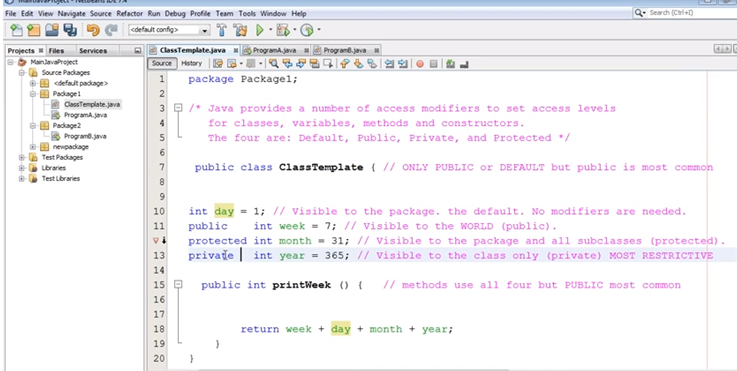
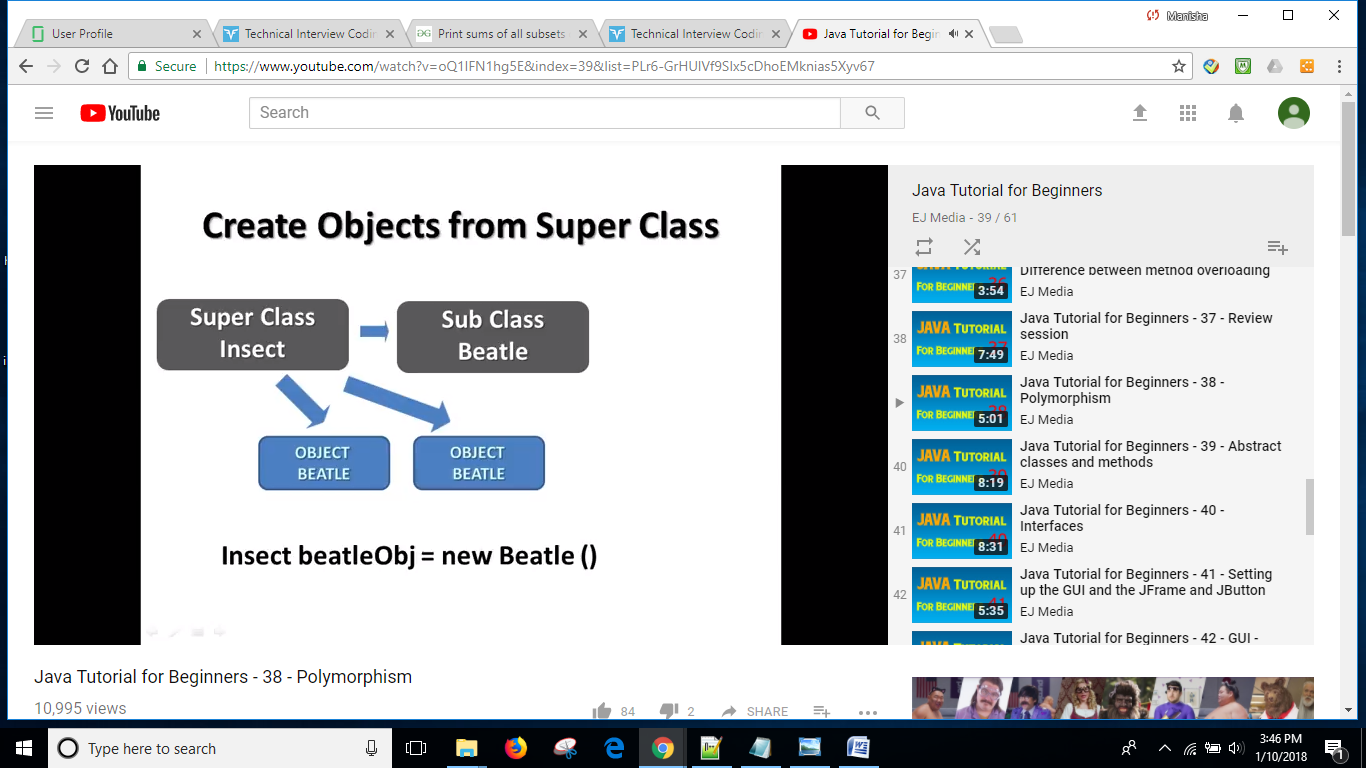
1. **Access modifiers:**



1. Polymorphism:

Reference type is super class and the object created is from subclass. This is polymorphism.



Note: static methods, fields in super class cannot be overridden. But they are hidden.

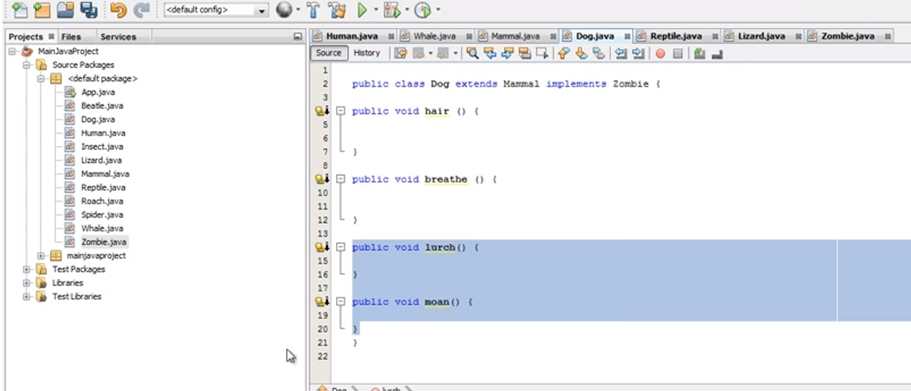
Refer:

<https://www.programcreek.com/2012/11/java-field-overriding/>

<https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/>

1. **Interface:**

Multiple inheritance. “extend” for inheriting from Class. “implement” for inheriting from Interface.





Example for parent class object = child class.

**package** interview;

**class** Animal {

**int** a;

**public** **void** move() {

System.***out***.println("Animals can move");

}

**public** **void** printa() {

System.***out***.println("Animals can print a: "+a);

}

}

**class** Dog **extends** Animal {

**int** a;

**public** **void** move() {

System.***out***.println("Dogs can walk and run");

}

**public** **void** bark() {

System.***out***.println("Dogs can bark");

}

}

**public** **class** practice {

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

Animal a = **new** Animal(); // Animal reference and object

Animal b = **new** Dog(); // Animal reference but Dog object

a.move(); // runs the method in Animal class

b.move(); // runs the method in Dog class

//b.bark(); //compile time error. Does not compile as Animal does not have 'bark' method

((Dog)b).bark(); //changed type of 'b' to Dog. This works. Bad design though.

Dog d = **new** Dog();

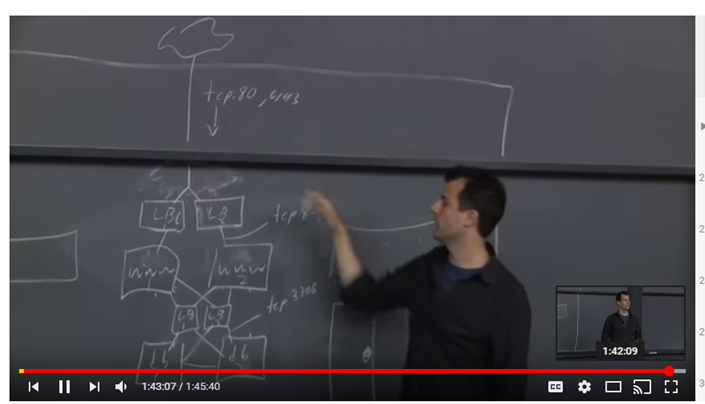
d.a=90; // sets variable 'a' in Dog class

d.printa(); //print 0 as variable 'a' in Animal class is not set.

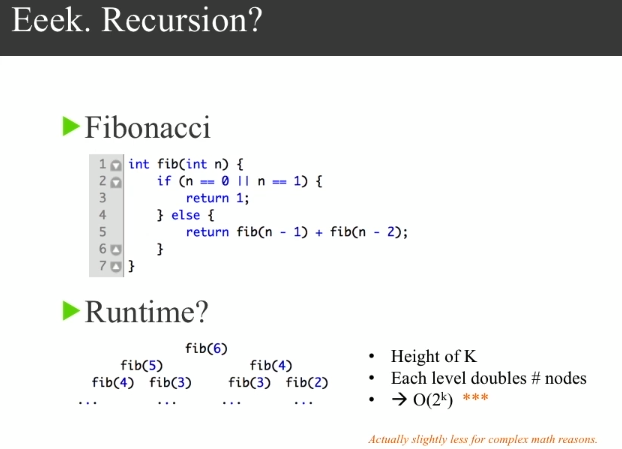
}

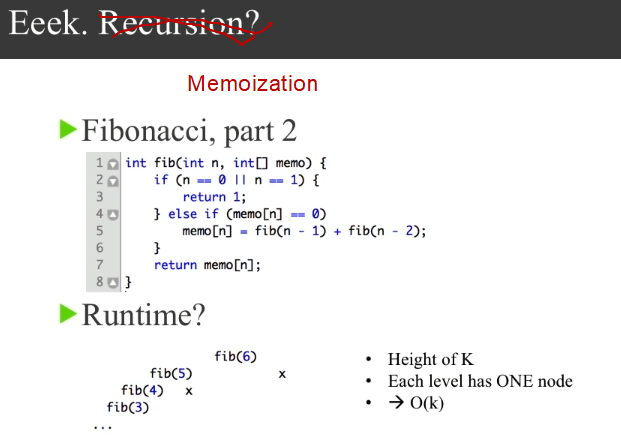
}

1. Scalability, Data center, how cloud goes to application level and database level



1. Big O for Fibonacci





1. Deadlock

Process1 is holding Resource1 and waiting for Resource2 which is held by Process2 that is waiting for Resource1. If following 4 constraints happens together, then the deadlock occurs.

* Mutual Exclusion: If there is resource that are not sharable.
* No Preemption: Resource cannot be taken from Process unless Process releases it.
* Circular Wait: Set of Processes are waiting for each other in circular form.
* Hold and Wait: A Process is holding at least one Resource and waiting for other Resources.

1. Strings are immutable. Different ways of creating string difference.

String s = new string(“ABC”); vs

String s= “ABC”;

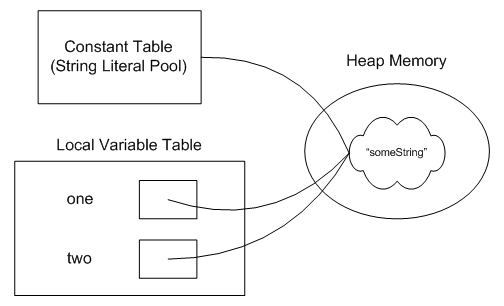
----------------------------------------

**Storage of Strings - The String Literal Pool**

If you've done any preparation for the SCJP exam (and quite possibly even if you haven't), you've probably heard of the "String Literal Pool." *What is the String Literal Pool?* Most often, I hear people say that it is a collection of String objects. Although that's close, it's not exactly correct. Really, it's a collection of *references* to String objects. Strings, even though they are immutable, are still objects like any other in Java. Objects are created on the heap and Strings are no exception. So, Strings that are part of the "String Literal Pool" still live on the heap, but they have references to them from the String Literal Pool.  
  
