

# JAVA 7 AND 8 FEATURES

---

# Java 7 feature: underscores in numeric literals

- You can place underscores between digits of any numeric literal like int, byte, short, float, long, double.
  - Using underscores in numeric literals will allow you to divide them in groups for better readability.

```
long ccNumber = 1234_5678_9012_3456L;  
long ssn = 999_99_9999L;  
float pi = 3.14_15F;  
long hexadecimalBytes = 0xFF_EC_DE_5E;  
byte byteInBinary = 0b0010_0101;  
int add = 12_3 + 3_2_1;  
long longInBinary = 0b11010010_01101001_10010100_10010010;
```

```
ccNumber=1234567890123456  
ssn=999999999  
pi=3.1415  
hexadecimalBytes=-1253794  
byteInBinary=37  
longInBinary=-764832622  
add=444
```

- ✓ Underscores can be placed only between digits.
- ✓ Can't put underscores next to decimal places, L/F suffix or radix prefix. So 3.\_14, 110\_L, 0x\_123 are invalid and will cause compilation error.
- ✓ Multiple underscores are allowed between digits, so 12\_\_\_\_3 is a valid number.
- ✓ Can't put underscores at the end of literal. So 123\_ is invalid and cause compile time error.

# Java 7 feature : Catching multiple exceptions

- Java 7 provides new MultiCatch clause
  - You can handle multiple discrete exception types in a single catch clause using the “|” or syntax

```
public class MultiCatchDemo {  
    public static void main(String[] args) {  
        int b = 0, x[] = { 10, 20, 30 };  
        /*try {  
            int c = x[3] / b;  
        } catch (ArithmeticException e) {  
            System.out.println(e);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println(e);  
        }*/  
  
        try {  
            int c = x[3] / b;  
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {  
            System.out.println(e);  
        }  
    }  
}
```

The type alternatives that are separated with vertical bars cannot have an inheritance relationship

```
try {  
    // execute code that may throw 1 of the 3 exceptions below.  
}  
catch(SQLException e) {  
    logger.log(e);  
}  
catch(IOException e) {  
    logger.log(e);  
}  
catch(Exception e) {  
    logger.severe(e);  
}
```

The two exceptions SQLException and IOException are handled in the same way, but you still have to write two individual catch blocks for them.

```
try {  
    // execute code that may throw 1 of th  
}  
catch(SQLException | IOException e) {  
    logger.log(e);  
}  
catch(Exception e) {  
    logger.severe(e);  
}
```

In Java 7 you can catch multiple exceptions using the multi catch syntax

## Other Java 7 features

- String in Switch
- The Diamond Operator
- try-with-resources

# JAVA 8 FEATURES

---

# forEach() method in Iterable interface

- Java 8 provides a new *forEach* method in java.lang.Iterable interface
  - Allows us to focus on business logic only.
  - *forEach()* takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse.

```
public class Java8ForEachExample {  
    public static void main(String[] args) {  
        List<Integer> myList = new ArrayList<Integer>();  
        for (int i = 0; i < 10; i++)  
            myList.add(i);  
  
        Iterator<Integer> it = myList.iterator();  
        while (it.hasNext()) {  
            Integer i = it.next();  
            System.out.println("Iterator Value::" + i);  
        }  
  
        myList.forEach(new Consumer<Integer>() {  
            public void accept(Integer t) {  
                System.out.println("forEach anonymous class Value::" + t);  
            }  
        });  
    }  
}
```

# Default and static methods in Interfaces

- Use **default** (must always be public) and **static** keyword to create interfaces with method implementation.

```
interface Sayable {  
    // Default method  
    default void say() {  
        System.out.println("Hello, this is default method");  
    }  
  
    // Abstract method  
    void sayMore(String msg);  
}  
  
public class DefaultMethods implements Sayable {  
    public void sayMore(String msg) { // implementing abstract method  
        System.out.println(msg);  
    }  
  
    public static void main(String[] args) {  
        DefaultMethods dm = new DefaultMethods();  
        dm.say(); // calling default method  
        dm.sayMore("Work is worship"); // calling abstract method  
    }  
}
```

# Static Methods inside Java 8 Interface

- You can also define static methods inside the interface. Static methods are used to define utility methods.

```
interface Sayable1 {  
    default void say() { // default method  
        System.out.println("Hello, this is default method");  
    }  
  
    void sayMore(String msg); // Abstract method  
  
    static void sayLouder(String msg) { // static method  
        System.out.println(msg);  
    }  
}  
  
public class DefaultStaticMethods implements Sayable1 {  
    public void sayMore(String msg) { // implementing abstract method  
        System.out.println(msg);  
    }  
  
    public static void main(String[] args) {  
        DefaultMethods dm = new DefaultMethods();  
        dm.say(); // calling default method  
        dm.sayMore("Work is worship"); // calling abstract method  
        Sayable1.sayLouder("Helloooo..."); // calling static method  
    }  
}
```



# Implementing Multiple Interfaces

- With default methods, there's the possibility of a class inheriting more than one method with the same signature.
- Which version of the method should be used? When such conflicts occur there are rules that specify how to deal with the conflict.

**1. Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.**

**2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected.** (If B extends A, B is more specific than A).

**3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly override the default methods if more than one versions are inherited.**

Compiler complains – you will HAVE to decide which interface method to override!!

- ResolvingMultipleDefaultsDemo.java
- For rule-2, see how B extends A →
- For seeing rule-3, simply remove extends A

```
interface A{
    default void sayHello(){
        System.out.println("Hello from A");
    }
}
interface B extends A{
    default void sayHello(){
        System.out.println("Hello from B");
    }
}
public class MultipleDefaultsDemo implements A, B {
    public static void main(String[] args) {
        new MultipleDefaultsDemo().sayHello();
    }
}
```

```
public class MyClass implements Interfacel, Interface2 {

    @Override
    public void method2() {}

    @Override
    public void method1(String str) {}

    // Myclass wont compile without having its
    // own log implementation
    @Override
    public void log(String str) {
        Interfacel.print(str);
    }
}
```

```
interface Interfacel {
    void method1(String str);

    default void log(String str) {
        System.out.println("Il logging : " + str);
    }

    static void print(String str) {
        System.out.println("Printing : " + str);
    }
}

interface Interface2 {
    void method2();

    default void log(String str) {
        System.out.println("Il logging : " + str);
    }
}
```

# Functional Interface

- A functional interface, introduced in Java 8, is an interface which has only a **single abstract method**.
  - It can have any number of default, static methods but can contain only one abstract method..
  - used extensively in lambda expressions.
  - Conversely, if you have *any* interface which has only a single abstract method, then that will effectively be a functional interface.
- **@FunctionalInterface** (optional) annotation can be used to explicitly specify that a given interface is to be treated as a functional interface.
  - Compiler would check and give a compile-time error in case the annotated interface does not satisfy the basic condition of qualifying as a functional interface

```
interface sayable{ //functional interface
    void say(String msg); // abstract method
}
@FunctionalInterface
interface doable extends sayable{
    // Invalid '@FunctionalInterface' annotation; doable is not a functional interface
    void doIt();
}
```

# Lambda Expressions

- A Lambda is a representation of an anonymous function which can be passed around as a parameter thus achieving behaviour parameterization.
  - A lambda consists of a list of parameters, a body, a return type and a list of exceptions which can be thrown. I.e. it is very much a function, just anonymous.
- Syntax : **variable-name = () -> { ... }**

```
() -> statement    // No argument and one-statement method body
```

```
arg -> statement    // One argument (parentheses can be omitted) and method body
```

```
(arg1, arg2, ...) -> {  
    body-block  
}    // Arguments separated by commas and block body
```

```
(Type1 arg1, Type2 arg2, ...) -> {  
    method-body-block;  
    return return-value;  
}    // With arguments and block body
```

```

interface FirstIntf{
    public void mymethod(String param);
}

interface CalcIntf{
    public int add(int a, int b);
}

public class LambdaDemo {
    public static void main(String[] args) {
        FirstIntf i1 = new FirstIntf() {
            @Override
            public void mymethod(String param) {
                System.out.println("Hello1 " + param);
            }
        };
        FirstIntf i2 = p -> {System.out.println("Hello " + p)};

        i1.mymethod("shri");
        i2.mymethod("soha");

        CalcIntf calc = (a,b) -> a+b;
        System.out.println(calc.add(10,20));
    }
}

