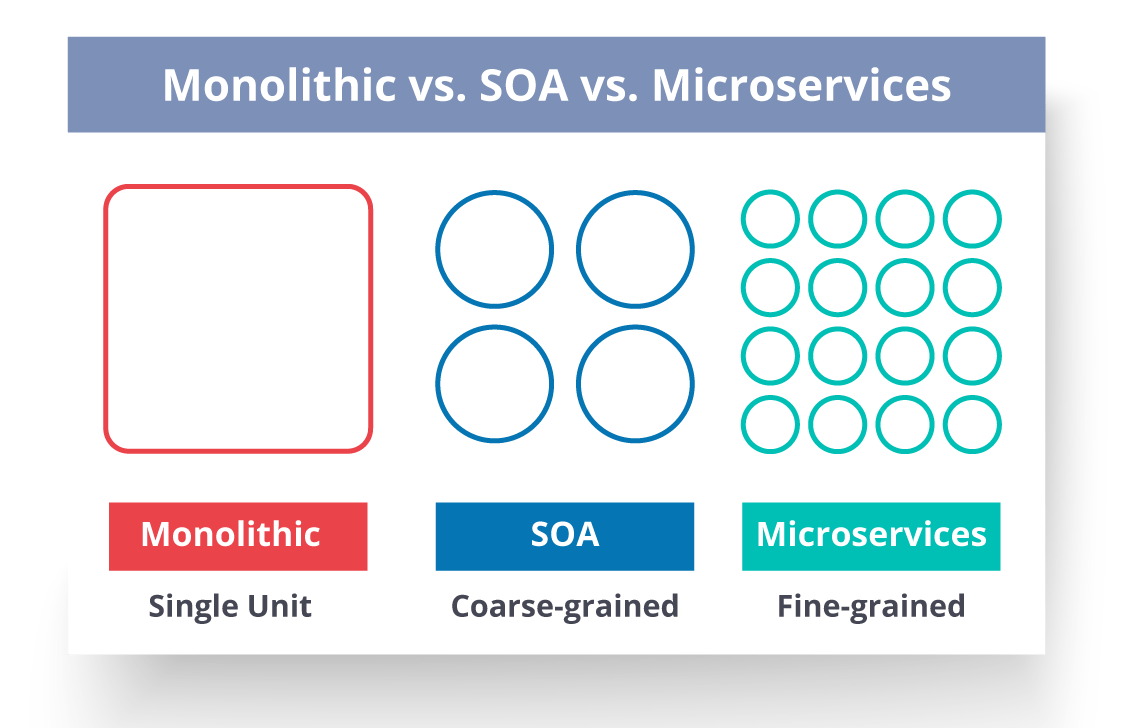
**Fig 5:**Architecture of Microservices – Microservices Interview Questions

* **Clients** – Different users from various devices send requests.
* **Identity Providers** – Authenticates user or clients identities and issues security tokens.
* **API Gateway** – Handles client requests.
* **Static Content** – Houses all the content of the system.
* **Management** –  Balances services on nodes and identifies failures.
* **Service Discovery** – A guide to find the route of communication between microservices.
* **Content Delivery Networks** – Distributed network of proxy servers and their data centers.
* **Remote Service** – Enables the remote access information that resides on a network of IT devices.

### ****Q6. What are the pros and cons of Microservice Architecture?****

|  |  |
| --- | --- |
| **Pros of Microservice Architecture** | **Cons of Microservice Architecture** |
| Freedom to use different technologies | Increases troubleshooting challenges |
| Each microservices focuses on single capability | Increases delay due to remote calls |
| Supports individual deployable units | Increased efforts for configuration and other operations |
| Allow frequent software releases | Difficult to maintain transaction safety |
| Ensures security of each service | Tough to track data across various boundaries |
| Mulitple services are parallelly developed and deployed | Difficult to code between services |

### ****Q7. What is the difference between Monolithic, SOA and Microservices Architecture?****



**Fig 6:**Comparison Between Monolithic SOA & Microservices – Microservices Interview Questions

* **Monolithic Architecture** is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
* A **Service-Oriented Architecture** is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
* **Microservice Architecture** is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.

### ****Q8. What are the challenges you face while working Microservice Architectures?****

Developing a number of smaller microservices sounds easy, but the challenges often faced while developing them are as follows.

* **Automate the Components**: Difficult to automate because there are a number of smaller components. So for each component, we have to follow the stages of  Build, Deploy and, Monitor.
* **Perceptibility**: Maintaining a large number of components together becomes difficult to deploy, maintain, monitor and identify problems. It requires great perceptibility around all the components.
* **Configuration Management**: Maintaining the configurations for the components across the various environments becomes tough sometimes.
* **Debugging**: Difficult to find out each and every service for an error. It is essential to maintain centralized logging and dashboards to debug problems.

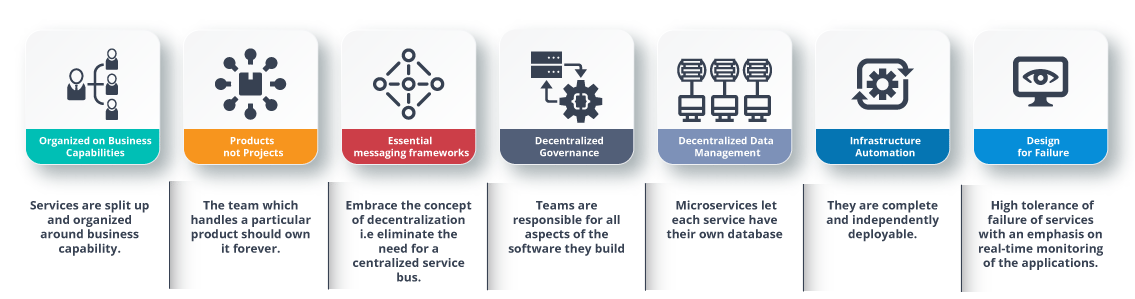
### Q9. What are the key differences between SOA and Microservices Architecture?

The key differences between SOA and microservices are as follows:

|  |  |
| --- | --- |
| **SOA** | **Microservices** |
| Follows “**share-as-much-as-possible**” architecture approach | Follows “**share-as-little-as-possible**” architecture approach |
| Importance is on **business functionality** reuse | Importance is on the concept of “**bounded context**” |
| They have **common** **governance** and standards | They focus on **people** **collaboration** and freedom of other options |
| Uses **Enterprise Service bus (ESB)** for communication | Simple messaging system |
| They support **multiple message protocols** | They use **lightweight protocols** such as **HTTP/REST** etc. |
| **Multi-threaded** with more overheads to handle I/O | **Single-threaded** usually with the use of Event Loop features for non-locking I/O handling |
| Maximizes application service reusability | Focuses on **decoupling** |
| **Traditional Relational Databases** are more often used | **Modern Relational Databases**are more often used |
| A systematic change requires modifying the monolith | A systematic change is to create a new service |
| DevOps / Continuous Delivery is becoming popular, but not yet mainstream | Strong focus on DevOps / Continuous Delivery |