Yeah, so that doesn't really explain what the pool is, or what it's for, does it? Well, because String objects are immutable, it's safe for multiple references to "share" the same String object. Take a look at this example:

|  |
| --- |
| **Source Code** |
| public class ImmutableStrings  {  public static void main(String[] args)  {  String one = "someString";  String two = "someString";    System.out.println(one.equals(two));  System.out.println(one == two);  }  }  // Output  true  true |

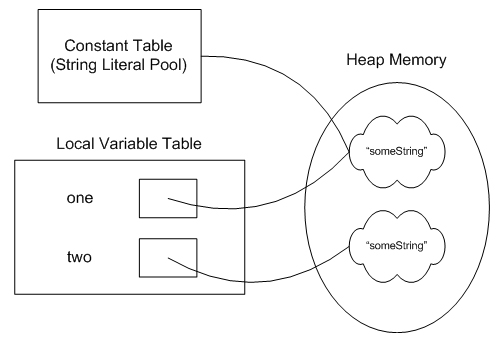
In such a case, there is really no need to make two instances of an identical String object. If a String object could be changed, as a StringBuffer can be changed, we would be forced to create two separate objects. But, as we know that String objects cannot change, we can safely share a String object among the two String references, one and two. This is done through the String literal pool. Here's how it is accomplished:  
  
When a .java file is compiled into a .class file, any String literals are noted in a special way, just as all constants are. When a class is [loaded](http://docs.oracle.com/javase/specs/jls/se8/html/jls-12.html#jls-12.2) (note that loading happens prior to initialization), the JVM goes through the code for the class and looks for String literals. When it finds one, it checks to see if an equivalent String is already referenced from the heap. If not, it creates a String instance on the heap and stores a reference to that object in the constant table. Once a reference is made to that String object, any references to that String literal throughout your program are simply replaced with the reference to the object referenced from the String Literal Pool.  
  
So, in the example shown above, there would be only one entry in the String Literal Pool, which would refer to a String object that contained the word "someString". Both of the local variables, one and two, would be assigned a reference to that single String object. You can see that this is true by looking at the output of the above program. While the equals() method checks to see if the String objects contain the same data ("someString"), the == operator, when used on objects, checks for referential equality - that means that it will return true if and only if the two reference variables refer to the *exact same object*. In such a case, the references are equal. From the above output, you can see that the local variables, one and two, not only refer to Strings that contain the same data, they refer to the same object.  
  
Graphically, our objects and references would look something like this:



Note, however, that this is a special behavior for *String Literals*. Constructing Strings using the "new" keyword implies a different sort of behavior. Let's look at an example:

|  |
| --- |
| **Source Code** |
| public class ImmutableStrings  {  public static void main(String[] args)  {  String one = "someString";  String two = new String("someString");    System.out.println(one.equals(two));  System.out.println(one == two);  }  }  // Output  true  false |

In this case, we actually end up with a slightly different behavior because of the keyword "new." In such a case, references to the two String literals are still put into the constant table (the String Literal Pool), but, when you come to the keyword "new," the JVM is obliged to create a new String object at run-time, rather than using the one from the constant table.  
  
In such a case, although the two String references refer to String objects that contain the same data, "someString", they *do not* refer to the same object. That can be seen from the output of the program. While the equals() method returns true, the == operator, which checks for referential equality, returns false, indicating that the two variables refer to distinct String objects.  
  
Once again, if you'd like to see this graphically, it would look something like this. Note that the String object referenced from the String Literal Pool is created when the class is loaded while the other String object is created at runtime, when the "new String..." line is executed.



If you'd like to get both of these local variables to refer to the same object, you can use the [intern()](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#intern--) method defined in String. Invoking two.intern() will look for a String object referenced from the String Literal Pool that has the same value as the one you invoked the intern method upon. If one is found, a reference to that String is returned and can be assigned to your local variable. If you did so, you'd have a picture that looks just like the one above, with both local variables, one and two, referring to the same String object, which is also referenced from the String Literal Pool. At that point, the second String object, which was created at run-time, would be eligible for garbage collection.

## AngularJS Applications

AngularJS **modules** define AngularJS applications.

AngularJS **controllers** control AngularJS applications.

The **ng-app** directive defines the application, the **ng-controller** directive defines the controller.

### AngularJS Example

<div ng-app="**myApp**" ng-controller="**myCtrl**">  
  
First Name: <input type="text" ng-model="firstName"><br>  
Last Name: <input type="text" ng-model="lastName"><br>  
<br>  
Full Name: {{firstName + " " + lastName}}  
  
</div>  
  
<script>  
var app = angular.module('**myApp**', []);  
app.controller('**myCtrl**', function($scope) {  
    $scope.firstName= "John";  
    $scope.lastName= "Doe";  
});  
</script>

AngularJS modules define applications:

### AngularJS Module

var app = angular.module('myApp', []);

AngularJS controllers control applications:

### AngularJS Controller

app.controller('myCtrl', function($scope) {  
    $scope.firstName= "John";  
    $scope.lastName= "Doe";  
});

## The Model

The model is responsible for managing application data. It responds to the request from view and to the instructions from controller to update itself.

## The View

A presentation of data in a particular format, triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.

## The Controller

The controller responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

## Nested JSON Objects

Values in a JSON object can be another JSON object.

### Example

myObj = {  
    "name":"John",  
    "age":30,  
    "cars": {  
        "car1":"Ford",  
        "car2":"BMW",  
        "car3":"Fiat"  
    }  
 }

You can access nested JSON objects by using the dot notation or bracket notation:

### Example

x = myObj.cars.car2;  
//or:  
x = myObj.cars["car2"];

## AJAX / JSON / The PHP File

## JSON From the Server

You can request JSON from the server by using an AJAX request

As long as the response from the server is written in JSON format, you can parse the string into a JavaScript object.

### Example

Use the XMLHttpRequest to get data from the server:

var xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
        var myObj = JSON.parse(this.responseText);  
        document.getElementById("demo").innerHTML = myObj.name;  
    }  
};  
xmlhttp.open("GET", "json\_demo.txt", true);  
xmlhttp.send();

// text in file: https://www.w3schools.com/js/json\_demo.txt

{

"name":"John",

"age":31,

"pets":[

{ "animal":"dog", "name":"Fido" },

{ "animal":"cat", "name":"Felix" },

{ "animal":"hamster", "name":"Lightning" }

]

}

-------------------------------------------------------------------

PHP has some built-in functions to handle JSON.

Objects in PHP can be converted into JSON by using the PHP function json\_encode():

### PHP file

<?php  
$myObj->name = "John";  
$myObj->age = 30;  
$myObj->city = "New York";  
  
$myJSON = json\_encode($myObj);  
  
echo $myJSON;  
?>

## The Client JavaScript

Here is a JavaScript on the client, using an AJAX call to request the PHP file from the example above:

### Example

Use JSON.parse() to convert the result into a JavaScript object:

var xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
        var myObj = JSON.parse(this.responseText);  
        document.getElementById("demo").innerHTML = myObj.name;  
    }  
};  
xmlhttp.open("GET", "demo\_file.php", true);  
xmlhttp.send();

----------------------------------------------------------------

## PHP Database

PHP is a server side programming language, and should be used for operations that can only be performed by a server, like accessing a database.

Imagine you have a database on the server, containing customers, products, and suppliers.

You want to make a request to the server where you ask for the first 10 records in the "customers" table:

### Example

Use JSON.stringify() to convert the JavaScript object into JSON:

obj = { "table":"customers", "limit":10 };  
dbParam = JSON.stringify(obj);  
xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
        document.getElementById("demo").innerHTML = this.responseText;  
    }  
};  
xmlhttp.open("GET", "json\_demo\_db.php?x=" + dbParam, true);  
xmlhttp.send();

### Example explained:

* Define an object containing a table property and a limit property.
* Convert the object into a JSON string.
* Send a request to the PHP file, with the JSON string as a parameter.
* Wait until the request returns with the result (as JSON)
* Display the result received from the PHP file.

Take a look at the PHP file:

### PHP file

<?php  
header("Content-Type: application/json; charset=UTF-8");  
$obj = json\_decode($\_GET["x"], false);  
  
$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");  
$result = $conn->query("SELECT name FROM ".$obj->table." LIMIT ".$obj->limit);  
$outp = array();  
$outp = $result->fetch\_all(MYSQLI\_ASSOC);  
  
echo json\_encode($outp);  
?>

### PHP File explained:

* Convert the request into an object, using the PHP function json\_decode().
* Access the database, and fill an array with the requested data.
* Add the array to an object, and return the object as JSON using the json\_encode() function.

-------------------------------------------------------------------------------

## PHP Method = POST

When sending data to the server, it is often best to use the HTTP POST method.

To send AJAX requests using the POST method, specify the method, and the correct header.

The data sent to the server must now be an argument to the .send() method:

### Example

obj = { "table":"customers", "limit":10 };  
dbParam = JSON.stringify(obj);  
xmlhttp = new XMLHttpRequest();  
xmlhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {  
        myObj = JSON.parse(this.responseText);  
        for (x in myObj) {  
            txt += myObj[x].name + "<br>";  
        }  
        document.getElementById("demo").innerHTML = txt;  
    }  
};  
**xmlhttp.open("POST", "json\_demo\_db\_post.php", true);  
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
xmlhttp.send("x=" + dbParam);**

The only difference in the PHP file is the method for getting the transferred data.

### PHP file

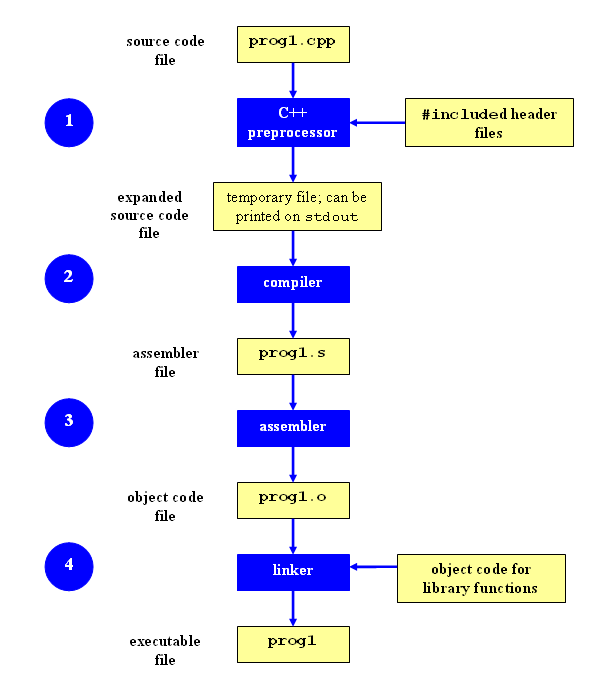
Use $\_POST instead of $\_GET:

<?php  
header("Content-Type: application/json; charset=UTF-8");  
**$obj = json\_decode($\_POST["x"], false);**  
$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");  
$result = $conn->query("SELECT name FROM ".$obj->table." LIMIT ".$obj->limit);  
$outp = array();  
$outp = $result->fetch\_all(MYSQLI\_ASSOC);  
  
echo json\_encode($outp);  
?>

1. **C++ compilation process vs Java compilation process**

In C++, the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.