```

```

Hello1 shri
Hello soha
30

```

# Lambda Expressions in Practice

- `Arrays.asList( "a", "b", "d" ).forEach( e -> System.out.println( e ) );`
  - Please notice the type of argument `e` is being inferred by the compiler. Alternatively, you may explicitly provide the type of the parameter, wrapping the definition in brackets. For example:
- `Arrays.asList( "a", "b", "d" ).forEach( ( String e ) -> System.out.println( e ) );`
  - If lambda's body is more complex, it may be wrapped into curly brackets. Eg:

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {  
    System.out.print( e );  
    System.out.print( e );  
} );
```

- Lambdas may return a value. The type of the return value will be inferred by compiler. The **return** statement is not required if the lambda body is just a one-liner. The two code snippets below are equivalent

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

```
Arrays.asList( "a", "b", "d" ).sort( ( e1, e2 ) -> {  
    int result = e1.compareTo( e2 );  
    return result;  
} );
```

```

public class ForEachDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("aaa", "bbb", "ccc");

        list.forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });
        list.forEach(item -> {
            System.out.println(item);
            System.out.println(item.length());
        });
    }
}

```

```

Map<String, Integer> items = new HashMap<>();
items.put("A", 10);
items.put("B", 20);
items.put("C", 30);
items.put("D", 40);
items.put("E", 50);
items.put("F", 60);

```

```

Key: A Val: 10
Key: B Val: 20
Key: C Val: 30
Key: D Val: 40
Key: E Val: 50
Key: F Val: 60

```

```

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

```

# Maps, forEach() and lambda

```
Map<Integer, String> hmap = new HashMap<>();  
hmap.put(1, "Jan");  
hmap.put(2, "Feb");  
hmap.put(3, "Mar");  
hmap.put(4, "Apr");
```

```
hmap.forEach((key, val) -> System.out.println(key + ":" + val));
```

```
hmap.forEach((key, value) -> {  
    if (key == 4) {  
        System.out.println("Value for key 4: " + value);  
    }  
});
```

```
hmap.forEach((key, value) -> {  
    if ("Mar".equals(value)) {  
        System.out.println("Key for Mar: " + key);  
    }  
});
```

```
1:Jan  
2:Feb  
3:Mar  
4:Apr  
Value for key 4: Apr  
Key for Mar: 3
```



# What Is a Stream?

- A stream is a sequence of data elements supporting sequential and parallel aggregate operations.
  - Computing the sum of all elements in a stream of integers, mapping all names in list to their lengths, etc. are examples of aggregate operations on streams.
  - How do streams differ from collections?
    - Both are abstractions for a collection of data elements. Collections focus on storage of data elements for efficient access whereas streams focus on aggregate computations on data elements from a data source that is typically, but not necessarily, collections.

# Features of streams

- Streams have no storage.
- Streams can represent a sequence of infinite elements.
- The design of streams is based on internal iteration.
- Streams are designed to be processed in parallel
- Streams are designed to support functional programming.
- Streams can be ordered or unordered.
- Streams cannot be reused.
- Streams support lazy operations.
  - A stream supports two types of operations:
    - Intermediate operations : are also known as lazy operations; does not process the elements of the stream until another eager operation is called on the stream.
    - Terminal operations : are also known as eager

```
List<Integer> ints = Arrays.asList(10,20,30,40,50);  
Stream instStream = ints.stream();
```

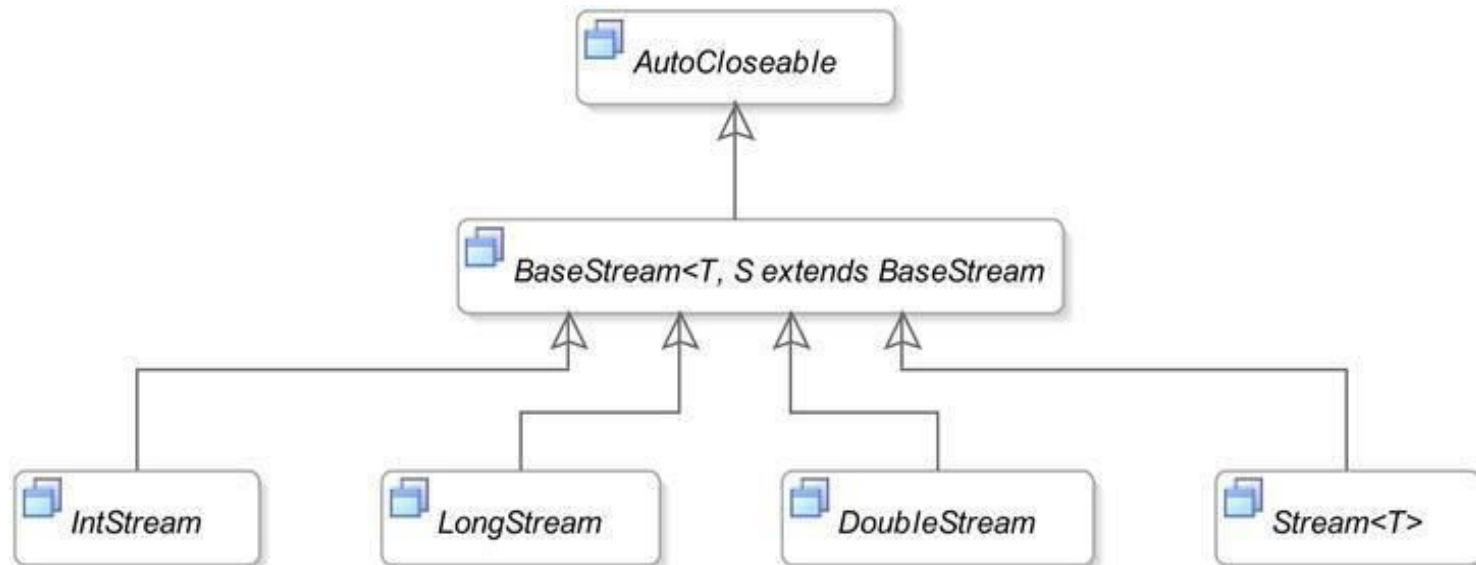
```
List<Person1> list = Arrays.asList(  
    new Person1("Asha", "Patel", 60),  
    new Person1("Alka", "Shachdeo", 62),  
    new Person1("Anil", "Patil", 45),  
    new Person1("Preet", "Singh", 49));
```

```
Stream pstream = list.stream();
```

```
Stream intParaStream = ints.parallelStream();
```

# Architecture of the Streams API : `java.util.stream` package

- All stream interfaces inherit from the `BaseStream` interface, which inherits from the `AutoCloseable` interface



# Example

- Read a list of integers and compute the sum of the squares of all odd integers in the list. Returns 35 as sum

```
public class FirstDemo {  
    public static void main(String[] args) {  
        List<Integer> nosList = Arrays.asList(1, 2, 3, 4, 5);  
        // Get the stream from the list  
        Stream<Integer> nosStream = nosList.stream();  
        // Get a stream of odd integers  
        Stream<Integer> oddNosStream = nosStream.filter(n -> n % 2 == 1);  
        // Get a stream of the squares of odd integers  
        Stream<Integer> sqNosStream = oddNosStream.map(n -> n * n);  
        // Sum all integers in the stream  
        int sum = sqNosStream.reduce(0, (n1, n2) -> n1 + n2);  
        System.out.println("The sum is : " + sum);  
    }  
}
```

```
/* The Integer class contains a static sum() method to perform sum of  
 * two integers. You can rewrite the code using a method reference: */  
int sum1 = sqNosStream.reduce(0, Integer::sum);  
System.out.println("The sum is : " + sum);
```

## Example repeated with statements combined

```
public class SecondDemo {  
  
    public static void main(String[] args) {  
        // Sum all integers in the numbers list  
        List<Integer> nosList = Arrays.asList(1, 2, 3, 4, 5);  
        int sum = nosList.stream()  
            .filter(n -> n % 2 == 1)  
            .map(n -> n * n)  
            .reduce(0, Integer::sum);  
  
        System.out.println("The sum is : " + sum);  
    }  
}
```

## Example with map()

```
List<String> names1 = Arrays.asList("Ben", "Anita", "Ava", "Brinda", "Acura");  
names1.stream()  
    .map(String::toLowerCase)  
    .filter(p -> p.startsWith("a"))  
    .forEach(System.out::println);
```