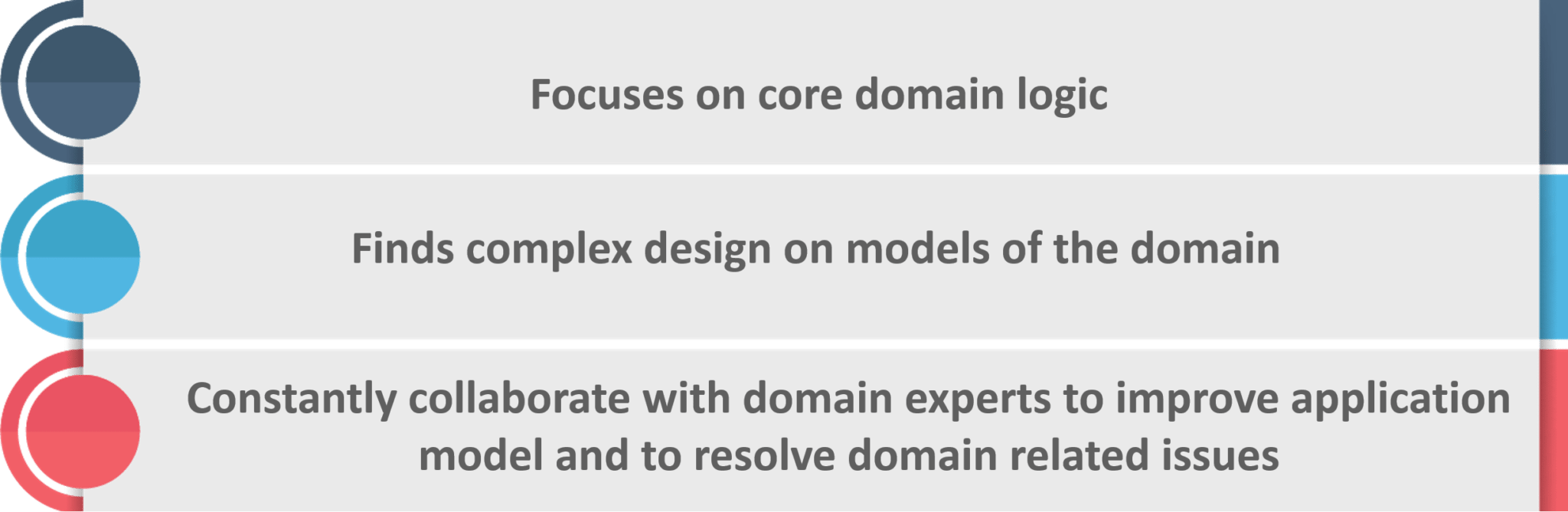
### ****Q10. What are the characteristics of Microservices?****

You can list down the characteristics of microservices as follows:



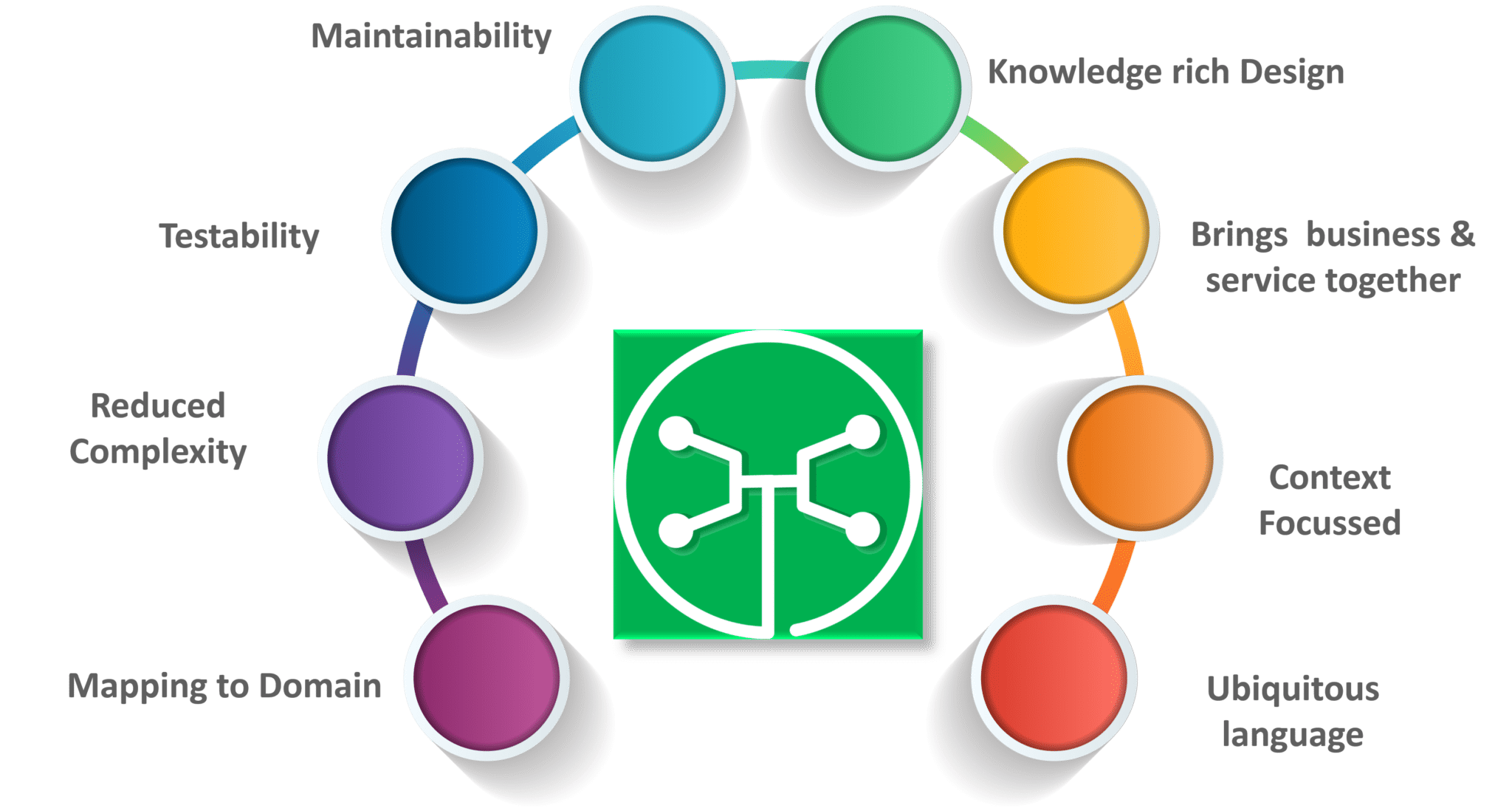
**Fig 7:**Characteristics of Microservices – Microservices Interview Questions

### ****Q11. What is Domain Driven Design?****



**Fig 8:**Principles of DDD – Microservices Interview Questions

### ****Q12. Why there is a need for Domain Driven Design (DDD)?****



**Fig 9:**Factors Why we need DDD – Microservices Interview Questions

### ****Q13. What is Ubiquitous language?****

If you have to define the**Ubiquitous Language (UL)**, then it is a common language used by developers and users of a specific domain through which the domain can be explained easily.

The ubiquitous language has to be crystal clear so that it brings all the team members on the same page and also translates in such a way that a machine can understand.

### ****Q14. What is Cohesion?****

The degree to which the elements inside a module belong together is said to be **cohesion**.

### ****Q15.  What is Coupling?****

The measure of the strength of the dependencies between components is said to be **coupling**. A good design is always said to have **High Cohesion** and**Low Coupling**.