In Java, **Compiling the Program**

A program has to be converted to a form the Java VM can understand so any computer with a Java VM can interpret and run the program. Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the Java VM.

**Interpreting and Running the Program**

Once your program successfully compiles into Java bytecodes, you can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

1. How does Garbage Collection work?

<https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/>

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. In order 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.

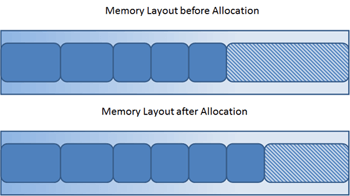


Figure 2.1: New objects are simply allocated at the end of the used heap.

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](http://www.wikipedia.com/osgi) 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.

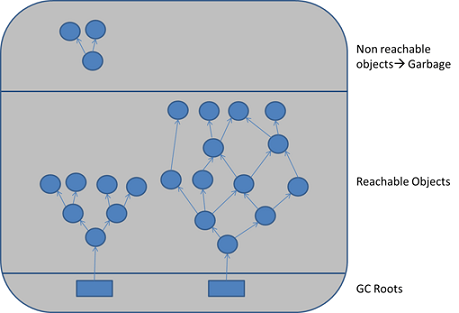


Figure 2.2: GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected.

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](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)). As you might intuit, it’s a straightforward, two-step process:

1. The algorithm traverses all object references, starting with the GC roots, and marks every object found as alive.
2. 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.

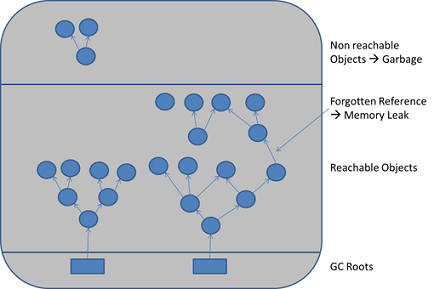


Figure 2.3: When objects are no longer referenced directly or indirectly by a GC root, they will be removed. There are no classic memory leaks. Analysis cannot really identify memory leaks; it can only point out suspicious objects.

1. **Is Java Functional programming language?**

Java is Imperative programming language. It can have external global variable that can be updated.

int total =0;

for(int i=0;i<=10;i++){

total=total+i; //total is outside for loop and is updated from inside this loop.

}

Haskell is functional programming language.

<https://www.youtube.com/watch?v=sqV3pL5x8PI>

1. Spring:

Interview questions: https://www.edureka.co/blog/interview-questions/spring-interview-questions/

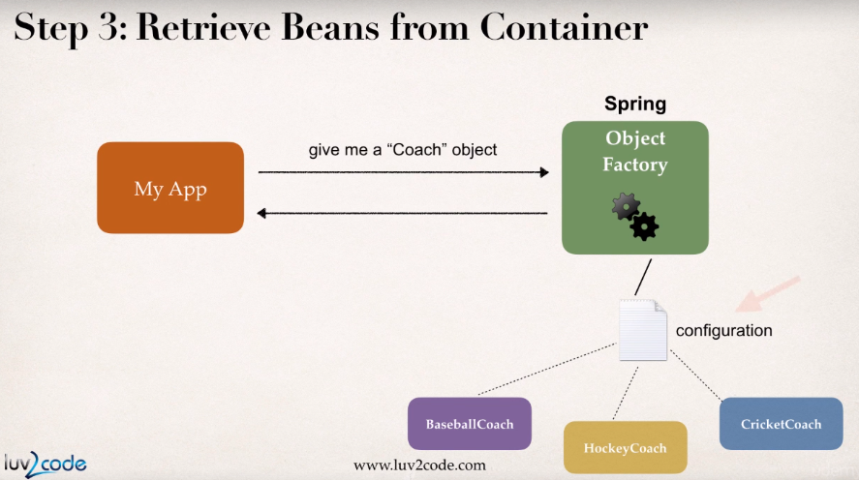
**Primary Function:**

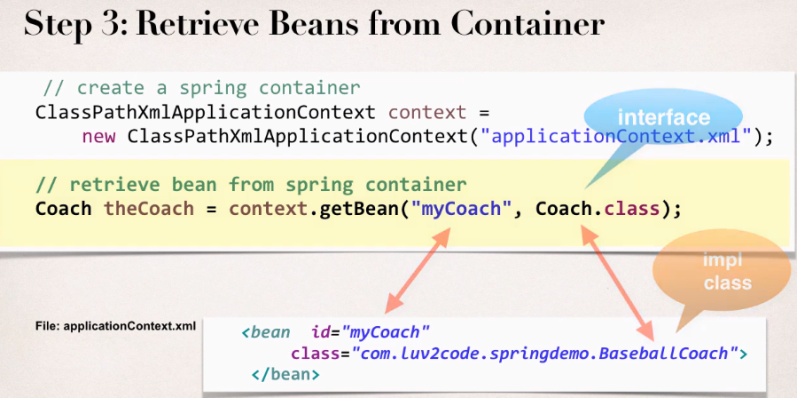
* Outsourcing the creation and management of objects to an object factory (Inversion of Control)
* Inject object’s dependencies (Dependency Injection)

**Types of Injection**

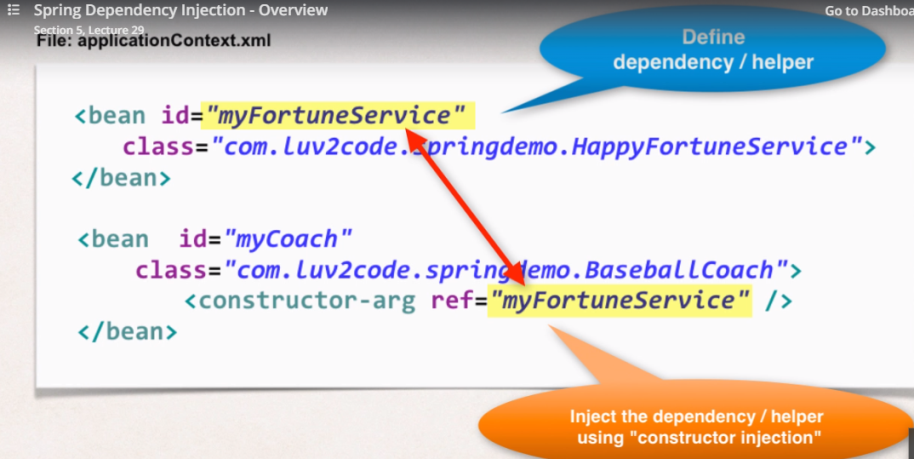
* Constructor injection
* Setter injection
* Any Method injection
* Field Injection

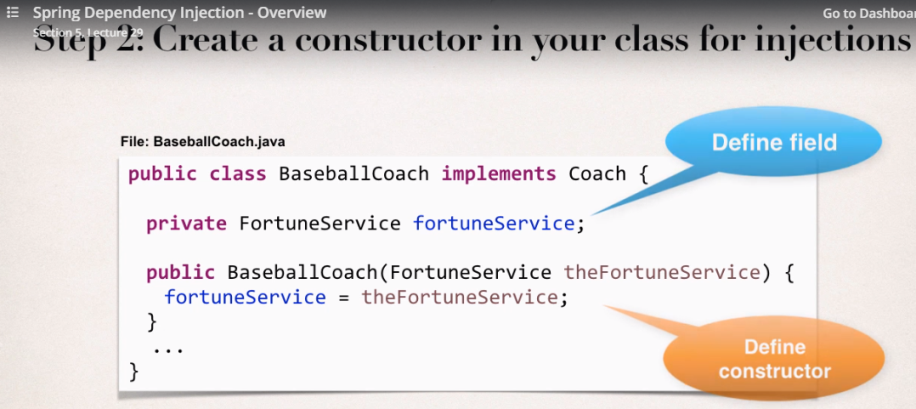
**IoC (Inversion of Control):**



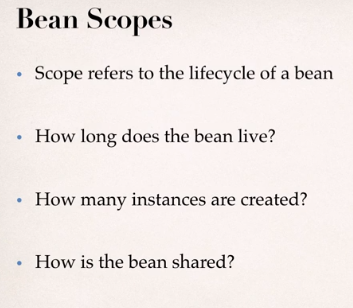


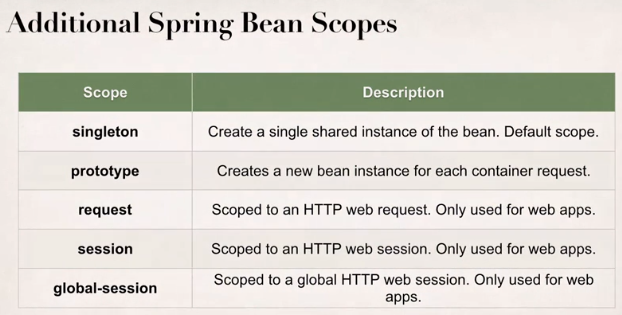
**Dependency Injection:** In this example, Spring will create the object(for BaseballCoach) and inject the dependency(object for HappyFortuneService) as an argument to constructor.

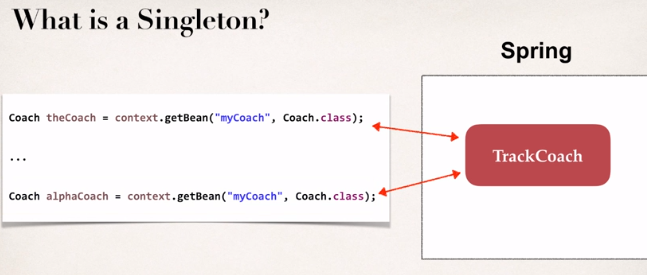
****

****

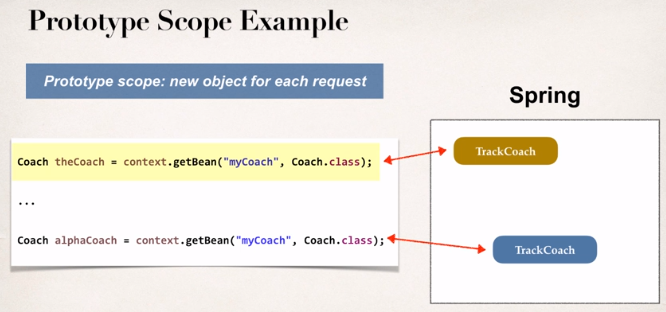
1. Spring Bean Scope



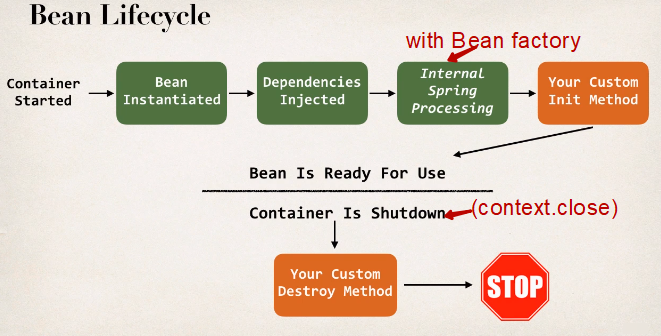








1. Bean lifecycle



**What is Spring MVC?**

- Framework for building web applications in Java

- Based on Model-View\_Controller design pattern

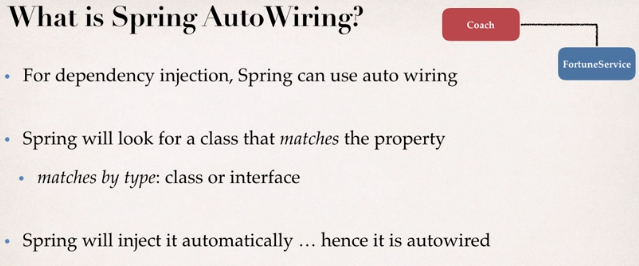
- Use features of the Core Spring Framework(IoC, DI)

Spring **Model**: It is a container of your application data. In controller, you can put anything in the model(string, objects, info from database etc). The view page (JSP) can access data from the model.