```
anita  
ava  
acura
```

# Creating Streams

- Based on the data source, stream creation can be categorized as:
  - Streams from values
  - Streams from functions
  - Streams from arrays
  - Streams from collections
  - Streams from files
  - Streams from other sources

# Streams from Values

- Stream interface contains two static **of()** methods to create a sequential Stream from a single value and multiple values:
  - <T> Stream<T> of(T t)
  - <T> Stream<T> of(T...values)

```
public class ThirdDemo {  
    public static void main(String[] args) {  
        //Create a stream with one string element  
        Stream<String> stream1 = Stream.of("Hello");  
        //Create a stream with multiple strings  
        Stream<String> stream2 = Stream.of("Ken", "Jeff", "Chris");  
        //Create a Stream of integers :  
        Stream<Integer> stream3 = Stream.of(1, 2, 3, 4);  
        //Create Stream from an array of objects.  
        String[] names = { "Ken", "Joy", "Al" };  
        Stream<String> stream4 = Stream.of(names);  
        //Create a stream from a String array returned from split()  
        String str = "meera,beena,cina";  
        Stream<String> stream5 = Stream.of(str.split(", "));  
  
        stream5.forEach(System.out::println);  
    }  
}
```

```
Stream.of("Anita", "Ava", "Acura", "Agatha")  
    .sorted()  
    .findFirst()  
    .ifPresent(System.out::print); //Acura
```

# IntStream interface

- The IntStream interfaces contain two static methods:
  - IntStream range(int start, int end) : specified end is exclusive
  - IntStream rangeClosed(int start, int end) : specified end is inclusive
    - They produce an IntStream that contains ordered integers between the specified start and end.
  - Eg : create an IntStream having integers 1, 2, 3, 4, and 5 as their elements:
  - IntStream is1 = IntStream.range(1, 6); //contains 1, 2, 3, 4, and 5
  - IntStream is2 = IntStream.rangeClosed(1, 5); //contains 1, 2, 3, 4, and 5

```
int sum = IntStream.range(1,11)
                    .sum();
System.out.println(sum);    //55
```

- Like the IntStream interface, the LongStream class also contains range() and rangeClosed() methods that takes arguments of type long and return a LongStream.



# Streams from Functions

- You can have a function that can generate infinite number of values on demand
- The Stream interface contains two static methods to generate an infinite stream:
  - `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)` : creates a **ordered** stream
  - `<T> Stream<T> generate(Supplier<T> s)` : creates a sequential **unordered** stream
- The `iterate()` method takes two arguments: a seed and a function.
  - 1st argument is a seed and the first element of the stream. Second element is generated by applying the function to the first element. Third element is generated by applying the function on the second element and so on
- Eg Create a stream of natural numbers
- `Stream<Long> naturalNumbers = Stream.iterate(1L, n -> n + 1);`
- Eg Create a stream of odd natural numbers
- `Stream<Long> oddNaturalNumbers = Stream.iterate(1L, n -> n + 2);`
- Eg Create a stream of the first 10 natural numbers; limits to 10 nos

```
Stream.iterate(1L, n -> n + 2)
    .skip(10)
    .limit(5)
    .forEach(System.out::println);
```

```
Stream.iterate(1L, n -> n + 1)
    .limit(10)
    .forEach(System.out::println); //1 3 5 7 9
```

# Using the generate( ) Method

- The generate(Supplier<T> s) method uses the specified Supplier to generate an infinite **sequential unordered** stream.
  - Eg : print five random numbers  $\geq 0.0$  and  $< 1.0$

```
Stream.generate(Math::random)
    .limit(5)
    .forEach(System.out::println);
```

```
0.7837051084269403
0.6981776357570071
0.37264422577861267
0.26388843562302533
0.03463039024854664
```

- Java 8 has added many methods to java.util.Random to work with streams
  - Methods like ints(), longs(), and doubles() return infinite IntStream, LongStream, and DoubleStream, respectively, which contain random numbers of the int, long, and double types.

- Alternatively:

```
// Print five random integers
new Random().ints()
    .limit(5)
    .forEach(System.out::println);
```

```
2050271737
-1427199800
-949910296
1066232863
-888885829
```

```
Stream.generate(new Random()::nextInt)
    .limit(5)
    .forEach(System.out::println);
```

# Using the generate( ) Method

- If you want to work with only primitive values, you can use the generate() method of the primitive type stream interfaces.
  - Eg : print five random integers using the generate() static method of the IntStream interface

```
IntStream.generate(new Random()::nextInt)
    .limit(5)
    .forEach(System.out::println);
```

- Eg generate an infinite stream of a repeating value

```
IntStream zeroes = IntStream.generate(() -> 0);
```

```
Stream<String> infinite1 = Stream.generate(() -> "hello")
    .limit(10);
```

# Streams from Arrays

- The Arrays class in the java.util package contains an overloaded stream() static method to create sequential streams from arrays
  - Eg Create a stream from an int array with elements 1, 2, and 3
  - `IntStream numbers = Arrays.stream(new int[]{1, 2, 3});`
  - Eg Creates a stream from a String array with elements "Ken", and "Jeff"
  - `Stream<String> nm = Arrays.stream(new String[] {"Ken", "Jeff"});`

Equivalent to : `Stream<String> nm = Stream.of(new String[] {"Ken", "Jeff"});`

```
String[] names = {"Ben", "Anita", "Joy", "Ava", "Brinda", "Acura"};
Arrays.stream(names)           //same as Stream.of(names)
    .filter(e -> e.startsWith("A"))
    .sorted()
    .forEach(System.out::println); //Acura, Anita, Ava
```

```
Arrays.stream(new int[] {2, 4, 6, 8, 10})
    .map(x -> x * x)
    .average()
    .ifPresent(System.out::println);|
```

# Streams from Collections

- The Collection interface contains the stream() and parallelStream() methods that create sequential and parallel streams from a Collection

```
List<String> strlist = Arrays.asList("aaa", "bbb", "ccc");  
// Create a sequential stream from the set  
Stream<String> seqStream = strlist.stream();  
// Create a parallel stream from the set  
Stream<String> parStream = strlist.parallelStream();
```

```
public class FourthDemo {  
  
    public static void main(String[] args) {  
        List<Employee> empList = new ArrayList<>();  
        empList.add(new Employee("Nataraja G", "Accounts", 8000));  
        empList.add(new Employee("Nagesh Y", "Admin", 15000));  
        empList.add(new Employee("Vasu V", "Security", 2500));  
        empList.add(new Employee("Amar", "Entertainment", 12500));  
  
        // find employees whose salaries are above 10000  
        empList.stream().filter(emp -> emp.getSalary() > 10000)  
                .forEach(System.out::println);  
    }  
}
```

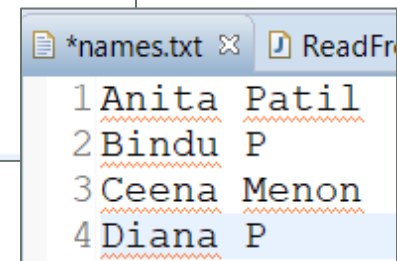
# Streams from Files

- Java 8 has added many methods to the classes in the `java.io` and `java.nio.file` packages to support I/O operations using streams.
  - For example, you can read text from a file as a stream of strings in which each element represents one line of text from the file.
  - Eg The `BufferedReader` and `Files` classes contains a `lines()` method that reads a file lazily and returns the contents as a stream of strings.
  - Each element in the stream represents one line of text from the file.