#### Subscribe to our youtube channel to get new updates..!

### ****Q16.  What is REST/RESTful and what are its uses?****

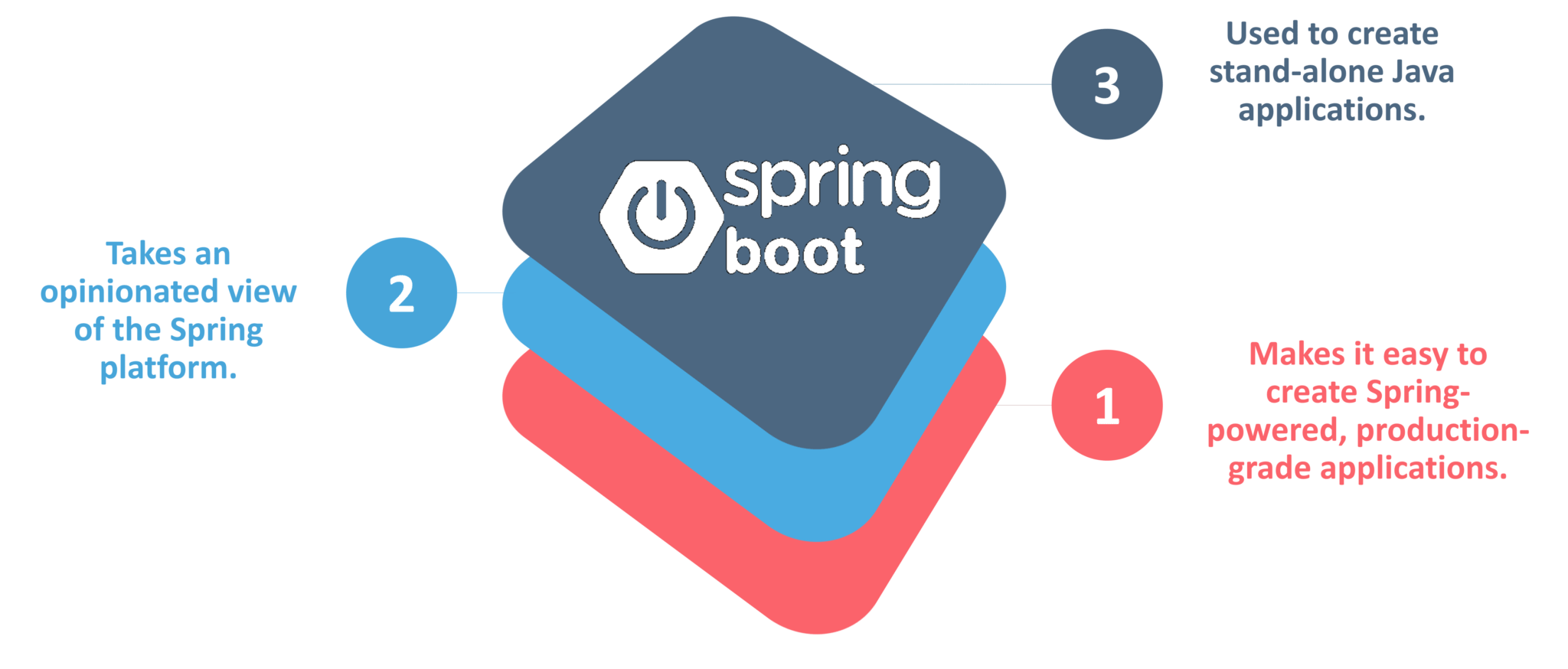
**Representational State Transfer (REST)/RESTful** web services are an architectural style to help computer systems communicate over the internet. This makes microservices easier to understand and implement.

Microservices can be implemented with or without RESTful APIs, but it’s always easier to build loosely coupled microservices using RESTful APIs.

### ****Q17. What do you know about Spring Boot?****

It’s a knows fact that spring has become more and more complex as new functionalities have been added. If you have to start a new spring project, then you have to add build path or add maven dependencies, configure application server, add spring configuration. So everything has to be done from scratch.

**Spring Boot** is the solution to this problem. Using spring boot you can avoid all the boilerplate code and configurations. So basically consider yourself as if you’re baking a cake spring is like the ingredients that are required to make the cake and spring boot is the complete cake in your hand.



**Fig 10:**Factors of Spring Boot – Microservices Interview Questions

### ****Q18. What is an actuator in Spring boot?****

Spring Boot actuator provides restful web services to access the current state of running an application in the production environment. With the help of actuator, you can check various metrics and monitor your application.

### ****Q19. What is Spring Cloud?****

According to the official website of Spring Cloud, Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, leadership election, distributed sessions, cluster state).

### ****Q20. What problems are solved by Spring Cloud?****

While developing distributed microservices with Spring Boot we face few issues which are solved by Spring Cloud.

* **The complexity associated with distributed systems –**This includes network issues, Latency overhead, Bandwidth issues, security issues.
* **Ability to handle Service Discovery –**Service discovery allows processes and services in a cluster to find each other and communicate.
* **Solved redundancy issues –**Redundancy issues often occur in distributed systems.
* **Load balancing –**Improves the distribution of workloads across multiple computing resources, such as a computer cluster, network links, central processing units.
* **Reduces performance issues –**Reduces performance issues due to various operational overheads.

**[Microservices Architecture Training](https://www.edureka.co/microservices-architecture-training" \t "_blank)**[Watch The Course Preview](https://www.edureka.co/microservices-architecture-training" \t "_blank)

### ****Q21.  What is the use of WebMvcTest annotation in Spring MVC applications?****

**WebMvcTest** annotation is used for unit testing Spring MVC Applications in cases where the test objective is to just focus on Spring MVC Components. In the snapshot shown above, we want to launch only the ToTestController. All other controllers and mappings will not be launched when this unit test is executed.

### ****Q22. Can you give a gist about Rest and Microservices?****

#### ****REST****

Though you can implement microservices in multiple ways, REST over HTTP is a way to implement Microservices. REST is also used in other applications such as web apps, API design, and MVC applications to serve business data.

#### ****Microservices****

Microservices is an architecture wherein all the components of the system are put into individual components, which can be built, deployed, and scaled individually. There are certain principles and best practices of Microservices that help in building a resilient application.

In a nutshell, you can say that REST is a medium to build Microservices.

### ****Q23. What are different types of Tests for Microservices?****

While working with microservices, testing becomes quite complex as there are multiple microservices working together. So, tests are divided into different levels.

* At the **bottom level**, we have **technology-facing tests** like- unit tests and performance tests. These are completely automated.
* At the **middle level**, we have tests for **exploratory testing** like the stress tests and usability tests.
* At the **top level,**we have **acceptance tests** that are few in number. These acceptance tests help stakeholders in understanding and verifying software features.

### ****Q24. What do you understand by Distributed Transaction?****

**Distributed Transaction** is any situation where a single event results in the mutation of two or more separate sources of data which cannot be committed atomically. In the world of microservices, it becomes even more complex as each service is a unit of work and most of the time multiple services have to work together to make a business successful.

### ****Q25. What is an Idempotence and where it is used?****

**Idempotence** is the property of being able to do something twice in such a way that the end result will remain the same i.e. as if it had been done once only.

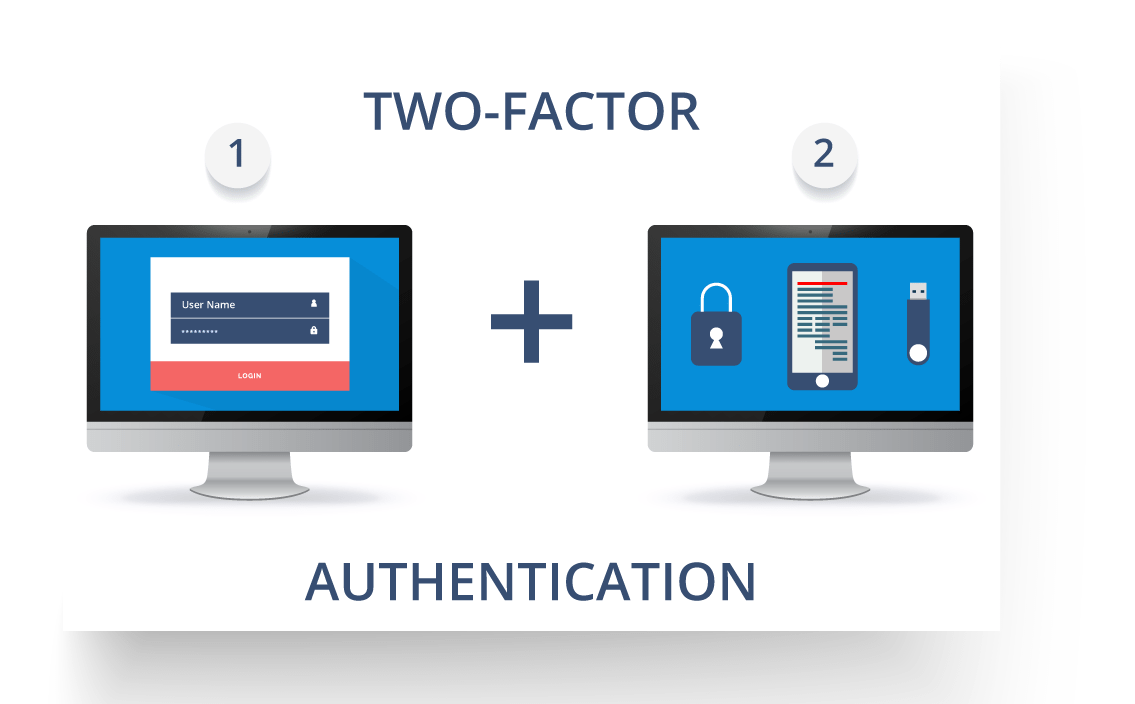
**Usage**: Idempotence is used at the remote service, or data source so that, when it receives the instruction more than once, it only processes the instruction once.