1. Some Annotations:

@Component: Spring does component scan for your Java Beans. Register the Spring Bean automatically.

@Autowired: For dependency injection, Spring can automatically wire up the objects together.



@Qualifier: If there is more than one bean for Autowire, we have to specify the bean id to tell Spring which bean to use. This is done via @Qualifier (@Qualifier(*Beanid*))

@Scope: For defining bean scope(singleton, prototype etc) instead of writing in XML configuration.

@PostConstruct: Used in Bean lifecycle. Code annotated by this will execute after constructor and after injection of dependencies. In this code/method, you can write your custom initialization work.

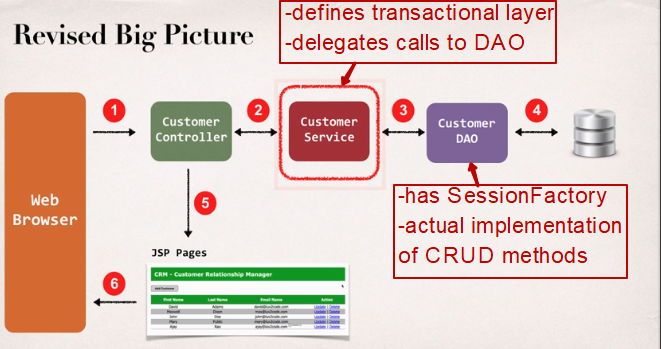
@PreDestroy: Used in Bean lifecycle. Code annotated by this will execute before bean is destroyed. In this method, you can write your custom cleanup code.

@Transactional: Automatically begin and end a transaction for your Hibernate code. No need for you to explicitly do this in your code. (session.beginTransaction(), session.getTransaction().commit() are not needed).

@Repository: Placed in DAO implementation class. It is a subclass annotation of @Component. So the DAO implementation is available for component scanning. Also, provides translation of any JDBC related exceptions

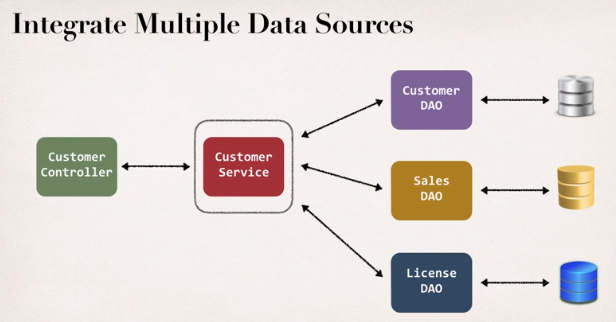
@Service: Applied to Service implementation class. It is a subclass annotation of @Component. Spring will automatically register the Service implementation by using component scanning.

1. Spring MVC web application big picture



Why “Customer Service” layer and “Customer DAO” layer needed?

In future, we can have different data and they may be coming from different DAO as shown below.



1. Dispatcher Servlet in Spring:

In Spring MVC all incoming requests go through a single servlet. This servlet - DispatcherServlet - is the front controller. Front controller is a typical design pattern in the web applications development. In this case, a single servlet receives all requests and transfers them to to all other components of the application.

The task of the DispatcherServlet is to send request to the specific Spring MVC controller.

1. Hibernate:

* A framework for saving Java objects in database.
* Provides Object-to-Relational Mapping (ORM)

SessionFactory:

* Reads the hibernate config file.
* Creates Session objects.
* Heavy-weight object.
* Only create once in your app

Session:

* Wraps a JDBC connection
* Main object used to save/retrieve objects
* Short-lived object
* Retrieved from SessionFactory

1. **Implement Singleton pattern**

**Method 3: Eager Instantiation**

|  |
| --- |
| // Static initializer based Java implementation of  // singleton design pattern  class Singleton  {      private static Singleton obj = new Singleton();       private Singleton() {}       public static Singleton getInstance()      {          return obj;      }  } |

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

**Method 4 (Best): Use “**[**Double Checked Locking**](https://en.wikipedia.org/wiki/Double-checked_locking)**”**

If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.  
So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

|  |
| --- |
| // Double Checked Locking based Java implementation of  // singleton design pattern  class Singleton  {      private volatile static Singleton obj;       private Singleton() {}       public static Singleton getInstance()      {          if (obj == null)          {              // To make thread safe              synchronized (Singleton.class)              {                  // check again as multiple threads                  // can reach above step                  if (obj==null)                      obj = new Singleton();              }          }          return obj;      }  } |

We have declared the obj [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

1. Implement Factory Pattern

In Factory Pattern, we create object without exposing creation logic to client.

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shapeobject. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

## Step 1

Create an interface.

*Shape.java*

public interface Shape {

void draw();

}

## Step 2

Create concrete classes implementing the same interface.

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

*Square.java*

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Inside Circle::draw() method.");

}

}

## Step 3

Create a Factory to generate object of concrete class based on given information.

*ShapeFactory.java*

public class ShapeFactory {

//use getShape method to get object of type shape

public Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

## Step 4

Use the Factory to get object of concrete class by passing an information such as type.

*FactoryPatternDemo.java*

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory shapeFactory = new ShapeFactory();

//get an object of Circle and call its draw method.

Shape shape1 = shapeFactory.getShape("CIRCLE");

//call draw method of Circle

shape1.draw();

//get an object of Rectangle and call its draw method.

Shape shape2 = shapeFactory.getShape("RECTANGLE");

//call draw method of Rectangle

shape2.draw();

//get an object of Square and call its draw method.

Shape shape3 = shapeFactory.getShape("SQUARE");

//call draw method of square

shape3.draw();

}

}

## Step 5

Verify the output.

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

1. Javascript Closure explained

**We're building a web game where everybody wins and we are all friends forever.**

It's simple—you click on one of three boxes to see what nice thing you've won. You always win something nice. Because we love you.

Here's what we have so far. Something's going wrong though. Can you tell what it is?

<button id="btn-0">Button 1!</button>

<button id="btn-1">Button 2!</button>

<button id="btn-2">Button 3!</button>

<script type="text/javascript">

var prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];

for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

// for each of our buttons, when the user clicks it...

document.getElementById('btn-' + btnNum).onclick = function() {

// tell her what she's won!

alert(prizes[btnNum]);

};

}

</script>

HTML

The syntax is just fine—the problem is some unexpected *behavior*.

**Gotchas**

Coding style choices aside, what we found is a problem in *behavior*.

**Solution**

The user's prize is *always* undefined!

**The Problem**

The anonymous method we're assigning to the buttons' onclicks has access to variables in the scope outside of it (this is called a closure ↴ ). In this case, it has access to btnNum.

**When a method accesses a variable outside its scope, it accesses *that variable*, not a frozen copy**. So when the value held by the variable changes, the method gets that new value. By the time the user starts pressing buttons, our loop will have already completed and btnNum will be 3, so this is what each of our anonymous methods will get for btnNum!

**Why 3?** The for loop will increment btnNum until the conditional in the middle is no longer met—that is, until it's not true that btnNum < prizes.length. So the code in the for loop won't run with btnNum = 3, but btnNum will be 3 when the loop is done.

**Why undefined?** prizes has 3 elements, but they are at indices 0,1,2. Array indices start at 0, remember? (Write this down—forgetting this is an easy way to create an off-by-one error in a whiteboard interview.) In JavaScript, accessing a nonexistent index in an array returns undefined(Python throws an IndexError, but Ruby returns nil).

**The Solution**

We can solve this by wrapping our anonymous method in *another anonymous method* that takes btnNum as an argument. Like so:

<button id="btn-0">Button 1!</button>

<button id="btn-1">Button 2!</button>

<button id="btn-2">Button 3!</button>

<script type="text/javascript">

var prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];

for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

// for each of our buttons, when the user clicks it...

document.getElementById('btn-' + btnNum).onclick = function(frozenBtnNum){

return function() {

// tell her what she's won!

alert(prizes[frozenBtnNum]);

};

}(btnNum); // LOOK! We're passing btnNum to our anonymous function here!

}

</script>

HTML

This "freezes" the value of btnNum. Why? Well...

**Primitives vs. Objects**

btnNum is a number, which is a **primitive** type in JavaScript.

Primitives are "simple" data types (string, number, boolean, null, and undefined in JavaScript). Everything else is an *object* in JavaScript (methods, arrays, Date() values, etc).

**Arguments Passed by Value vs. Arguments Passed by Reference**

One important property of primitives in JS is that when they are passed as arguments to a method, they are *copied* ("passed by value"). So for example:

Heads up: This is *not* well-formed JavaScript. We're using it to prove a point.

var threatLevel = 1;

function inspireFear(threatLevel){

threatLevel += 100;

}

inspireFear(threatLevel);

console.log(threatLevel); // Whoops! It's still 1!

JavaScript

The threatLevel inside inspireFear() is a *new* number, initialized to the same *value* as the threatLevel outside of inspireFear(). Giving these *different* variables the same name might cause confusion here. If we change the two variables to have different names we get the exact same behavior:

var threatLevel = 1;

function inspireFear(theThreatLevel){

theThreatLevel += 100;

}

inspireFear(threatLevel);

console.log(threatLevel); // Whoops! It's still 1!

JavaScript

In contrast, **when a method takes an object, it actually takes a *reference* to *that very object***. So changes you make to the object in the method persist after the method is done running. This is sometimes called a **side effect**.

var scaryThings = ['spiders', 'Cruella de Vil'];

function inspireFear(scaryThings){

scaryThings.push('nobody ever using Interview Cake');

scaryThings.push('i should have gotten a real job');

scaryThings.push('why am i doing this to myself');

}

inspireFear(scaryThings);

console.log(scaryThings);

// ['spiders', 'Cruella de Vil', 'nobody ever using Interview Cake', 'i should have gotten a real job', 'why am i doing this to myself']

JavaScript

**Bringing it home**

Back to our solution:

<button id="btn-0">Button 1!</button>

<button id="btn-1">Button 2!</button>

<button id="btn-2">Button 3!</button>

<script type="text/javascript">

var prizes = ['A Unicorn!', 'A Hug!', 'Fresh Laundry!'];

for (var btnNum = 0; btnNum < prizes.length; btnNum++) {

// for each of our buttons, when the user clicks it...

document.getElementById('btn-' + btnNum).onclick = function(frozenBtnNum){

return function() {

// tell her what she's won!

alert(prizes[frozenBtnNum]);

};

}(btnNum);

}

</script>

HTML

So when we pass btnNum to the outer anonymous method as its one argument, we create a *new*number inside the outer anonymous method called frozenBtnNum that has the value that btnNumhad *at that moment* (0, 1, or 2).