```
public static void main(String[] args) {  
    String fileName = "names.txt";  
  
    // read file into stream, try-with-resources  
    try (Stream<String> stream = Files.lines(Paths.get(fileName))) {  
        stream.forEach(System.out::println);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

# Streams from Files

```
public class ReadFromFileDemo4 {  
    public static void main(String[] args) throws IOException {  
        String fileName = "names.txt";  
        Stream<String> stream = Files.lines(Paths.get(fileName));  
        stream.sorted()  
            .filter(e -> e.length() > 10)  
            .forEach(System.out::println);  
    }  
}
```



1	Anita	Patil
2	Bindu	P
3	Ceena	Menon
4	Diana	P

Eg : convert the entire content to upper case, ignore line that starts with “three” and return content as a List.

```
public class ReadFromFileDemo2 {  
    public static void main(String[] args) {  
        String fileName = "nos.txt";  
        List<String> list = new ArrayList<>();  
  
        try (Stream<String> stream = Files.lines(Paths.get(fileName))) {  
            list = stream.filter(line -> !line.startsWith("three"))  
                .map(String::toUpperCase)  
                .collect(Collectors.toList());  
        } catch (IOException e) { e.printStackTrace(); }  
  
        list.forEach(System.out::println);  
    }  
}
```

# Streams from Files

```
public class ReadFromFileDemo3 {  
  
    public static void main(String[] args) {  
        String fileName = "names.txt";  
        List<String> list = new ArrayList<>();  
  
        try (BufferedReader br =  
            Files.newBufferedReader(Paths.get(fileName))) {  
            //br returns as stream and convert it into a List  
            list = br.lines().collect(Collectors.toList());  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        list.forEach(System.out::println);  
    }  
}
```



## collect()

- The collect() method of Stream class can be used to accumulate elements of any Stream into a Collection.
  - It is defined in Stream class. It accepts a Collector to accumulate elements of Stream into specified Collection.

```
public class CollectorsDemo {  
    public static void main(String[] args) {  
        List<String> strlist = Arrays.asList("Abc", "Joy", "Xyz", "Jamie");  
        List<String> strListWithJ =  
            strlist.stream()  
                .filter(s -> s.startsWith("J"))  
                .collect(Collectors.toList());  
        System.out.println(strListWithJ);  
    }  
}
```

```
public class CollectorsDemo {  
    public static void main(String[] args) {  
        List<Person> list = new ArrayList<>();  
        list.add(new Person(100, "Mohan"));  
        list.add(new Person(200, "Sohan"));  
        list.add(new Person(300, "Mahesh"));  
        List<String> names = list.stream()  
            .map(Person::getName)  
            .collect(Collectors.toList());  
        System.out.println(names);  
    }  
}
```

## collect()

- A method that returns a comma-separated string based on a given list of integers. Each element should be preceded by the letter 'e' if the number is even, and preceded by the letter 'o' if the number is odd.

```
public static String getString(List<Integer> list) {  
    return list.stream()  
        .map(i -> i % 2 == 0 ? "e" + i : "o" + i)  
        .collect(Collectors.joining(",")); //Returns a Collector that concatenates the input elements  
}
```

```
-----  
List<Integer> input2 = Arrays.asList(6, 3, 10, 34, 67);  
System.out.println(getString(input2));
```

e6,o3,e10,e34,o67

```
public static long count(List<String> list) {  
    return list.stream()  
        .collect(Collectors.counting()); //Returns a Collector that counts the number of input elements  
}
```

```
-----  
List<String> input1 = Arrays.asList("Ted", "Anna", "Bob", "Ana");  
System.out.println(count(input2)); //4
```

# The Date-Time API Java 8

- Java 8 introduced a new Date-Time API to work with date and time.
  - The earlier (pre-1.8) Date-Time API includes classes like Date, Calendar, GregorianCalendar, etc; in the java.util and java.sql packages.
  - The Java 8 release API can be found under the [java.time](#) package in JDK8; There are tons of useful classes in this package
- Following are some of the important classes introduced in java.time package.
  - Local: Simplified date-time API with no complexity of timezone handling.
  - Zoned: Specialized date-time API to deal with various timezones.
- Local API classes:
  - LocalDate — Local date without time in ISO format (yyyy-MM-dd).
  - LocalTime — Local time in ISO format.
  - LocalDateTime — Local date and time, where time zone is not taken into account.
- Zoned type has the only implementation class:
  - ZonedDateTime : Deals with time zone specific date and time; The ZoneId is an identifier used to represent different zones.

# The Date-Time API

- `java.time.LocalDate`
  - we can create `LocalDate` objects using one of the following its static methods:
    - `LocalDate nowDate = LocalDate.now();` //Obtains the current date from the system clock in the default time-zone.
    - `LocalDate date = LocalDate.of(int year, int month, int day);` //Jan is equivalent to 1
  - we can also create `LocalDate` from a `String`:
    - `LocalDate date = LocalDate.parse("2012-01-26");` // default format is yyyy-MM-dd

```
public class LocalDateDemo {  
  
    public static void main(String[] args) {  
        LocalDate nowDate = LocalDate.now();  
        LocalDate date = LocalDate.of(2020, 01, 01);  
        LocalDate strdate = LocalDate.parse("2012-01-26");  
  
        System.out.println("Current date : " + nowDate);  
        System.out.println("Set date : " + date);  
        System.out.println("Date with string : " + strdate);  
    }  
}
```

```
Current date : 2020-10-03  
Set date : 2020-01-01  
Date with string : 2012-01-26
```

```
LocalDate date = LocalDate.of(2020, 01, 01);
System.out.println("Set date : " + date);

System.out.println(date.plusDays(2));
System.out.println(date.plusMonths(3L));
System.out.println(date.plusWeeks(1));
System.out.println(date.plusYears(12));

System.out.println(date.minusDays(2L));
System.out.println(date.minusMonths(3));
System.out.println(date.minusWeeks(1L));
System.out.println(date.minusYears(12));

System.out.println(date.isBefore(LocalDate.now()));
System.out.println(date.isAfter(LocalDate.now()));

System.out.println(date.isLeapYear());
System.out.println(date.getDayOfWeek()); // returns
System.out.println(date.getYear());
System.out.println(date.getMonth());
System.out.println(date.getDayOfMonth());
System.out.println(date.lengthOfMonth());
System.out.println(nowDate.getDayOfYear());
```

```
Set date : 2020-01-01
2020-01-03
2020-04-01
2020-01-08
2032-01-01
2019-12-30
2019-10-01
2019-12-25
2008-01-01
true
false
true
WEDNESDAY
2020
JANUARY
1
31
277
```

# A Quick Example

```
import java.time.ZonedDateTime;
import static java.time.Month.DECEMBER;
public class FirstDemo {
    public static void main(String[] args) {
        LocalDate dateOnly = LocalDate.now();
        LocalTime timeOnly = LocalTime.now();
        LocalDateTime dateTime = LocalDateTime.now();
        ZonedDateTime dateTimeWithZone = ZonedDateTime.now();
        System.out.println(dateOnly);
        System.out.println(timeOnly);
        System.out.println(dateTime);
        System.out.println(dateTimeWithZone);
        // Construct a birth date and time from date-time components
        LocalDate bday = LocalDate.of(1993, DECEMBER, 18);
        LocalTime btime = LocalTime.of(4, 52);
        System.out.println("My Birth Date: " + bday);
        System.out.println("My Birth Time: " + btime);
    }
}
```

```
2020-10-03
10:35:39.180
2020-10-03T10:35:39.181
2020-10-03T10:35:39.181+05:30[Asia/Calcutta]
My Birth Date: 1993-12-18
My Birth Time: 04:52
```

# Time demos

- We can create `LocalTime` objects using one of the following its static methods:
  - `LocalTime.now();`
  - `LocalTime.of(int hour, int minute);`
  - `LocalTime.of(int hour, int minute, int second);`
  - `LocalTime.of(int hour, int minute, int second, int nanoOfSecond);`
  - `LocalTime.parse("12:05:12:001");`

```
LocalTime time = LocalTime.now();
System.out.println("local time now : " + time);
LocalTime parseTime = LocalTime.parse("02:08"); // parse a string
System.out.println(parseTime);
// specify hour-minute-second - 08:45:20 AM
LocalTime specificTime = LocalTime.of(8, 45, 20);
System.out.println(specificTime);
LocalTime newTime = time.plusHours(2); // adding two hours
System.out.println("Time after 2 hours : " + newTime);
```

```
local time now : 10:49:57.286
02:08
08:45:20
Time after 2 hours : 12:49:57.286
```

# Zone Demos

- Zoned date-time API is to be used when time zone is to be considered.
- The `ZoneId` is an identifier used to represent different zones.

```
ZoneId currentZone = ZoneId.systemDefault();
System.out.println("CurrentZone: " + currentZone);

//alternatively:
System.out.println("Current zone: " +
    ZonedDateTime.now().getZone());

ZoneId zone = ZoneId.of("Europe/Paris");
System.out.println("ZoneId: " + zone);

ZonedDateTime datel = ZonedDateTime.
    parse("2007-12-03T10:15:30+05:30[Asia/Karachi]");
System.out.println("datel: " + datel);

Set<String> allZoneIds = ZoneId.getAvailableZoneIds();
System.out.println(allZoneIds);
```

```
Current zone: Asia/Calcutta
ZoneId: Europe/Paris
datel: 2007-12-03T10:15:30+05:00[Asia/Karachi]
[Asia/Aden, America/Cuiaba, Etc/GMT+9, Etc/GMT
```



# XML

---

# Introduction to XML

- XML, the eXtensible Markup Language, is described as a means of structuring data.
  - XML provides rules for placing text and other media into structures and allows you to manage and manipulate the results.
  - XML standard is a subset of the Standard Generalized Markup Language (SGML), and was developed in 1996 by the SGML working group.
- HTML, an application of SGML, contains predefined set of tags.
  - HTML is designed for Web publishing. However, HTML is inflexible in that it cannot allow domain-specific tag sets to be created and used without formally introducing them into the HTML DTDs.