### ****Q26. What is Bounded Context?****

Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD’s strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their inter-relationships.

### ****Q27. What is Two Factor Authentication?****

Two-factor authentication enables the second level of authentication to an account log-in process.

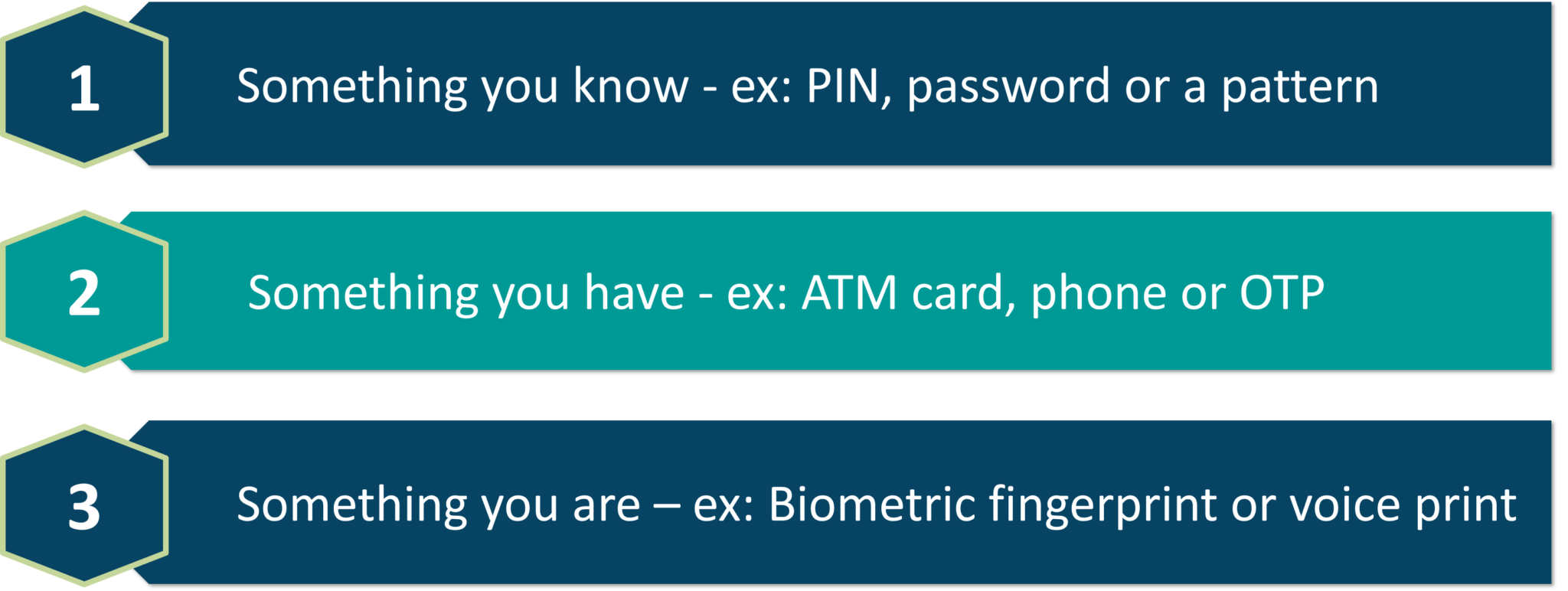


**Fig11:**Representation of Two Factor Authentication – Microservices Interview Questions

So suppose a user has to enter only username and password, then that’s considered a single-factor authentication.

### ****Q28. What are the types of credentials of Two Factor Authentication?****

The three types of credentials are:



**Fig 12:**Types of Credentials of Two Factor Authentication – Microservices Interview Questions

### ****Q29. What are Client certificates?****

A type of digital certificate that is used by client systems to make authenticated requests to a remote server is known as the**client certificate**. Client certificates play a very important role in many mutual authentication designs, providing strong assurances of a requester’s identity.

### ****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:****

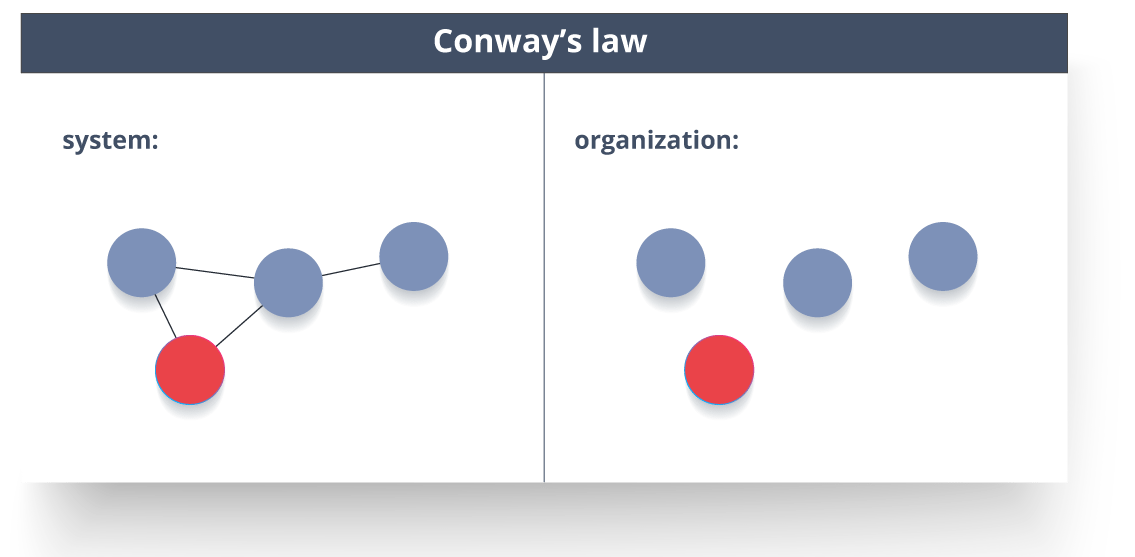
* Used to implement Consumer Driven Contract in Microservices.
* Tests the consumer-driven contracts between consumer and provider of a Microservice.

### ****Q31. What is OAuth?****

**OAuth**stands for open authorization protocol. This allows accessing the resources of the resource owner by enabling the client applications on HTTP services such as third-party providers Facebook, GitHub, etc. So with this, you can share resources stored on one site with another site without using their credentials.

### ****Q32. What is Conway’s law?****

“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure.” –***Mel Conway***



**Fig 13:**Representation of Conway’s Law – Microservices Interview Questions

This law basically tries to convey the fact that, in order for a software module to function, the complete team should communicate well. Therefore the structure of a system reflects the social boundaries of the organization(s) that produced it.