Our inner anonymous method is still a closure because it still reaches outside its scope, but now it closes over this *new* number called frozenBtnNum, whose value will not change as we iterate through our for loop.

**What We Learned**

Like several common Javascript interview questions, this question hinges on a solid understanding of closures and *pass by reference* vs *pass by value*. If you're shaky on either of those, look back at the examples in the solution.

1. Parallellism

Threads and processes: A computer will often appear to be doing many things simultaneously, such as  checking for new e‐mail messages, saving a Word document, and loading a website.   Each program is a separate "process".  Each process has one or more "threads."  If a  process has several threads, they appear to run simultaneously.  For example, an e‐ mail client may have one thread that checks for new e‐mail messages and one  thread for the GUI so that it can show a button being pressed.  In fact, only one  thread is being run at any given time.  The processor switches between threads so  quickly that they appear to be running simultaneously. Multiple threads in a single process have access to the same memory.  By contrast,  multiple processes have separate regions of memory and can only communicate by  special mechanisms.  The processor loads and saves a separate set of registers for  each thread. Remember, each process has one or more threads, and the processor switches  between threads.

Mutexes and semaphores: A mutex is like a lock.  Mutexes are used in parallel programming to ensure that only  one thread can access a shared resource at a time.  For example, say one thread is  modifying an array.  When it has gotten halfway through the array, the processor  switches to another thread.  If we were not using mutexes, the thread might try to  modify the array as well, which is probably not what we want. To prevent this, we could use a mutex.  Conceptually, a mutex is an integer that  starts at 1.  Whenever a thread needs to alter the array, it "locks" the mutex.  This  causes the thread to wait until the number is positive and then decreases it by one.   When the thread is done modifying the array, it "unlocks" the mutex, causing the  number to increase by 1.  If we are sure to lock the mutex before modifying the  array and to unlock it when we are done, then we know that no two threads will  modify the array at the same time. Semaphores are more general than mutexes.  They differ only in that a semaphore's  integer may start at a number greater than 1.  The number at which a semaphore  starts is the number of threads that may access the resource at once.  Semaphores  support "wait" and "signal" operations, which are analogous to the "lock" and  "unlock" operations of mutexes.

Synchronized methods (in Java): Another favorite question of interviewers is, "What is a synchronized method in  Java?"  Each object in Java has its own mutex.  Whenever a synchronized method is  called, the mutex is locked.  When the method is finished, the mutex is unlocked.   This ensures that only one synchronized method is called at a time on a given object.

Deadlock: Deadlock is a problem that sometimes arises in parallel programming.  It is typified  by the following, which is supposedly a law that came before the Kansas legislature: "When two trains approach each other at a crossing, both shall come to a full stop  and neither shall start up again until the other has gone." Strange as this sounds, a similar situation can occur when using mutexes.  Say we  have two threads running the following code:

Thread 1: acquire(lock1); acquire(lock2); [do stuff] release(lock1); release(lock2);

Thread 2: acquire(lock2); acquire(lock1); [do stuff] release(lock2); release(lock1);

Suppose that thread 1 is executed to just after the first statement.  Then, the  processor switches to thread 2 and executes both statements.  Then, the processor  switches back to thread 1 and executes the second statement.  In this situation,  thread 1 will be waiting for thread 2 to release lock1, and thread 2 will be waiting  for thread 1 to release lock2.  Both threads will be stuck indefinitely.  This is called  deadlock.

How can we ensure that deadlock does not occur? Answer: There are many possible answers to this problem, but the answer the  interviewer will be looking for is this: we can prevent deadlock if we assign an order  to our locks and require that locks always be acquired in order.  For example, if a  thread needs to acquire locks 1, 5, and 2, it must acquire lock 1, followed by lock 2,  followed by lock 5.  That way we prevent one thread trying to acquire lock 1 then  lock 2, and another thread trying to acquire lock 2 then lock 1, which could cause  deadlock.  (Note that this approach is not used very often in practice.)

1. REST: REpresentational State Transfer

This is an architectural style. The web services that use this architecture are called Restful web services. This architectural style has 6 constraints to follow. They are:

* Statelessness
* Client-Server separation
* Cacheability
* Layered system
* Uniform interface
* Code on demand(optional)

1. SOAP: Simple Object Access Protocol.

This is a messaging protocol for exchanging structured information in computer networks. It uses XML for messaging format. WSDL(Web Service Definition Language) defines SOAP. WSDL is not a must have for SOAP but it is hard to use web service without WSDL support.

1. Observer pattern vs Pub-Sub Pattern

https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

**Observer Design Pattern:**

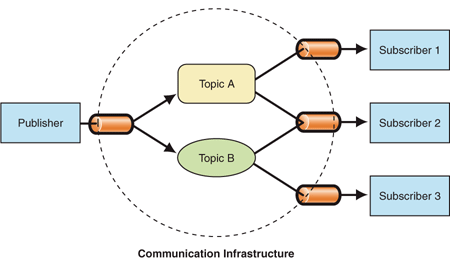
Let’s assume that you are searching for job as a software engineer and very interested in a company named ‘Banana Inc.’. So, you contacted with their Hiring Manager and gave him your contact number. He ensured you that if there is any vacancy they will let you know. And there are several other candidate interested too, like you. They will let all of the candidates know about the vacancy and maybe if you response then they will conduct interview. So, how is this scenario related to ‘Observer’ design pattern? Here, the company ‘Banana Inc.’ is the **Subject**which is maintaining a list of all the **Observers**(candidates like you) and will **notify**the observers for a certain **event**‘vacancy’. Ain’t it easy, mate?



**Pub-Sub(Publisher-Subscriber) Design Pattern:**

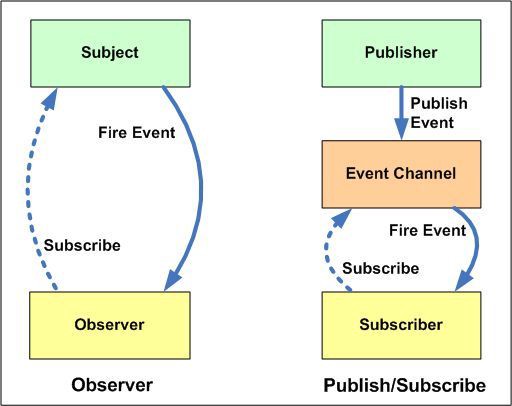
Yes, the **Subject** in the Observer Pattern is like a **Publisher**and the **Observer**can totally be related to a **Subscriber**and Yes, the Subject notify the Observers like how a Publisher generally notify his subscribers. That’s why most of the Design Pattern books or articles use ‘Publisher-Subscriber’ notion to explain Observer Design Pattern. But there is another popular Pattern called **‘Publisher-Subscriber’** and it is conceptually very similar to the Observer pattern. The major difference between the (real) **‘Publisher-Subscriber’** pattern and **‘Observer’** pattern is this:

This means that the publisher and subscriber don’t know about the existence of one another. There is a third component, called **broker or message broker or event bus**, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly. In other words, pub-sub is a pattern used to communicate messages between different system components without these components knowing anything about each other’s identity. how does the broker filter all the messages? Actually there are several processes for message filtering. Most popular methods are: Topic basedand Content based.





Let’s list out the differences as a quick Summary:



* In the **Observer** pattern, the O***bservers are aware of the Subject, also the Subject maintains a record of the Observers***. Whereas, in **Publisher/Subscriber**, publishers and subscribers ***don’t need to know each other***. They simply communicate with the help of message queues or broker.
* In **Publisher/Subscriber** pattern, components are loosely coupled as opposed to **Observer**pattern.
* **Observer** pattern is mostly implemented in a ***synchronous*** way, i.e. the Subject calls the appropriate method of all its observers when some event occurs. The **Publisher/Subscriber** pattern is mostly implemented in an ***asynchronous*** way (using message queue).
* **Observer**pattern need to be implemented in a single application address space. On the other hand, **Publisher/Subscriber** pattern is more of a cross application pattern.

Despite of the differences between these patterns, some might say that **Publisher-Subscriber** pattern is a variation of **Observer** pattern because of the conceptual similarity between them. And it won’t be wrong at all. Don’t need to take the differences religiously.

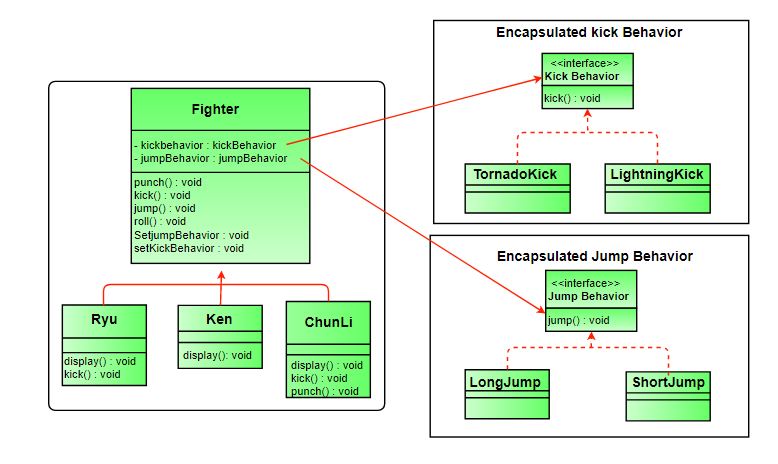
1. Strategy pattern:

https://www.geeksforgeeks.org/strategy-pattern-set-1/

<https://www.geeksforgeeks.org/strategy-pattern-set-2/>

This is a design pattern that contains family of algorithm and individual algorithm can be selected at runtime.

It relies on composition instead of inheritance for code reuse.



**Advantages:**

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages:**

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.
4. SQl vs NoSQL

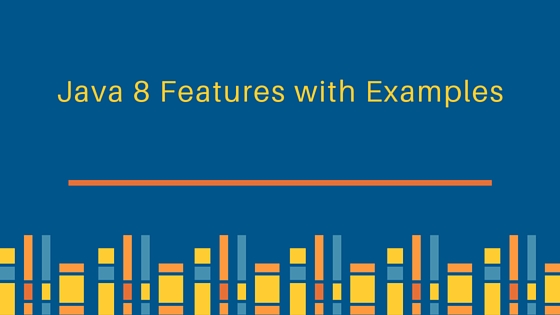
|  |  |
| --- | --- |
| Sql | NoSql |
| Structure: Table based | Key-value based, document based, wide-column based, graph based |
| Ex: MySql, Oracle, PostgreSQL, Microsoft SQL Server | Ex: MongoDB, CouchDB, Big Table, Redis, Cassandra |
| Better for: applications that require multi-row transactions like accounting system, inventory system. | Better for: if schema continues to change, real-time analytics, content management system |

# Java 8 Features with Examples

APRIL 6, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [70 COMMENTS](https://www.journaldev.com/2389/java-8-features-with-examples#comments)

**Java 8** was released in 18th March 2014, so it’s high time to look into Java 8 Features. In this tutorial, we will look into Java 8 features with examples.