# XML versus HTML

- Both XML and HTML are based on SGML.
- HTML tells how the data should look. However, XML tells you what it means.
- XML is not meant to replace HTML; XML focuses on data and HTML focuses on how data is presented
- XML is not a way to design your home page, nor will it change the way you build sites
- Unlike HTML, with XML you can create your own tags.
- Its mainly used to **structure, store and transport data**

*HTML tells how the data should look, but XML tells you what it means.*

## In HTML:

```
<p>P200 Laptop  
<br>Friendly Computer Shop  
<br>$1438
```

## In XML:

```
<product>  
<model>P200 Laptop</model>  
<dealer>Friendly Computer Shop</dealer>  
<price>$1438</price>  
</product>
```

## Anatomy of an XML Document : an example

```
<?xml version="1.0"?>
<BOOK>
  <TITLE>King of the Murgos</TITLE>
  <AUTHOR>Eddings, David</AUTHOR>
  <PUBLISHER>Del Ray</PUBLISHER>
  <COVER TYPE="PAPERBACK" />
  <CATEGORY CLASS="FANTASY" />
  <ISBN>0-345-41920-0</ISBN>
  <RATING NUMBER="5" />
  <COMMENTS>Book 2 of the Malloreon. </COMMENTS>
</BOOK>
```

- Different parts of the XML file:
  - XML Declaration: It is a processing instruction
  - Root Element::Each XML document must have only one root element, all the other elements must be completely enclosed in that element.
  - An Empty Element: is an element that may not have any content.
  - Attributes:
    - Element tags can include one or more optional or mandatory attributes that give further information about the element they delimit.

# Well Formed XML Documents

- An XML document with correct syntax is "Well Formed".
- The syntax rules:
  - must begin with the XML declaration
  - XML documents must have a root element
  - XML elements must have a closing tag
  - XML tags are case sensitive
  - If an element is empty, it still must be closed.
  - XML elements must be properly nested
  - XML attribute values must be quoted
- Naming Rules:
  - A name consists of at least one letter: a to z, or A to Z.
  - May start with an underscore ( \_ ) or a colon ( : ) for multicharacter name.
  - Initial letter can be followed by letters, digits, hyphens, underscores, full stops.
- Comments:
  - Comments have the following form: <!-- This is comment text -->
  - Everything in comment text is completely ignored by the XML processor.

```
<?xml version="1.0"?>
<Employee>
  <ECode>1111</ECode>
  <Ename>
    <Fname>Neeta</Fname>
    <Lname>Singh</Lname>
  </Ename>
  <Desig desigId="4"/>
  <Salary> 21000 </Salary>
</Employee>
```

# Examples

```
<?xml version="1.0"?>
<note time="12:03:46">
  <to>Tove</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Meeting this weekend!</body>
</note>
```

```
<?xml version="1.0" ?>
<BankAccount acctId="1234">
  <Name>Darshan Singh</Name>
  <Type>Checking</Type>
  <OpenDate>11/04/1974</OpenDate>
  <Balance>25382.20</Balance>
</BankAccount>
```

```
<bookstore>
  <book category="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

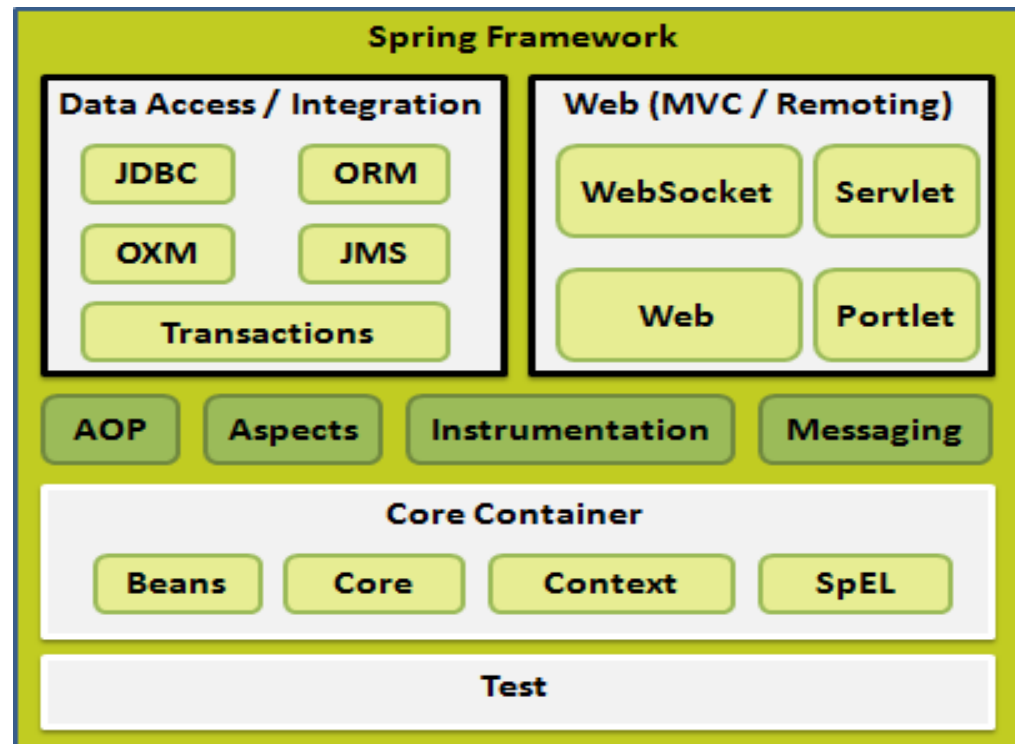
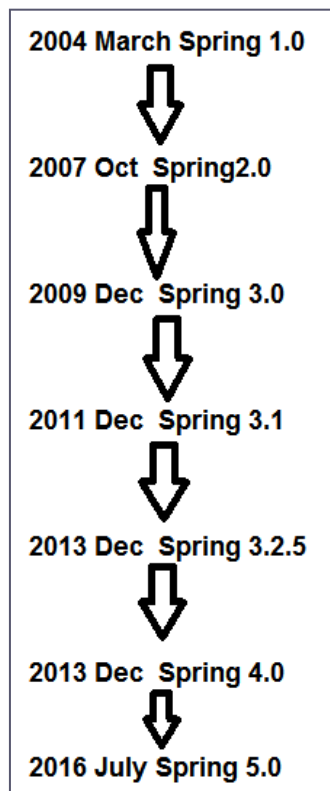
```
<?xml version="1.0" ?>
<Employees>
  <Employee>
    <empid>1001</empid>
    <EmpName>Vipul</EmpName>
    <Desig>Software Analyst</Desig>
  </Employee>
  <Employee>
    <Empid>1002</Empid>
    <EmpName>Vivek</EmpName>
    <Desig>Software Analyst</Desig>
  </Employee>
</Employees>
```

SPRING 5

---

# Spring

- Spring Framework project founded in Feb 2003
- Spring is an open source framework created by Rod Johnson, Juergen Hoeller etc
- Spring is a lightweight inversion of control and aspect-oriented container framework
  - version 5.0.6 released on May 8, 2018. Version 5.2.7 as in Jun2020





# Spring Jumpstart with HelloWorld

```
package training.spring;
public class HelloWorld {
    public void sayHello(){
        System.out.println("Hello Spring 3.0");
    }
}
```

```
<?xml .....>
<beans ....>
<bean id="HWBean" class =
    "training.spring.HelloWorld" />
</beans>
```

The Spring configuration file