### ****Q33. What do you understand by Contract Testing?****

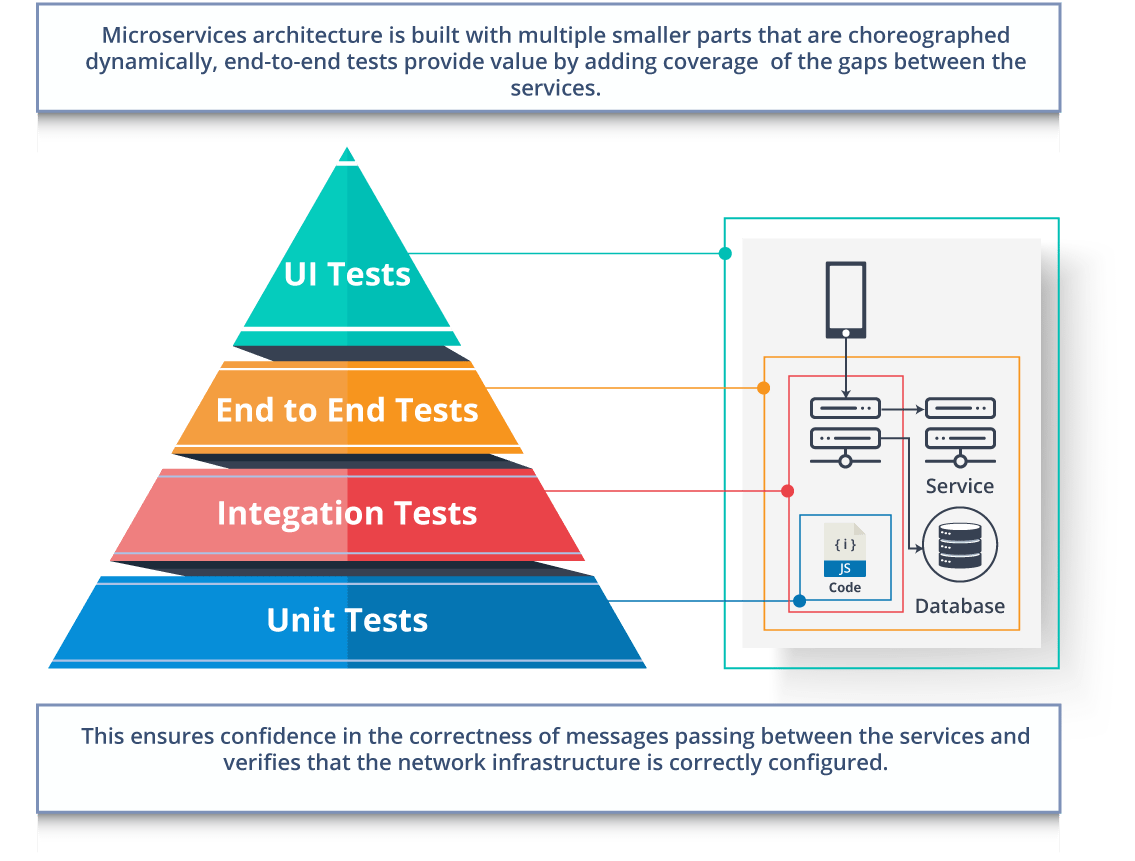
According to Martin Flower,**contract test**is a test at the boundary of an external service which verifies that it meets the contract expected by a consuming service.

Also, contract testing does not test the behavior of the service in depth. Rather, it tests that the inputs & outputs of service calls contain required attributes and the response latency, throughput is within allowed limits.

### ****Q34. What is End to End Microservices Testing?****

End-to-end testing validates each and every process in the workflow is functioning properly. This ensures that the system works together as a whole and satisfies all requirements.

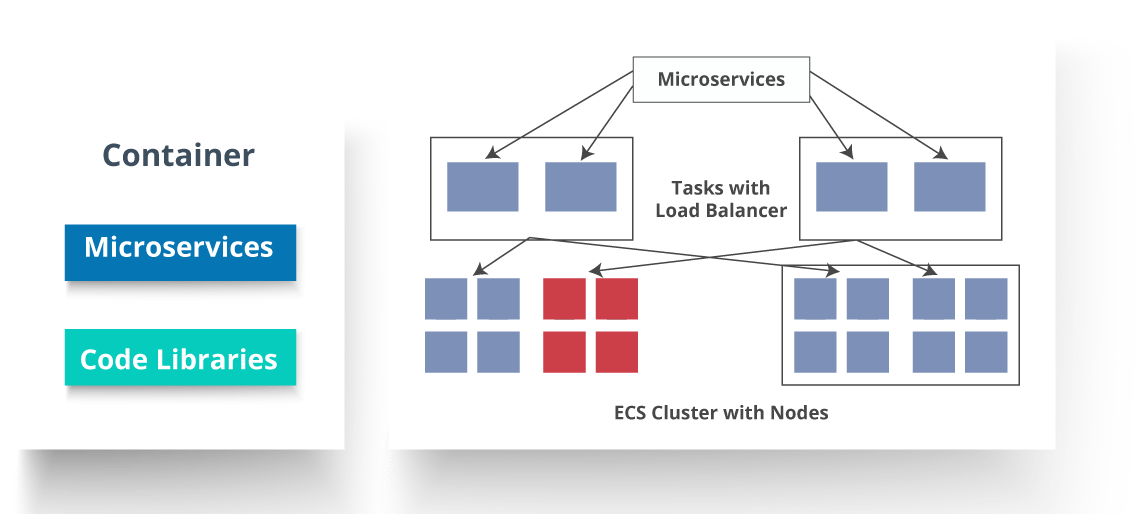
In layman terms, you can say that end to end testing is a kind of tests where everything is tested after a particular period.



**Fig 14:**Hierarchy of Tests – Microservices Interview Questions

### ****Q35. What is the use of Container in Microservices?****

Containers are a good way to manage microservice based application to develop and deploy them individually*.* You can encapsulate your microservice in a container image along with its dependencies, which then can be used to roll on-demand instances of microservice without any additional efforts required.



**Fig 15:**Representation of Containers and How they are used in Microservices – Microservices Interview Questions

### ****Q36. What is DRY in Microservices architecture?****

**DRY** stands for **Don’t Repeat Yourself**. It basically promotes the concept of reusing the code. This results in developing and sharing the libraries which in turn result in tight coupling.

### ****Q37. 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.

**[Microservices Architecture Training](https://www.edureka.co/microservices-architecture-training" \t "_blank)**[Watch The Course Preview](https://www.edureka.co/microservices-architecture-training" \t "_blank)

### ****Q38****. What is the role of Web, RESTful APIs in Microservices?

A microservice architecture is based on a concept wherein all its services should be able to interact with each other to build a business functionality. So, to achieve this, each microservice must have an interface. This makes the web API a very important enabler of microservices. Being based on the open networking principles of the Web, RESTful APIs provide the most logical model for building interfaces between the various components of a microservice architecture.

### ****Q39. What do you understand by Semantic monitoring in Microservices architecture?****

Semantic monitoring, also known as**synthetic monitoring** combines automated tests with monitoring the application in order to detect business failing factors.

### ****Q40. How can we perform Cross-Functional testing?****

Cross-functional testing is a verification of non-functional requirements, i.e. those requirements which cannot be implemented like a normal feature.

### ****Q41. How can we eradicate non-determinism in tests?****

**Non-Deterministic Tests** (NDT)  are basically unreliable tests.  So, sometimes it may happen that they pass and obviously sometimes they may also fail. As and when they fail, they are made to re-run to pass.

Some ways to remove non-determinism from tests are as follows:

1. Quarantine
2. Asynchronous
3. Remote Services
4. Isolation
5. Time
6. Resource leaks

### ****Q42. What is the difference between Mock or Stub?****

#### ****Stub****

* A dummy object that helps in running the test.
* Provides fixed behavior under certain conditions which can be hard-coded.
* Any other behavior of the stub is never tested.

For example, for an empty stack, you can create a stub that just returns true for empty() method. So, this does not care whether there is an element in the stack or not.