## Java 8 Features

[](https://cdn.journaldev.com/wp-content/uploads/2013/12/java-8-features-with-examples.jpg)

Some of the important Java 8 features are;

1. [forEach() method in Iterable interface](https://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach)
2. [default and static methods in Interfaces](https://www.journaldev.com/2389/java-8-features-with-examples#interface-default-static-method)
3. [Functional Interfaces and Lambda Expressions](https://www.journaldev.com/2389/java-8-features-with-examples#functional-interface-lambdas)
4. [Java Stream API for Bulk Data Operations on Collections](https://www.journaldev.com/2389/java-8-features-with-examples#java-stream-api)
5. [Java Time API](https://www.journaldev.com/2389/java-8-features-with-examples#java8-time)
6. [Collection API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-collection)
7. [Concurrency API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-concurrency)
8. [Java IO improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-io)
9. [Miscellaneous Core API improvements](https://www.journaldev.com/2389/java-8-features-with-examples#java8-core)

Let’s have a brief look on these Java 8 features. I will provide some code snippets for better understanding, so if you want to run programs in Java 8, you will have to setup Java 8 environment by following steps.

* [Download JDK8](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) and install it. Installation is simple like other java versions. JDK installation is required to write, compile and run the program in Java.
* Download latest Eclipse IDE, it provides support for java 8 now. Make sure your projects build path is using Java 8 library.

Learning Java? Nothing better than a video course trusted by over 1,70,000 students (yes, that many students). [Follow this link](https://www.journaldev.com/java-udemy-course) to get heavy discount on the course.

### forEach() method in Iterable interface

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get [ConcurrentModificationException](https://www.journaldev.com/378/java-util-concurrentmodificationexception) if iterator is not used properly.

Java 8 has introduced forEach method in java.lang.Iterable interface so that while writing code we focus on business logic only. forEach method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse. Let’s see forEach usage with simple example.

package com.journaldev.java8.foreach;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.function.Consumer;

import java.lang.Integer;

public class Java8ForEachExample {

public static void main(String[] args) {

//creating sample Collection

List<Integer> myList = new ArrayList<Integer>();

for(int i=0; i<10; i++) myList.add(i);

//traversing using Iterator

Iterator<Integer> it = myList.iterator();

while(it.hasNext()){

Integer i = it.next();

System.out.println("Iterator Value::"+i);

}

//traversing through forEach method of Iterable with anonymous class

myList.forEach(new Consumer<Integer>() {

public void accept(Integer t) {

System.out.println("forEach anonymous class Value::"+t);

}

});

//traversing with Consumer interface implementation

MyConsumer action = new MyConsumer();

myList.forEach(action);

}

}

//Consumer implementation that can be reused

class MyConsumer implements Consumer<Integer>{

public void accept(Integer t) {

System.out.println("Consumer impl Value::"+t);

}

}

1. The number of lines might increase but forEach method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

### default and static methods in Interfaces

If you read forEach method details carefully, you will notice that it’s defined in Iterable interface but we know that interfaces can’t have method body. From Java 8, interfaces are enhanced to have method with implementation. We can use default and static keyword to create interfaces with method implementation. forEach method implementation in Iterable interface is:

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

We know that Java doesn’t provide [multiple inheritance in Classes](https://www.journaldev.com/1775/multiple-inheritance-in-java) because it leads to **Diamond Problem**. So how it will be handled with interfaces now, since interfaces are now similar to abstract classes. The solution is that compiler will throw exception in this scenario and we will have to provide implementation logic in the class implementing the interfaces.

package com.journaldev.java8.defaultmethod;

@FunctionalInterface

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

}

static void print(String str){

System.out.println("Printing "+str);

}

//trying to override Object method gives compile time error as

//"A default method cannot override a method from java.lang.Object"

// default String toString(){

// return "i1";

// }

}

package com.journaldev.java8.defaultmethod;

@FunctionalInterface

public interface Interface2 {

void method2();

default void log(String str){

System.out.println("I2 logging::"+str);

}

}

Notice that both the interfaces have a common method log() with implementation logic.

package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

@Override

public void method2() {

}

@Override

public void method1(String str) {

}

//MyClass won't compile without having it's own log() implementation

@Override

public void log(String str){

System.out.println("MyClass logging::"+str);

Interface1.print("abc");

}

}

As you can see that Interface1 has static method implementation that is used in MyClass.log()method implementation. Java 8 uses **default** and **static** methods heavily in **[Collection API](https://www.journaldev.com/1260/collections-in-java-tutorial)** and default methods are added so that our code remains backward compatible.

If any class in the hierarchy has a method with same signature, then default methods become irrelevant. Since any class implementing an interface already has Object as superclass, if we have equals(), hashCode() default methods in interface, it will become irrelevant. Thats why for better clarity, interfaces are not allowed to have Object class default methods.

For complete details of interface changes in Java 8, please read [Java 8 interface changes](https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method" \o "Java 8 Interface Changes – static methods, default methods, functional Interfaces).

### Functional Interfaces and Lambda Expressions

If you notice above interfaces code, you will notice @FunctionalInterface [annotation](https://www.journaldev.com/721/java-annotations). Functional interfaces are new concept introduced in Java 8. An interface with exactly one abstract method becomes Functional Interface. We don’t need to use @FunctionalInterface annotation to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. You can think of it like [@Override annotation](https://www.journaldev.com/817/java-override-annotation)and it’s best practice to use it. java.lang.Runnable with single abstract method run() is a great example of functional interface.

One of the major benefits of functional interface is the possibility to use **lambda expressions** to instantiate them. We can instantiate an interface with [anonymous class](https://www.journaldev.com/996/java-inner-class) but the code looks bulky.

Runnable r = new Runnable(){

@Override

public void run() {

System.out.println("My Runnable");

}};

Since functional interfaces have only one method, lambda expressions can easily provide the method implementation. We just need to provide method arguments and business logic. For example, we can write above implementation using lambda expression as:

Runnable r1 = () -> {

System.out.println("My Runnable");

};

If you have single statement in method implementation, we don’t need curly braces also. For example above Interface1 anonymous class can be instantiated using lambda as follows:

Interface1 i1 = (s) -> System.out.println(s);

i1.method1("abc");

So lambda expressions are means to create anonymous classes of functional interfaces easily. There are no runtime benefits of using lambda expressions, so I will use it cautiously because I don’t mind writing few extra lines of code.

A new package java.util.function has been added with bunch of functional interfaces to provide target types for lambda expressions and method references. Lambda expressions are a huge topic, I will write a separate article on that in future.

You can read complete tutorial at [Java 8 Lambda Expressions Tutorial](https://www.journaldev.com/2763/java-8-functional-interfaces).

### Java Stream API for Bulk Data Operations on Collections

A new java.util.stream has been added in Java 8 to perform filter/map/reduce like operations with the collection. Stream API will allow sequential as well as parallel execution. This is one of the best feature for me because I work a lot with Collections and usually with Big Data, we need to filter out them based on some conditions.

Collection interface has been extended with stream() and parallelStream() default methods to get the Stream for sequential and parallel execution. Let’s see their usage with simple example.

package com.journaldev.java8.stream;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class StreamExample {

public static void main(String[] args) {

List<Integer> myList = new ArrayList<>();

for(int i=0; i<100; i++) myList.add(i);

//sequential stream

Stream<Integer> sequentialStream = myList.stream();

//parallel stream

Stream<Integer> parallelStream = myList.parallelStream();

//using lambda with Stream API, filter example

Stream<Integer> highNums = parallelStream.filter(p -> p > 90);

//using lambda in forEach

highNums.forEach(p -> System.out.println("High Nums parallel="+p));

Stream<Integer> highNumsSeq = sequentialStream.filter(p -> p > 90);

highNumsSeq.forEach(p -> System.out.println("High Nums sequential="+p));

}

}

If you will run above example code, you will get output like this:

High Nums parallel=91

High Nums parallel=96

High Nums parallel=93

High Nums parallel=98

High Nums parallel=94

High Nums parallel=95

High Nums parallel=97

High Nums parallel=92

High Nums parallel=99

High Nums sequential=91

High Nums sequential=92

High Nums sequential=93

High Nums sequential=94

High Nums sequential=95

High Nums sequential=96

High Nums sequential=97

High Nums sequential=98

High Nums sequential=99

Notice that parallel processing values are not in order, so parallel processing will be very helpful while working with huge collections.  
Covering everything about Stream API is not possible in this post, you can read everything about Stream API at [Java 8 Stream API Example Tutorial](https://www.journaldev.com/2774/java-8-stream).

### Java Time API

It has always been hard to work with Date, Time and Time Zones in java. There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.timepackage that will streamline the process of working with time in java.

Just by looking at Java Time API packages, I can sense that it will be very easy to use. It has some sub-packages java.time.format that provides classes to print and parse dates and times and java.time.zone provides support for time-zones and their rules.

The new Time API prefers enums over integer constants for months and days of the week. One of the useful class is DateTimeFormatter for converting datetime objects to strings.

For complete tutorial, head over to [Java Date Time API Example Tutorial](https://www.journaldev.com/2800/java-8-date-localdate-localdatetime-instant).

### Collection API improvements

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

* + Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
  + Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
  + Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
  + Map replaceAll(), compute(), merge() methods.
  + Performance Improvement for HashMap class with Key Collisions

### Concurrency API improvements

Some important concurrent API enhancements are:

* 1. ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  2. CompletableFuture that may be explicitly completed (setting its value and status).
  3. Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

### Java IO improvements

Some IO improvements known to me are:

* 1. Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
  2. Files.lines(Path path) that reads all lines from a file as a Stream.
  3. Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
  4. BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

### Miscellaneous Core API improvements

Some misc API improvements that might come handy are:

* + [ThreadLocal](https://www.journaldev.com/1076/java-threadlocal-example) static method withInitial(Supplier supplier) to create instance easily.
  + [Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example) interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
  + min(), max() and sum() methods in Integer, Long and Double wrapper classes.
  + logicalAnd(), logicalOr() and logicalXor() methods in Boolean class.
  + [ZipFile](https://www.journaldev.com/957/java-zip-file-folder-example).stream() method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
  + Several utility methods in Math class.
  + jjs command is added to invoke Nashorn Engine.
  + jdeps command is added to analyze class files
  + JDBC-ODBC Bridge has been removed.
  + PermGen memory space has been removed

# Microservice Architecture — Learn, Build, and Deploy Applications

### Get a better understanding of microservice architecture and, as an example of its benefits, how Uber broke down their monolith into microservices.

From my [previous blog](https://www.edureka.co/blog/what-is-microservices/), you must have gotten a basic understanding of microservice architecture. In this blog, you will get into the depth of the architectural concepts and implement them using an Uber case study.