```
public class HelloWorldClient {
    public static void main(String[] args) {
        XmlBeanFactory beanFactory = new XmlBeanFactory
            (new ClassPathResource("HelloWorld.xml"));
        HelloWorld bean = (HelloWorld) beanFactory.getBean("HWBean");
        bean.sayHello();
    }
}
```

Output:  
Hello Spring 3.0

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(resource);
HelloWorld hw = (HelloWorld)factory.getBean("hw");
```

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("currencyconverter.xml");
CurrencyConverter cc = (CurrencyConverter)ctx.getBean("converter");
```

# Injecting dependencies via setter methods

```
public interface CurrencyConverter {  
    public double dollarsToRupees(double dollars);  
}
```

```
public class CurrencyConverterImpl implements CurrencyConverter {  
    private double exchangeRate;  
    public double getExchangeRate(){ return exchangeRate; }  
    public void setExchangeRate(double exchangeRate){  
        this.exchangeRate = exchangeRate;    }  
    public double dollarsToRupees(double dollars) {  
        return dollars * exchangeRate;  
    }  
}
```

```
<beans .....>  
    <bean id="currencyConverter"  
        class="training.Spring.CurrencyConverterImpl">  
        <property name="exchangeRate" value="44.50" />  
    </bean>  
</beans>
```

```
<property name=" exchangeRate">  
    <value>44.50</value>  
</property>
```

```
public class CurrencyConverterClient {  
    public static void main(String args[]) throws Exception {  
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("currencyconverter.xml");  
        CurrencyConverter curr = (CurrencyConverter) factory.getBean("currencyConverter");  
        double rupees = curr.dollarsToRupees(50.0);  
        System.out.println("50 $ is "+rupees+" Rs.");  
    } }
```

Output:  
50 \$ is 2225.0 Rs.

# Injecting dependencies via constructor

- Bean classes can be programmed with constructors that take enough arguments to fully define the bean at instantiation

```
<bean id="currencyConverter" class="training.Spring.CurrencyConverterImpl">
  <constructor-arg>
    <value> 44.50 </value>
  </constructor-arg>
</bean>
```

```
public class CurrencyConverterImpl implements CurrencyConverter {
    private double exchangeRate;
    public CurrencyConverterImpl(double exchangeRate) {
        this.exchangeRate = exchangeRate;
    }
}
```

If a constructor has multiple arguments, then constructor arguments can be dealt with in two ways :

- by index : index attribute always starts with 0
- by type : used when class contains two constructor methods and both accept same number of arguments with different data types

```
<beans>
  <bean id="currencyConverter" class="com.trg.CurrencyConverterImpl">
    <constructor-arg><value>44.25</value></constructor-arg>
    <!--<constructor-arg index="0"><value>44.25</value></constructor-arg>-->
    <!--<constructor-arg type="double"><value>44.25</value></constructor-arg>-->
  </bean>
</beans>
```

# IoC in action: Wiring Beans

- The act of creating associations between application components is known as **wiring**.
  - There are many ways of wiring components together, but most commonly used is XML.

```
<bean id="exchangeService" class="ExchangeServiceImpl" />
<bean id="currencyConverter" class="CurrencyConverterImpl">
  <property name="exchangeService">
    <ref bean="exchangeService" />
  </property>
  <property name="message" ref="beanid" />
</bean>
```

Can be done for constructor injection too: `<constructor-arg ref="yetAnotherBean"/>`

```
<bean id="courseService" class="com.trg.courseServiceImpl">
  <property name="courseDAO">
    <ref bean="courseDAO" />
  </property>
  <property name="studentService">
    <ref bean="studentService" />
  </property>
</bean>
```

```
<bean id="courseDAO" class="com.trg.CourseDAO">
<bean id="studentService" class="com.trg.StudentService">
```

```
public class CourseServiceImpl{

    private CourseDAO courseDAO;
    private StudentService studentService;
    .....
}
```

# Bean scopes

- By default, all Spring beans are singletons.
  - But each time a bean is asked for, prototyping lets the container return a new instance. This is achieved through the scope attribute of `<bean>`
  - Example: `<bean id="foo" class="com.spring.Foo" scope="prototype" />`
- Additional Bean scopes:
  - **Singleton**: default scope; single instance created per Spring container
  - **Prototype**: new instance created for every `getBean()` request
  - **Request**: Only valid when used with a web-capable Spring context (such as with Spring MVC); bean instantiated for every HTTP request and then destroyed when the request is completed.
  - **Session**: beans with this scope will be instantiated for every HTTP session and then destroyed when the session is over. Valid for web apps only.
  - **Global session**: works only in portlet environments that use Spring and create a bean for every new portlet session.

# Autowiring

- Autowiring allows Spring to wire all bean's properties automatically by setting the autowire property on each <bean> that you want autowired

```
<bean id="foo" class="com.trg.Foo" autowire="autowire type" />
```

- Four types of autowiring:

no	(Default) No autowiring.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

# Using collections for injection

- Spring allows you to inject a collection of objects into your beans.
  - You can choose either <list>, <map>, <set> or <props> to represent a List, Map, Set or Properties instance.

```
public class CurrencyCollection {  
    private List currencies;  
  
    public List getCurrencies() {  
        return currencies;  
    }  
    public void setCurrencies(List currencies) {  
        this.currencies = currencies;  
    }  
}
```

```
<bean id="currencyList"  
      class="com.CurrencyCollection">  
    <property name="currencies">  
        <list>  
            <value>USD</value>  
            <value>EUR</value>  
            <value>INR</value>  
            <value>GBP</value>  
        </list>  
    </property>  
</bean>
```

```
public class ClientApp {  
    public static void main(String args[]) throws Exception {  
        ApplicationContext ctx = ...;  
        CurrencyConverterCollection curr = ctx.getBean("currencyList");  
        List<String> currencies = curr.getCurrencies();  
        System.out.println(currencies);  
    }  
}
```

Output : [USD, EUR, INR, GBP]

```

<bean id = "javaCollection" class = "com.JavaCollection">
  <property name = "addressList">
    <list>
      <value>Pune</value>
      <value>Mumbai</value>
    </list>
  </property>
  <property name = "addressSet">
    <set>
      <value>Pune</value>
      <value>Mumbai</value>
    </set>
  </property>
  <property name = "addressMap">
    <map>
      <entry key = "1" value = "Pune"/>
      <entry key = "2" value = "Nagpur"/>
    </map>
  </property>
  <property name = "addressProp">
    <props>
      <prop key = "one">Pune</prop>
      <prop key = "two">Mumbai</prop>
    </props>
  </property>
</bean>

```

```

public class JavaCollection {
    List addressList;
    Set addressSet;
    Map addressMap;
    Properties addressProp;
    //set/get methods
}

```

```

public class JavaCollectionClient {
    public static void main(String[] args) {
        ApplicationContext context = ....;
        JavaCollection jc =
            context.getBean("javaCollection");

        System.out.println(jc.getAddressList());
        System.out.println(jc.getAddressMap());
    }
}

```

```

[Pune, Mumbai]
{1=Pune, 2=Nagpur}

```



```

<bean id = "address1" class = "com.trg.ex5.Address">
  <property name="city" value="Pune"></property>
  <property name="state" value="Mah"></property>
</bean>
<bean id = "address2" class = "com.trg.ex5.Address">
  <property name="city" value="Mysore"></property>
  <property name="state" value="Kar"></property>
</bean>

<bean id = "bean" class = "com.BeanRefCollection">
  <property name = "addressList">
    <list>
      <ref bean = "address1"/>
      <ref bean = "address2"/>
    </list>
  </property>
  <property name = "addressSet">
    <set>
      <ref bean = "address1"/>
      <ref bean = "address2"/>
    </set>
  </property>
</bean>

```

```

public class Address {
  String city, state;
  //set/get methods
}

```

```

public class BeanRefCollection {
  List<Address> addressList;
  Set<Address> addressSet;
  //set/get methods
}

```

```

public static void main(String[] args) {
  ApplicationContext context = ...;
  BeanReferenceCollection br =
    context.getBean("....");