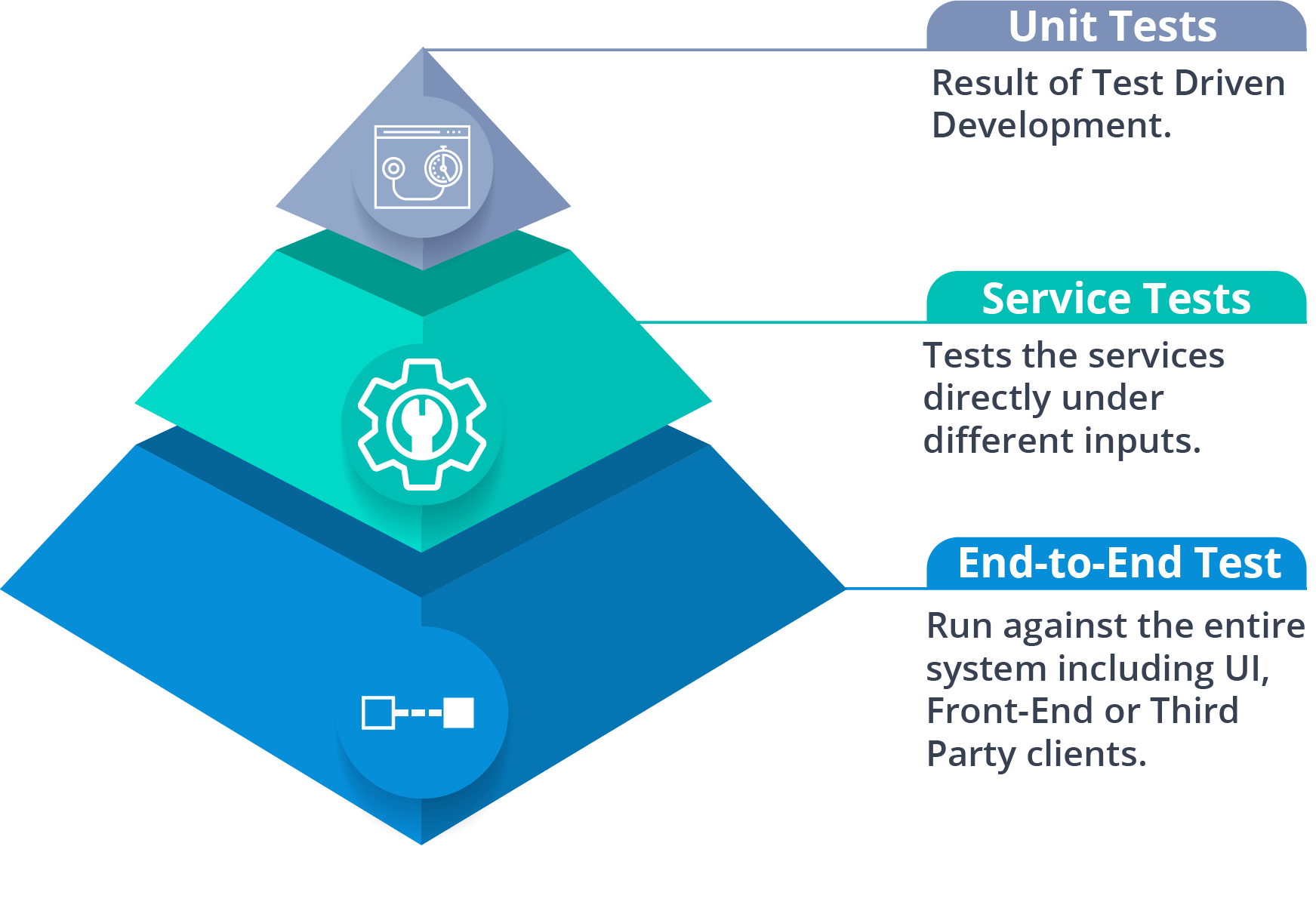
#### ****Mock****

* A dummy object in which certain properties are set initially.
* The behavior of this object depends on the set properties.
* The object’s behavior can also be tested.

For example, for a Customer object, you can mock it by setting name and age. You can set age as 12 and then test for isAdult() method that will return true for age greater than 18. So, your Mock Customer object works for the specified condition.

### ****Q43. What do you know about Mike Cohn’s Test Pyramid?****

**Mike Cohn** provided a model called **Test Pyramid.** This describes the kind of automated tests required for software development.



**Fig 16:**Mike Cohn’s Test Pyramid – Microservices Interview Questions

As per pyramid, the number of tests at first layer should be highest. At service layer, the number of tests should be less than at the unit test level, but more than at the end-to-end level.

### ****Q44. What is the purpose of Docker?****

**Docker** provides a container environment that can be used to host any application. In this, the software application and the dependencies which support it are tightly-packaged together.

So, this packaged product is called a **Container** and since it is done by Docker, it is called **Docker container!**

**[Microservices Architecture Training](https://www.edureka.co/microservices-architecture-training" \t "_blank)**[Watch The Course Preview](https://www.edureka.co/microservices-architecture-training" \t "_blank)

### ****Q45. What is Canary Releasing?****

**Canary Releasing** is a technique to reduce the risk of introducing a new software version in production. This is done by slowly rolling out the change to a small subset of users before giving it out to the entire infrastructure, i.e. making it available to everybody.

### Q46. What do you mean by Continuous Integration (CI)?

**Continuous Integration (CI)** is the process of automating the build and testing of code every time a team member commits changes to version control. This encourages developers to share code and unit tests by merging the changes into a shared version control repository after every small task completion.

### ****Q47. What is Continuous Monitoring?****

**Continuous monitoring**gets into the depth of monitoring coverage, from in-browser front-end performance metrics, through application performance, and down to host virtualized infrastructure metrics.

### ****Q48. What is the role of an architect in Microservices architecture?****

An architect in microservices architecture plays the following roles:

* Decides broad strokes about the layout of the overall software system.
* Helps in deciding the zoning of the components. So, they make sure components are mutually cohesive, but not tightly coupled.
* Code with developers and learn the challenges faced in day-to-day life.
* Make recommendations for certain tools and technologies to the team developing microservices.
* Provide technical governance so that the teams in their technical development follow principles of Microservice.

### ****Q49. Can we create State Machines out of Microservices?****

As we know that each Microservice owning its own database is an independently deployable program unit, this, in turn, lets us create a State Machine out of it. So, we can specify different states and events for a particular microservice.

For Example, we can define an Order microservice. An Order can have different states. The transitions of Order states can be independent events in the Order microservice.

### Q50. What are Reactive Extensions in Microservices?

Reactive Extensions also are known as Rx. It is a design approach in which we collect results by calling multiple services and then compile a combined response. These calls can be synchronous or asynchronous, blocking or non-blocking. Rx is a very popular tool in distributed systems which works opposite to legacy flows.

Hope these Microservices Interview Questions would help you in your Microservices Architect Interviews.

If you wish to learn Microservices and build your own applications, then check out our [***Microservices Architecture Training***](https://www.edureka.co/microservices-architecture-training) which comes with instructor-led live training and real-life project experience. This training will help you understand Microservices in depth and help you achieve mastery over the subject.

Docker

**What is Docker?**

* Docker is a containerization platform which packages your application and all its dependencies together in the form of containers so as to ensure that your application works seamlessly in any environment be it development or test or production.
* Internally it achieves it by using kernel containerization feature.
* Docker containers, wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries etc. anything that can be installed on a server.
* This guarantees that the software will always run the same, regardless of its environment.

**What is the advantage of Docker over hypervisors (Virtual Machine)?**

Docker is lightweight and more efficient in terms of resource uses because it uses the underlying host kernel rather than creating its hypervisor.

**What is Docker image?**

Docker image is the source of Docker container. In other words, Docker images are used to create containers. Images are created with the build command, and they’ll produce a container when started with run. Images are stored in a Docker registry such as registry.hub.docker.com because they can become quite large, images are designed to be composed of layers of other images, allowing a minimal amount of data to be sent when transferring images over the network.  
**What is Docker container?**

* Docker containers are basically runtime instances of Docker images.
* Docker containers include the application and all of its dependencies, but share the kernel with other containers, running as isolated processes in user space on the host operating system. Docker containers are not tied to any specific infrastructure: they run on any computer, on any infrastructure, and in any cloud.  
  Now explain how to create a Docker container, Docker containers can be created by either creating a Docker image and then running it or you can use Docker images that are present on the Dockerhub.

**What is Docker hub?**

Docker hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker cloud so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

**How is Docker different from other container technologies?**

Docker containers are easy to deploy in a cloud. It can get more applications running on the same hardware than other technologies, it makes it easy for developers to quickly create, ready-to-run containerized applications and it makes managing and deploying applications much easier. You can even share containers with your applications.  
If you have some more points to add you can do that but make sure the above the above explanation is there in your answer.

**What is Docker Swarm?**

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host. Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

I will also suggest you to include some supported tools:

* Dokku
* Docker Compose
* Docker Machine
* Jenkins

**What is Dockerfile used for?**

Docker can build images automatically by reading the instructions from a Dockerfile.