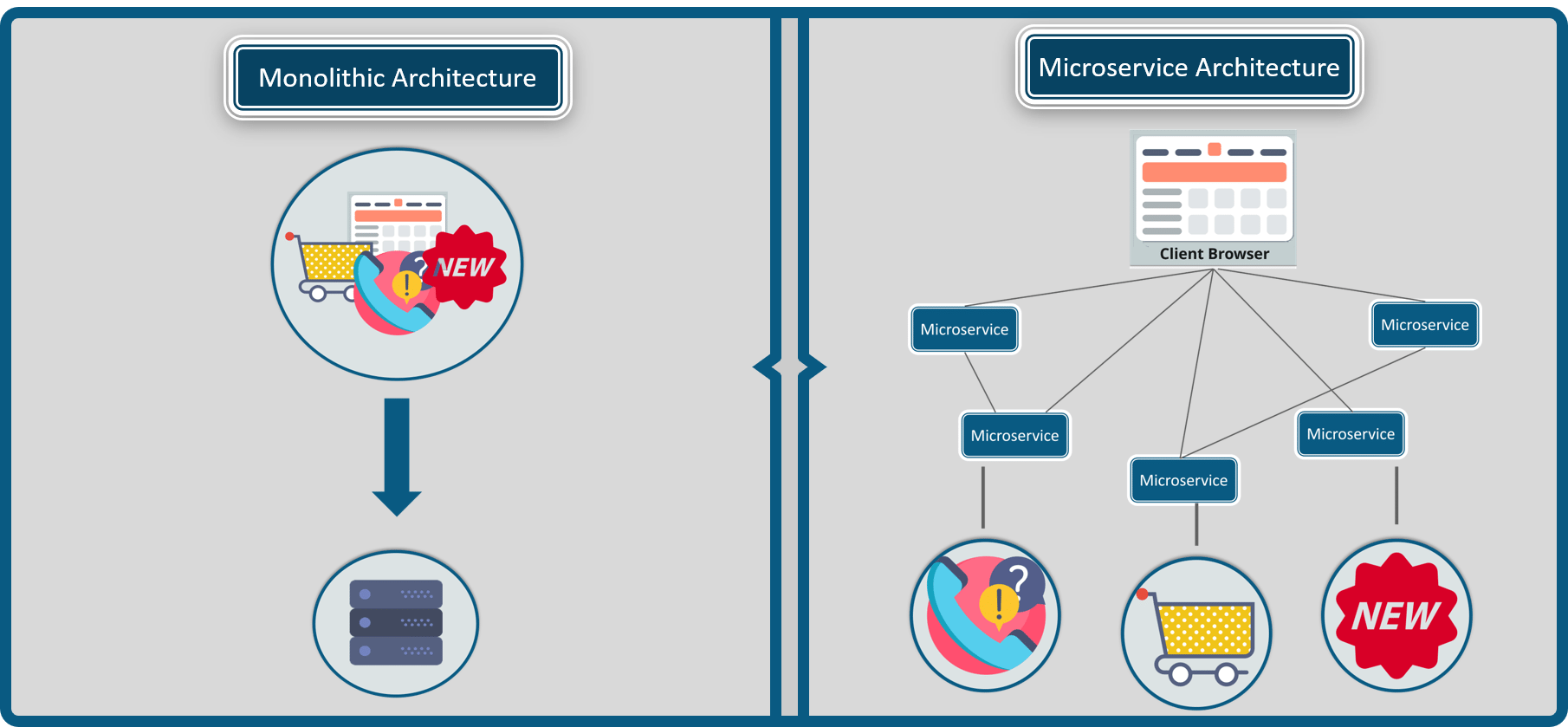
**In this blog, you will learn about the following:**

* Definition of Microservice Architecture
* Key Concepts of Microservice Architecture
* Pros and Cons of Microservice Architecture
* Uber Case Study

You can refer to [*What is Microservices*](https://www.edureka.co/blog/what-is-microservices/) to understand the fundamentals and benefits of microservices.

It will only be fair if I give you the definition of microservices. As such, there is no proper definition of microservices/microservice architecture, but you can say that it is a framework which consists of small, individually deployable services performing different operations.

Microservices focus on a single business domain that can be implemented as fully independent deployable services and implement them on different technology stacks.



**Figure 1:**Difference Between Monolithic and Microservice Architecture - Microservice Architecture.

Refer to the diagram above to understand the difference between monolithic and microservice architecture. For a better understanding of differences between both the architectures, you can refer to my previous blog, [What Is Microservices](https://www.edureka.co/blog/what-is-microservices/).

To make you understand better, let me tell you some key concepts of microservice architecture.

## ****Key Concepts of Microservice Architecture****

Before you start building your own applications using microservices, you need to be clear about the scope and functionalities of your application.

Following are some guidelines to be followed while discussing microservices.

* As a developer, when you decide to build an application separate the domains and be clear with the functionalities.
* Each microservice you design shall concentrate only on one service of the application.
* Ensure that you have designed the application in such a way that each service is individually deployable.
* Make sure that the communication between microservices is done via a stateless server.
* Each service can be furthered refactored into smaller services, having their own microservices.

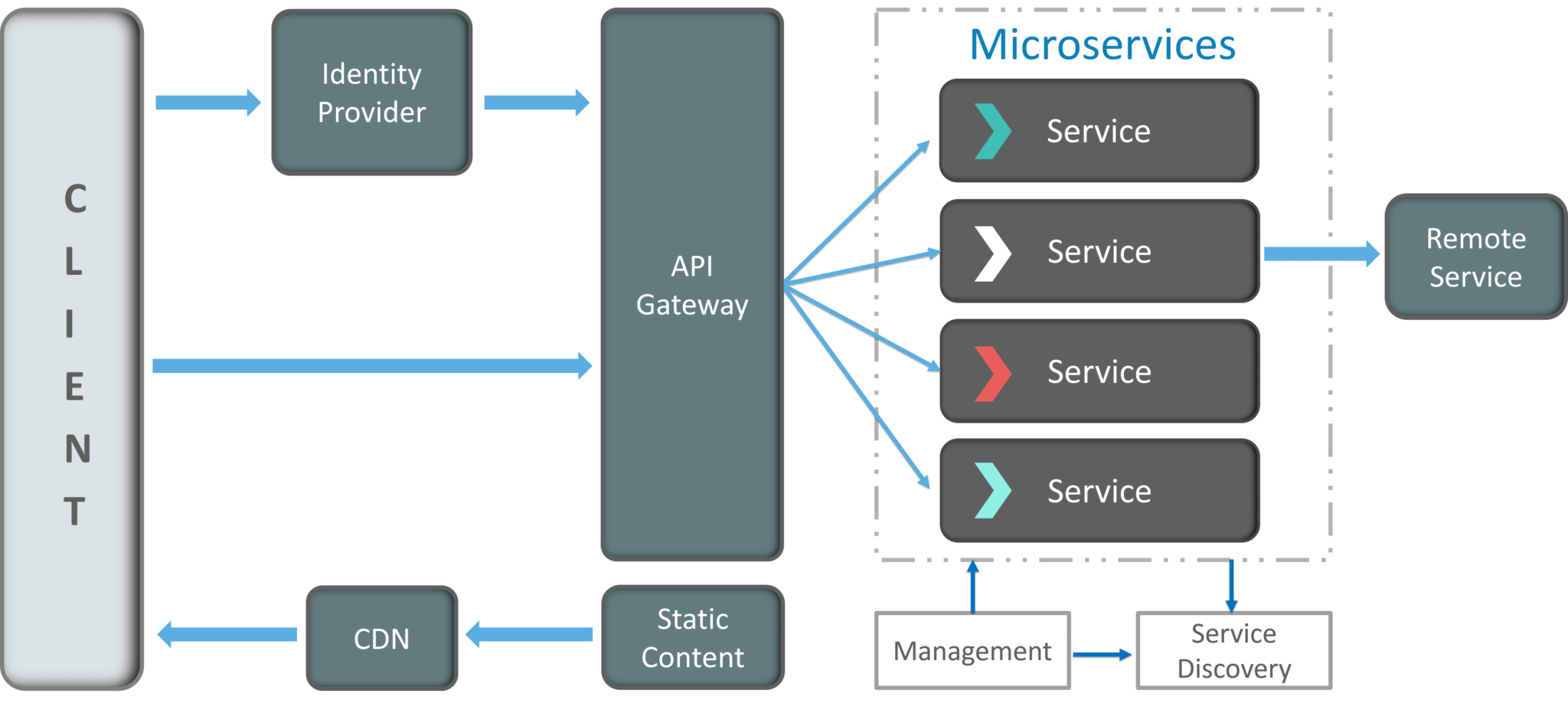
Now, that you have read through the basic guidelines while designing microservices, let's understand the architecture of microservices.

## ****How Does Microservice Architecture Work?****

A typical microservice architecture (MSA) should consist of the following components:

1. Clients
2. Identity Providers
3. API Gateway
4. Messaging Formats
5. Databases
6. Static Content
7. Management
8. Service Discovery

Refer to the diagram below.



**Figure 2:**Architecture Of Microservices - Microservice Architecture.

I know the architecture looks a bit complex, but let me simplify it for you.

### ****1. Clients****

The architecture starts with different types of clients, from different devices trying to perform various management capabilities such as search, build, configure etc.

### ****2. Identity Providers****

These requests from the clients are then passed on the identity providers who authenticate the requests of clients and communicate the requests to API Gateway. The requests are then communicated to the internal services via well-defined API Gateway.

### ****3. API Gateway****

Since clients don’t call the services directly, API Gateway acts as an entry point for the clients to forward requests to appropriate microservices.

**The advantages of using an API gateway include:**

* All the services can be updated without the clients knowing.
* Services can also use messaging protocols that are not web-friendly.
* The API Gateway can perform cross-cutting functions such as providing security, load balancing etc.

After receiving the requests of clients, the internal architecture consists of microservices which communicate with each other through messages to handle client requests.

### ****4. Messaging Formats****

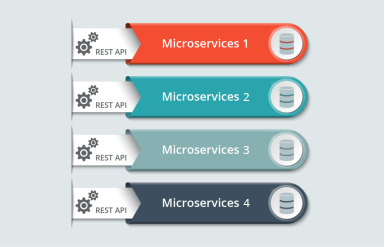
There are two types of messages through which they communicate:

* **Synchronous Messages:** In the situation where clients wait for the responses from a service, microservices usually tend to use **REST (Representational State Transfer)** as it relies on a stateless, client-server, and the **HTTP protocol**. This protocol is used as it is a distributed environment each and every functionality is represented with a resource to carry out operations
* **Asynchronous Messages:** In the situation where clients do not wait for the responses from a service, microservices usually tend to use protocols such as **AMQP, STOMP, MQTT**.These protocols are used in this type of communication since the nature of messages is defined and these messages have to be interoperable between implementations.

The next question that may come to your mind is how do the applications using microservices handle their data?

### ****5. Data Handling****

Well, each microservice owns a private database to capture their data and implement the respective business functionality. Also, the databases of microservices are updated through their service API only. Refer to the diagram below:



**Figure 3:**Representation Of Microservices Handling Data – Microservice Architecture.

The services provided by microservices are carried forward to any remote service which supports inter-process communication for different technology stacks.