  System.out.println(br.getAddressList());
  System.out.println(br.getAddressSet());
}

```

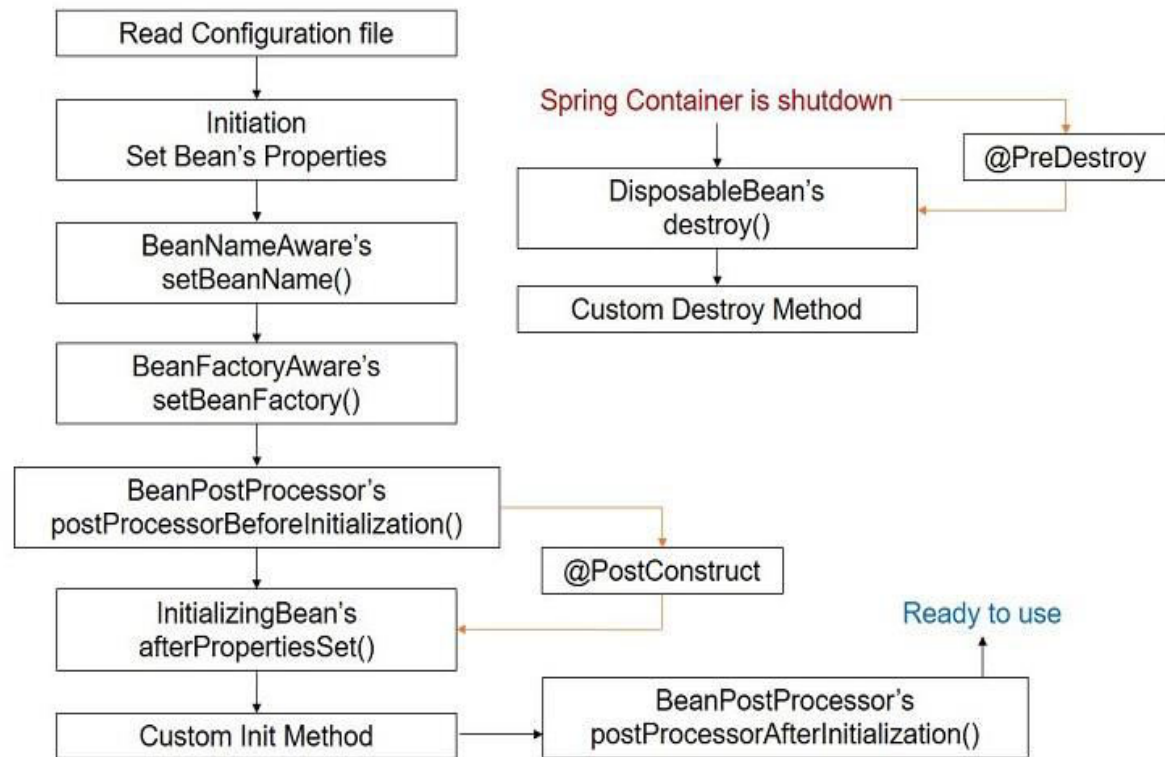
```

[Address [city=Pune, state=Mah], Address [city=Mysore, state=Kar]]
[Address [city=Pune, state=Mah], Address [city=Mysore, state=Kar]]

```

# Bean containers: concept

- The container or bean factory is at the core of the Spring framework and uses IoC to manage components.
- Bean factory is responsible to create and dispense beans.
- Spring has two types of containers:
  - BeanFactory : simplest, providing basic support for dependency injection
  - ApplicationContext : that build on bean factory by providing application framework services



# Initialization and Destruction

- When a bean is instantiated, some initialization can be performed to get it to a usable state
  - When the bean is removed from the container, some cleanup may be required
  - Spring can use two life-cycle methods of each bean to perform this setup and teardown.
  - Example:

```
<bean id="foo" class="com.spring.Foo"  
      init-method="setup"  
      destroy-method="teardown" />
```

*Method names can be anything; however, methods should return void and accept nothing as input arguments*

- **InitializingBean and DisposableBean**
  - InitializingBean interface : provides `afterPropertiesSet()` method which is called once all specified properties for the bean have been set.
  - DisposableBean interface : provides `destroy()` method which is called when the bean is disposed by the container
    - **Benefit:** Spring container is able to automatically detect beans without any external configuration
    - **Drawback:** application's beans are coupled to Spring API

# Bean containers : Application context

- Provides application framework services such as :
  - Resolving text messages, including support for internationalization of these messages
  - Easier integration with Spring's AOP features
  - Event publication; publish events to beans that are registered as listeners
- Many implementations of application context exist:
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext

## Some examples :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("app.xml");
```

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("/some/file/path/app.xml");
```

```
ApplicationContext ctx = new
```

```
    ClassPathXmlApplicationContext( new String[]{"app1.xml","app2.xml"});
```

```
// combines multiple xml file fragments
```

# Customizing beans with Post Processors

- BeanFactoryPostProcessor performs post processing on the entire Spring container.
  - Externalizing properties using PropertyPlaceholderConfigurer indicates Spring to load certain configuration from an external property file.

```
<bean id="placeHolderConfig"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="data.properties" />
</bean>
<bean id="dataSource" class="com.spring.ConnectionDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

## # data.properties file

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:trgdb
Jdbc.username=user1
Jdbc.password=user1
```

# PropertyPlaceholderConfigurer

- The other way is let spring does the bean registration automatically.
  - `<context:property-placeholder>` tag registers PropertyPlaceholderConfigurer automatically by spring context.
  - `<context:property-placeholder location="classpath:test.properties" />`
- You can include more than one properties file like below
  - `<context:property-placeholder  
location="classpath:test1.properties, classpath:test2.properties" />`

# Composing XML-based configuration metadata

- We have seen how bean definitions can span multiple XML files. Eg:
  - `ApplicationContext ctx =  
 new ClassPathXmlApplicationContext( new String[]{"app1.xml","app2.xml"});`
- One can also use one or more occurrences of the `<import/>` element to load bean definitions from another file or files. For example:

```
<beans>  
  <import resource="services.xml"/>  
  <import resource="resources/mybeans.xml"/>  
  <import resource="/resources/otherbeans.xml"/>  
  
  <bean id="bean1" class="..."/>  
  <bean id="bean2" class="..."/>  
</beans>
```

# Annotation-based configuration

- Spring has a number of custom annotations:

- @Required
- @Autowired
- @Resource
- @PostConstruct
- @PreDestroy

JSR-250 based annotations

Use @Required annotation over setter methods

The default mode in @Autowired is **byType**.

@Resource can be used instead of @Autowired + @Qualifier

- Annotations to configure beans:

- @Component
- @Controller
- @Repository
- @Service

- Some other transactions:

- @Transactional
- @AspectJ



## Some more annotations

- @Scope annotation

```
@Component
@Scope("prototype")
public class ShoppingCart { ... }
```

### Java Configuration (after Spring 4.3): Annotation-Driven Components

```
@Component
@RequestScope
public class OneClass { ... }
```

```
@Component
@SessionScope
public class AnotherClass { ... }
```

```
@Component
@ApplicationScope
public class YetAnotherClass { ... }
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
```

```
<context:annotation-config />
```

```
<!-- <bean class="org.springframework.beans.factory.annotation.
    AutowiredAnnotationBeanPostProcessor" /> -->
```

```
<context:component-scan base-package="training.spring" />
```

```
<!-- bean declarations go here -->
```

```
</beans>
```

Configuration file contents

# Some more annotations


- `@Value` : to inject property values into components.
  - The `@Value` annotation accepts a String value to specify the value to be injected into the built-in Java typed property.
  - Necessary type conversion is handled by the Spring Container.

```
public class SimpleCurrencyConverter {  
  
    @Value("68.0"); //injecting hardcoded value directly into property.  
    private double exchangeRate;  
  
    // setter/getter methods  
  
    public SimpleCurrencyConverter (@Value("68.0") double exchangeRate){  
        this.exchangeRate = exchangeRate;  
    }  
    public double dollarsToRupees(double amount) {  
        return exchangeRate * amount;  
    }  
}
```

# Some more annotations

- `@PropertySource` : used to read from properties file using Spring's Environment interface.