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

FROM artifactory.global.standardchartered.com/com/sc/rtdo/jdk:jdk-rhel-2

LABEL service="cashpi-services-benematch-service"

LABEL maintainer="KarthikVignesh.E1@sc.com"

USER root

RUN mkdir -p /app/cashpi-services-benematch-service && \

chmod -R 777 /app/cashpi-services-benematch-service && \

mkdir -p /app/cashpi-services-benematch-service/logs && \

mkdir -p /app/logs && \

mkdir -p /var/log && \

chmod -R 777 /app/cashpi-services-benematch-service/logs && \

chmod -R 777 /app/logs && \

chmod -R 777 /var/log

ADD target/cashpi-services-benematch-service-0.0.1.jar /app/cashpi-services-benematch-service/cashpi-services-benematch-service.jar

EXPOSE 8080

ENTRYPOINT [ "sh", "-c", "/opt/jdk8/jdk1.8.0\_91/jre/bin/java \

-DAPP\_NAME=cashpi-services-benematch-service -DSPRING\_JTA\_LOG=/app/cashpi-services-benematch-service/logs \

-jar /app/cashpi-services-benematch-service/cashpi-services-benematch-service.jar" ]

**Can I use json instead of yaml for my compose file in Docker?**

You can use json instead of yaml for your compose file, to use json file with compose, specify the filename to use for eg:  
**docker-compose -f docker-compose.json up**

**How to create Docker container?**

I will suggest you to give a direct answer to this.

We can use Docker image to create Docker container by using the below command:

|  |  |
| --- | --- |
|  | docker run -t -i command name |

This command will create and start a container.

You should also add, If you want to check the list of all running container with the status on a host use the below command:

|  |  |
| --- | --- |
|  | docker ps -a |

**How to stop and restart the Docker container?**

In order to stop the Docker container you can use the below command:

|  |  |
| --- | --- |
|  | docker stop container ID |

Now to restart the Docker container you can use:

|  |  |
| --- | --- |
|  | docker restart container ID |

How far do Docker containers scale?

Large web deployments like Google and Twitter, and platform providers such as Heroku and dotCloud all run on container technology, at a scale of hundreds of thousands or even millions of containers running in parallel.

What platforms does Docker run on?

I will start this answer by saying Docker runs on only Linux and Cloud platforms and then I will mention the below vendors of Linux:

* Ubuntu 12.04, 13.04 et al
* Fedora 19/20+
* RHEL 6.5+
* CentOS 6+
* Gentoo
* ArchLinux
* openSUSE 12.3+
* CRUX 3.0+

Cloud:

* Amazon EC2
* Google Compute Engine
* Microsoft Azure
* Rackspace

**Do I lose my data when the Docker container exits?**

You can answer this by saying, no I won’t lose my data when Docker container exits, any data that your application writes to disk gets preserved in its container until you explicitly delete the container. The file system for the container persists even after the container halts.

#### What is the lifecycle of Docker Container?

1. Create a container.
2. Run the Docker container.
3. Pause the Container.
4. Unpause the Container.
5. Start the Container.
6. Stop the Container.
7. Restart the Container.
8. Kill the Container.
9. Destroy the Container.

Docker Comments

docker login -u jenkins\_resolver -p \*\*\*\*\*\*\*\* https://artifactory.global.standardchartered.com/

docker build -t com/sc/cashpi/cashpi/cashpi-achprocess/master:cashpi-achprocess\_master-9682-0.0.682 --rm=true

execute docker file

docker tag com/sc/cashpi/cashpi/cashpi-achprocess/master:cashpi-achprocess\_master-9682-0.0.682 artifactory.global.standardchartered.com/com/sc/cashpi/cashpi/cashpi-achprocess/master:cashpi-achprocess\_master-9682-0.0.682

docker push artifactory.global.standardchartered.com/com/sc/cashpi/cashpi/cashpi-achprocess/master:cashpi-achprocess\_master-9682-0.0.682

OC Comments

oc login https://ocp-console-cib-np.ocp.standardchartered.com:8443 -u=cash -p=poiu1234 --insecure-skip-tls-verify=true

oc login

oc version

oc project dqmf

oc project

oc config view

oc status

oc get svc

oc get pods

oc describe svc cashpi-services-gcg-ach-outwardclient

oc get pod cashpi-achprocess-11-dm2d9

oc adm top pods --> Used to find the statistics

oc adm top pod --selector='cashpi-services-gcg-ach-outwardclient'

oc get poddisruptionbudget --all-namespaces

oc policy who-can get imagestreams/layers -n openshift --config=path/to/admin.kubeconfig --> Used to find who has rights to pull images from the openshift namespace.

oc get ClusterRoles | grep persistent-

--------------------service account----------------

oc get sa --> Get Service Account Details

oc describe serviceaccount builder

oc describe serviceaccount default

oc describe serviceaccount deployer

-----------------------scale up/down---------------------------------------------------------------------

oc scale --replicas=0 dc cashpi-services-sts-payment-processer

oc scale --replicas=1 dc cashpi-services-sts-payment-processer

oc scale dc cashpi-services-sts-payment-processer --replicas=3

oc scale --replicas=0 dc cashpi-achprocess

oc scale --replicas=0 dc cashpi-services-batch-payment-capture

oc scale --replicas=0 dc cashpi-services-batch-payment-download

oc scale --replicas=0 dc cashpi-services-benematch-service

oc scale --replicas=0 dc cashpi-services-inward-ach-payment-process

oc scale --replicas=0 dc cashpi-services-sts-payment-processer

--------------delete pods-------------------------------------------------------------------------------

oc delete pod <pod Name>

--------------logs ---------------------------------------------------------------------------------------

oc logs --follow dc/cashpi-services-sts-payment-processer

oc logs --follow dc/cashpi-services-batch-payment-capture

Object Type Abbreviated Version

build

buildConfig bc

deploymentConfig dc

imageStream is

imageStreamTag istag

imageStreamImage isimage

event ev

node pod

po replicationController

rc service

svc persistentVolume

pv persistentVolumeClaim

pvc

oc get bc cashpi-services-gcg-ach-outwardclient

oc stop

----------------Config Maps Comments------------------

Create from Directories...

cd /var/pi/logs/cashpi/UG/

cat userdetails.properties

oc create configmap test-config \--from-file=/var/pi/logs/cashpi/UG/

oc describe configmaps test-config

oc get configmaps test-config -o yaml

Create from File

oc create configmap test-config-fromfile1 --from-file=/var/pi/logs/cashpi/UG/userdetails.properties --from-file=/var/pi/logs/cashpi/UG/emp.properties

oc describe configmaps test-config-fromfile

Create from KeyPair

oc create configmap test-config1 \--from-literal=name=vadivelan \--from-literal=password=pass1234

oc get configmaps test-config1 -o yaml

---------------Add Config Map to Application/deployment.yaml ---------------------------------------

oc set env dc/cashpi-achprocess --from configmap/test-config1

oc edit pod cashpi-achprocess-12-g9xr8

=======================secrets==============================================================

oc secrets new gcg-keys C:/UG/certs/GCG\_UG\_TS.jks

oc secrets new gcg-keys C:/UG/certs/elastic.jks

/config/security/elk/

oc secrets new gcg-keys /var/pi/logs/cashpi/UG/BEANXML/cashpi-services-gcg-ach-outwardclient/config/security/certs/GCG\_UG\_TS.jks

oc describe secret gcg-keys

oc secrets test\_secrets\_incommand my\_docker\_registry\_resource\_name --docker-server=my\_docker\_registry\_hostname.com --docker-username=myusername --docker-password=mypassword --docker-email=nobody@ca.com

oc delete secret <secret name>

============================Deployment Command=================================================

oc login https://ocp-console-cib-np.ocp.standardchartered.com:8443 -u=cash -p=poiu1234 --insecure-skip-tls-verify=true

oc project cashpi-dev

if oc get dc cashpi-services-inward-ach-payment-process 2> /dev/null; then

oc replace -f cashpi-dev/dc.json

else

oc create -f cashpi-dev/dc.json

fi

if oc get imagestream cashpi-services-inward-ach-payment-process 2> /dev/null; then

echo "imagestream does not exist"

else

oc create -f cashpi-dev/imagestream.json

fi

if oc get service cashpi-services-inward-ach-payment-process 2> /dev/null; then

oc delete service cashpi-services-inward-ach-payment-process

oc create -f cashpi-dev/service.json

else

oc create -f cashpi-dev/service.json

fi

if oc get route cashpi-services-inward-ach-payment-process 2> /dev/null; then

oc delete route cashpi-services-inward-ach-payment-process

oc create -f cashpi-dev/route.json

else

oc create -f cashpi-dev/route.json

fi

oc import-image cashpi-services-inward-ach-payment-process:cashpi-services-inward-ach-payment-process-0.0.1\_develop.4061.73261.build.26 --from=artifactory.global.standardchartered.com/com/sc/cashpi/commonpipeline/cashpi-services-inward-ach-payment-process:cashpi-services-inward-ach-payment-process-0.0.1\_develop.4061.73261.build.26 --confirm

oc tag cashpi-services-inward-ach-payment-process:cashpi-services-inward-ach-payment-process-0.0.1\_develop.4061.73261.build.26 cashpi-services-inward-ach-payment-process:latest

oc logout

Service.JSON

{

"kind": "Service",

"apiVersion": "v1",

"metadata": {

"name": "cashpi-services-benematch-service",

"namespace": "cashpi-dr",

"creationTimestamp": null,

"labels": {

"app": "cashpi-services-benematch-service",

"application": "cashpi-services-benematch-service"