### ****6. Static Content****

After the microservices communicate within themselves, they deploy the static content to a cloud-based storage service that can deliver them directly to the clients via **Content Delivery Networks (CDNs)**.

Apart from the above components, there are some other components appear in a typical microservices Architecture:

### ****7. Management****

This component is responsible for balancing the services on nodes and identifying failures.

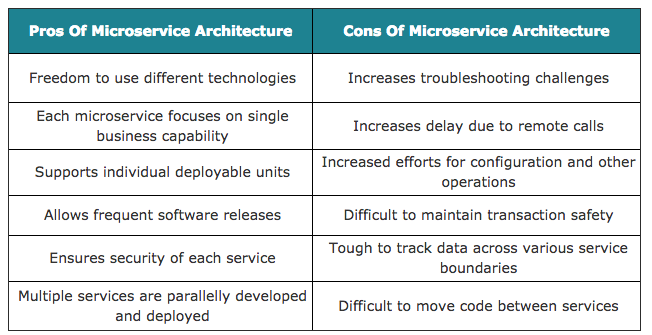
### ****8. Service Discovery****

Acts as a guide to microservices to find the route of communication between them as it maintains a list of services on which nodes are located.

Now, let’s look into the pros and cons of this architecture to gain a better understanding of when to use this architecture.

## ****Pros and Cons of Microservice Architecture****

Refer to the table below.

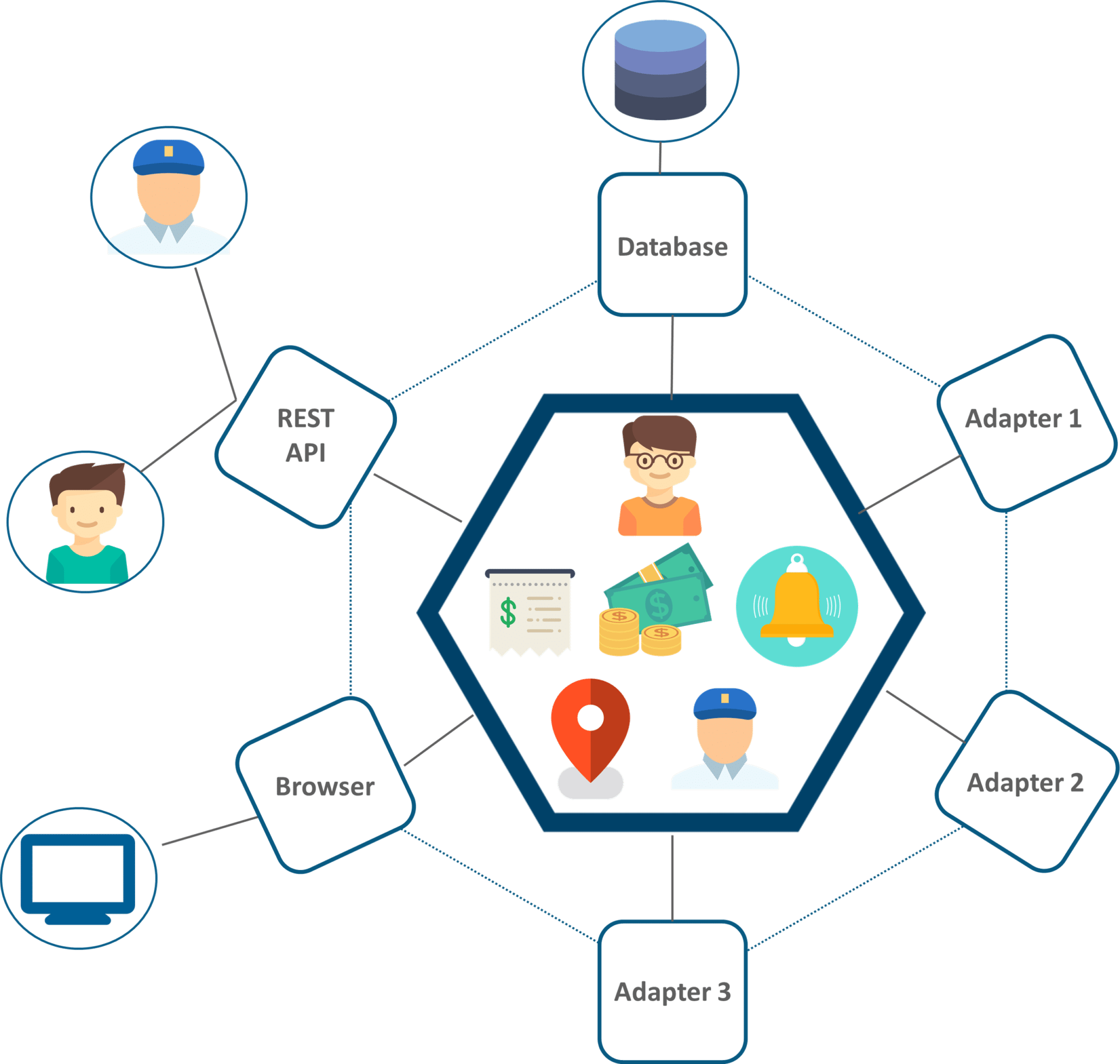


Let us understand more about microservices by comparing Uber previous architecture to the present one.

## ****Uber Case Study****

### ****Uber's Previous Architecture****

Like many startups, Uber began its journey with a monolithic architecture built for a single offering in a single city. Having one codebase seemed cleaned at that time, and solved Uber's core business problems. However, as Uber started expanding worldwide they rigorously faced various problems with respect to scalability and continuous integration.



**Figure 4:**Monolithic Architecture of Uber – Microservice Architecture.

The above diagram depicts Uber's previous architecture.

* A REST API is present with which the passenger and driver connect.
* Three different adapters are used with API within them, to perform actions such as billing, payments, sending emails/messages that we see when we book a cab.
* A MySQL database to store all their data.

So, if you notice here all the features such as passenger management, billing, notification features, payments, trip management and driver management were composed within a single framework.

### ****Problem Statement****

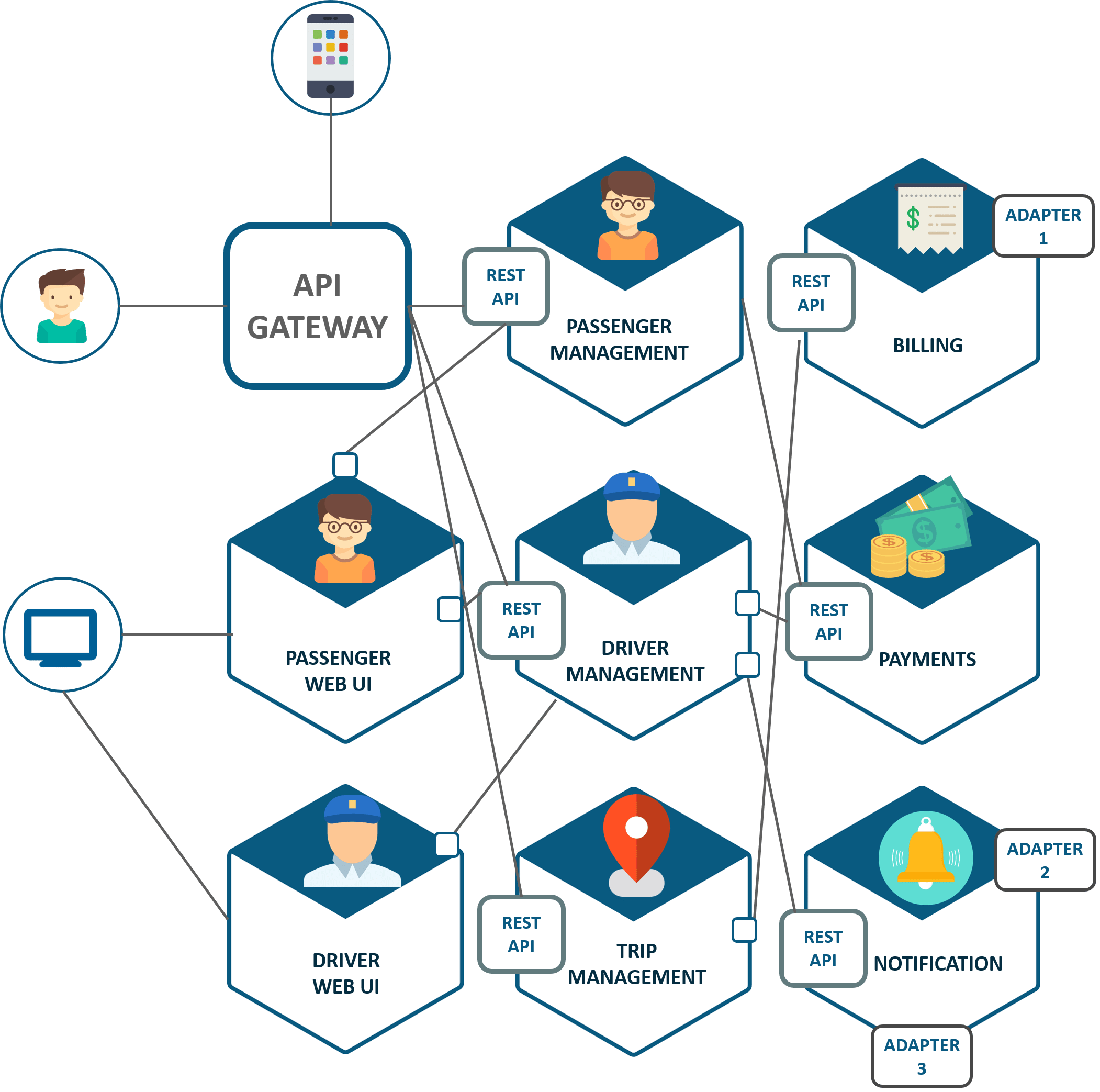
While Uber started expanding worldwide this kind of framework introduced various challenges. The following are some of the prominent challenges

* All the features had to be re-built, deployed and tested again and again to update a single feature.
* Fixing bugs became extremely difficult in a single repository as developers had to change the code again and again.
* Scaling the features simultaneously with the introduction of new features worldwide was quite tough to be handled together.

### ****Solution****

To avoid such problems Uber decided to change its architecture and follow the other hyper-growth companies like Amazon, Netflix, Twitter and many others. Thus, Uber decided to break its monolithic architecture into multiple codebases to form a microservice architecture.

Refer to the diagram below to look at Uber microservice architecture.



**Figure 5:**Microservice Architecture of Uber – Microservice Architecture.

* The major change that we observe here is the introduction of API Gateway through which all the drivers and passengers are connected. From the API Gateway, all the internal points are connected such as passenger management, driver management, trip management and others.
* The units are individual separate deployable units performing separate functionalities.
  + For Example: If you want to change anything in the billing microservices, then you just have to deploy only billing microservices and don’t have to deploy the others.
* All the features were now scaled individually i.e. The interdependency between each and every feature was removed.
  + For Example, we all know that the number of people searching for cabs is more comparatively more than the people actually booking a cab and making payments. This gets us an inference that the number of processes working on the passenger management microservice is more than the number of processes working on payments.

In this way, Uber benefited by shifting its architecture from monolithic to microservices.