}

},

"spec": {

"ports": [ {

"port": 8080,

"protocol": "TCP",

"name": "http",

"targetPort": 8080

}],

"selector": {

"app": "cashpi-services-benematch-service",

"deploymentconfig": "cashpi-services-benematch-service"

}

},

"status": {"loadBalancer": {}}

}

Dc.JSON

{

"kind": "DeploymentConfig",

"apiVersion": "v1",

"metadata": {

"name": "cashpi-services-benematch-service",

"namespace": "cashpi-dr",

"creationTimestamp": null,

"labels": {

"app": "cashpi-services-benematch-service",

"application": "cashpi-services-benematch-service"

},

"annotations": {}

},

"spec": {

"triggers": [{"type": "ConfigChange"}],

"replicas": 1,

"test": false,

"selector": {

"app": "cashpi-services-benematch-service",

"deploymentconfig": "cashpi-services-benematch-service"

},

"strategy": {

"activeDeadlineSeconds": 21600,

"resources": {},

"rollingParams": {

"intervalSeconds": 1,

"maxSurge": "25%",

"maxUnavailable": "25%",

"timeoutSeconds": 600,

"updatePeriodSeconds": 1

},

"type": "Rolling"

},

"template": {

"metadata": {

"creationTimestamp": null,

"labels": {

"app": "cashpi-services-benematch-service",

"deploymentconfig": "cashpi-services-benematch-service"

}

},

"spec": {

"containers": [ {

"name": "cashpi-services-benematch-service",

"env": [

{

"name": "SPRING\_PROFILES\_ACTIVE",

"value": "dr,native"

},

{

"name": "INIT\_JVM\_MEM\_ALLOC",

"value": "200m"

},

{

"name": "MAX\_JVM\_MEM\_ALLOC",

"value": "2500m"

},

{

"name": "countrycode",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "countrycode"

}}

},

{

"name": "driver",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "driver"

}}

},

{

"name": "holidaycheck",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "holidaycheck"

}}

},

{

"name": "jasypt\_pass",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "jasypt\_pass"

}}

},

{

"name": "maxconnection",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "maxconnection"

}}

},

{

"name": "maxwaitperiod",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "maxwaitperiod"

}}

},

{

"name": "minconnection",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "minconnection"

}}

},

{

"name": "password",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "password"

}}

},

{

"name": "url",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "url"

}}

},

{

"name": "username",

"valueFrom": {"configMapKeyRef": {

"name": "ug-database-details",

"key": "username"

}}

},

{

"name": "elk\_ssl\_path",

"valueFrom": {"configMapKeyRef": {

"name": "ug-logs",

"key": "elk\_ssl\_path"

}}

},

{

"name": "elk\_ip\_port",

"valueFrom": {"configMapKeyRef": {

"name": "ug-logs",

"key": "elk\_ip\_port"

}}

},

{

"name": "elk\_password",

"valueFrom": {"configMapKeyRef": {

"name": "ug-logs",

"key": "elk\_password"

}}

},

{

"name": "log\_path",

"valueFrom": {"configMapKeyRef": {

"name": "ug-logs",

"key": "log\_path"

}}

},

{

"name": "companyid",

"valueFrom": {"configMapKeyRef": {

"name": "ug-ebbs-communication",

"key": "companyid"

}}

},

{

"name": "ebbs\_url",

"valueFrom": {"configMapKeyRef": {

"name": "ug-ebbs-communication",

"key": "ebbs\_url"

}}

},

{

"name": "jksvalue",

"valueFrom": {"configMapKeyRef": {

"name": "ug-ebbs-communication",

"key": "jksvalue"

}}

},

{

"name": "bean\_xml",

"valueFrom": {"configMapKeyRef": {

"name": "ug-bene-match-service",

"key": "bean\_xml"

}}

}

],

"image": "docker-production.artifactory.global.standardchartered.com/com/sc/cashpi/commonpipeline/cashpi-services-benematch-service:cashpi-services-benematch-service-0.0.1\_releasev1.0.0.3239.08311.build.15",

"ports": [ {

"containerPort": 8080,

"name": "http",

"protocol": "TCP"

}],

"resources": {

"limits": {"memory": "2.5G"},

"requests": {"memory": "1.3G"}

},

"imagePullPolicy": "Always",

"terminationMessagePath": "/dev/termination-log",

"terminationMessagePolicy": "File",

"volumeMounts": [

{

"name": "pvol",

"mountPath": "/var/pi/data5/"

},

{

"name": "elk-keystore",

"mountPath": "/config/security/certs/",

"readOnly": true

}

]

}],

"dnsPolicy": "ClusterFirst",

"restartPolicy": "Always",

"schedulerName": "default-scheduler",

"securityContext": {},

"terminationGracePeriodSeconds": 75,

"volumes": [

{

"name": "pvol",

"persistentVolumeClaim": {"claimName": "ug-ach-claim-32"}

},

{

"name": "elk-keystore",

"secret": {

"defaultMode": "420",

"secretName": "elk-keystore"

}

}

]

}

}

},

"status": {

"latestVersion": 0,

"observedGeneration": 0,

"replicas": 0,

"updatedReplicas": 0,

"availableReplicas": 0,

"unavailableReplicas": 0

}

}

Imagestream.JSON

{

"kind": "ImageStream",

"apiVersion": "v1",

"metadata": {

"name": "cashpi-services-benematch-service",

"namespace": "cashpi-dr",

"creationTimestamp": null,

"labels": {

"application": "cashpi-services-benematch-service",

"app": "cashpi-services-benematch-service"

}

},

"spec": {

"lookupPolicy": {"local": false},

"tags": [ {

"name": "latest",

"from": {

"kind": "DockerImage",

"name": "docker-production.artifactory.global.standardchartered.com/com/sc/cashpi/commonpipeline/cashpi-services-benematch-service:cashpi-services-benematch-service-0.0.1\_releasev1.0.0.3239.08311.build.15"

},

"generation": null,

"importPolicy": {},

"referencePolicy": {"type": ""}

}]

},

"status": null

}

Route.Json

{

"apiVersion": "v1",

"kind": "Route",

"metadata": {

"annotations": {

"haproxy.router.openshift.io/timeout": "10m",

"openshift.io/generated-by": "OpenShiftWebConsole",

"openshift.io/host.generated": "true"

},

"creationTimestamp": "2017-11-02T11:04:34Z",

"labels": {

"app": "cashpi-services-benematch-service",

"application": "cashpi-services-benematch-service"

},

"name": "cashpi-services-benematch-service",

"namespace": "cashpi-dr"

},

"spec": {

"host": "cashpi-services-benematch-service-cashpi-dr.apps.cashpi-dr.ocp.standardchartered.com",

"to": {

"kind": "Service",

"name": "cashpi-services-benematch-service",

"weight": 100

},

"wildcardPolicy": "None",

"tls": null,

"port": {"targetPort": "http"}

},

"status": null

}

Microservice Key Features