```
@Component
@PropertySource("classpath:value.properties")
public class SimpleCurrencyConverter {
    @Value("${exrate}")
    private double exchangeRate;
    // setter/getter methods
}
```



```
# value.properties file:
exRate=68.0
```

```
public class SimpleCurrencyConverterClient {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("anno.xml");
        SimpleCurrencyConverter cc = ctx.getBean(SimpleCurrencyConverter.class);
        System.out.println(cc.dollarsToRupees(60.0));
    }
}
```

# @Qualifier Annotation

```
public class Student {  
    private Integer age;  
    private String name;  
    //get/set methods  
}
```

```
public class Profile {  
    @Autowired  
    @Qualifier("student1")  
    private Student student;  
    ...  
}
```

```
<beans ...>  
    <bean id = "profile" class = "com.trg.Profile"></bean>  
  
    <bean id = "student1" class = "com.trg.Student">  
        <property name = "name" value = "Zara" />  
        <property name = "age" value = "11"/>  
    </bean>  
  
    <bean id = "student2" class = "com.trg.Student">  
        <property name = "name" value = "Nuha" />  
        <property name = "age" value = "2"/>  
    </bean>  
</beans>
```

```
public static void main(String[] args) {  
    ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");  
  
    Profile profile = (Profile) context.getBean("profile");  
    profile.printAge();  
    profile.printName();  
}
```

# Spring 3.x: Java Based Configuration

## //The bean

```
package com;  
public class HelloWorld {  
    private String message;  
    //setter/getter methods  
}
```

## //Configuration file

```
package com; //appropriate imports  
@Configuration  
public class HelloWorldConfig {  
    @Bean  
    public HelloWorld helloWorld(){  
        return new HelloWorld();  
    }  
}
```

*Is equivalent to*



```
<beans>  
    <bean id="helloWorld"  
        class="com.HelloWorld" />  
</beans>
```

## //client app

```
public static void main(String[] args) {  
    ApplicationContext ctx =  
        new AnnotationConfigApplicationContext(HelloWorldConfig.class);  
    HelloWorld helloWorld = ctx.getBean(HelloWorld.class);  
    //or (HelloWorld)ctx.getBean("helloWorld");  
    helloWorld.setMessage("Hello World!");  
    helloWorld.getMessage();  
}
```

# @Configuration & @Bean Annotations

- **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- **@Bean** annotation tells Spring that a method annotated with **@Bean** will return an object that should be registered as a bean in the Spring application context.
  - The method name is annotated with **@Bean** works as bean ID and it creates and returns the actual bean.
  - The configuration class can have a declaration for more than one **@Bean**.
  - Once your configuration classes are defined, you can load and provide them to Spring container using [\*AnnotationConfigApplicationContext\*](#)
  - You can load various configuration classes :

```
AnnotationConfigApplicationContext context = new  
    AnnotationConfigApplicationContext(ctx.register(Config1.class, Config2.class);
```

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
ctx.register(AppConfig.class, OtherConfig.class);  
ctx.register(AdditionalConfig.class);  
ctx.refresh();  
MyService myService = ctx.getBean(MyService.class);  
myService.doStuff();
```

# Customizing bean naming

- By default, `@Bean` method's name is used as the name of the resulting bean.
  - Can override this functionality with the `name` attribute.

```
@Bean(name = "myFoo")
public Foo foo(){
    return new Foo();
}
```

- You can also provide alias:

```
@Bean({"b1", "b2"})
public MyBean myBean() {
    return obj;
}
```

- To add a description to a `@Bean` the `@Description` annotation can be used

```
@Bean
@Description("Provides a basic example of a bean")
public Foo foo(){
    return new Foo();
}
```

# Injecting Bean Dependencies

- When @Beans have dependencies on one another, expressing that the dependency is as simple; just have one bean method call another

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }
    @Bean
    public Bar bar() {
        return new Bar();
    }
    @Bean
    public Foo foo(Bar bar) {
        return new Foo(bar);
    }
}
```

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.setBar(bar());
        return foo;
    }
    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

*Here, the foo bean receives a reference to bar via the constructor injection and setter injection*



# Specifying Bean Scope

- The default scope is singleton, but you can override this with the `@Scope` annotation

```
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public Foo foo() {
        return new Foo();
    }
}
```

## Java Configuration (after Spring 4.3)

```
@Configuration
public class MyConfiguration {

    @Bean
    @RequestScope
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
    @SessionScope
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @ApplicationScope
    public YetAnotherClass myApplicationBean()
        return new YetAnotherClass();
    }
}
```

# Using JDBC with Spring

- Spring separates the fixed and variant parts of the data access process into two distinct classes:
  - **Templates**: manage the fixed part of the process like controlling transactions, managing resources, handling exceptions etc
  - **Callbacks**: define implementation details, specific to application such as creating statements, binding parameters etc.
- **The JdbcTemplate class**
- Central class in JDBC framework
  - It simplifies the use of JDBC and helps to avoid common errors.
  - JdbcTemplate can be used to execute SQL queries or insert, update, and delete statements.
  - It executes core JDBC workflow, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative DataAccessException hierarchy defined by Spring

# Example

- A DataSource obtains a connection to the database for working with data.
- Some of the implementations of DataSource are:
  - BasicDataSource
  - PoolingDataSource
  - SingleConnectionDataSource
  - DriverManagerDataSource
- Configuring the datasource
  - For your data access objects, you normally declare a DataSource variable and set it using dependency injection.

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>  
  <property name="url" value="jdbc:oracle:thin:@oraserver:1521:oradb"/>  
  <property name="username" value="scott"/>  
  <property name="password" value="tiger"/>  
</bean>
```

# Wiring beans in the Spring Context file : The JdbcTemplate

```
<beans>
  <bean id="dataSource" ....
</bean>
  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource">
      <ref bean="dataSource" />
    </property>
  </bean>
  <bean id="empDao" class="com.trg.ex13.jt.EmpDaoImpl" />
</beans>
```

```
public class TestApp {
  public static void main(String[] args) {
    ApplicationContext ctx = ...
    EmpDaoImpl dao = ctx.getBean("empdao", EmpDaoImpl.class);
    System.out.println(dao.getJdbcTemplate());
  }
}
```

```
public class EmpDao {

  @Autowired
  JdbcTemplate jdbcTemplate;
  ....
}
```

- Alternatively, you can retrieve DataSource in Dao class and inject into JdbcTemplate.

```
public class EventDao {
  private JdbcTemplate jdbcTemplate;
  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }
}
```

# Wiring beans using Annotation : The JdbcTemplate

## **@Repository**

```
public class EventDao {  
    private JdbcTemplate jdbcTemplate;  
    @Autowired  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
    .....  
}
```

```
<beans ...>  
  <context:component-scan base-package="com.trg" />  
  
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"  
        destroy-method="close">  
    <property name="driverClassName" value="{jdbc.driverClassName}"/>  
    <property name="url" value="{jdbc.url}"/>  
    <property name="username" value="{jdbc.username}"/>  
    <property name="password" value="{jdbc.password}"/>  
  </bean>  
  <context:property-placeholder location="jdbc.properties"/>  
</beans>
```

```

public class MyJTDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("emp1.xml");
        JdbcTemplate jt = ctx.getBean("jdbcTemplate", JdbcTemplate.class);

        int count = jt.queryForObject("select count(*) from emp", Integer.class);
        System.out.println(count);

        int count1 = jt.queryForObject("select count(*) from emp where job = ?", Integer.class, "CLERK");
        System.out.println(count1);

        String ename = jt.queryForObject("select ename from emp where empno = ?",
            new Object[]{7782}, String.class);
        System.out.println("Emp name : " + ename);

        int sumsal = jt.queryForObject("select sum(sal) from emp where deptno = ? and sal < ?",
            new Object[]{10, 5000}, Integer.class);
        System.out.println(sumsal);

        String name = (String)jt.queryForObject("select dname from dept where deptno=10", String.class);
        System.out.println(name);

        List rows = jt.queryForList("select * from dept");
        System.out.println(rows);

        Object params[] = new Object[]{new Double(3000.0)};
        List rows1 = jt.queryForList("Select * from emp where sal > ?", params);
        System.out.println(rows1);

        int cnt1=jt.update("insert into dept(deptno, dname, loc) values(1,'Accounting','Pune')");
        System.out.println(cnt1);

        int cnt2=jt.update("Delete from dept where deptno=1");
        System.out.println(cnt2);
    }
}

```

```

14
4
Emp name : CLARK
3851
ACCOUNTING
[{DEPTNO=10, DNAME=ACCOUNTING, LOC=NEW YOR
[{EMPNO=7839, ENAME=KING, JOB=PRESIDENT, M
1
1

```

# JdbcTemplate – Handling Callback Interfaces

- There are a number of callback methods supported by JdbcTemplate class for performing different SQL operations.
- Interfaces that can be implemented to handle the returned rows are :
  - ResultSetExtractor
  - RowCallbackHandler
  - RowMapper

# RowMapper

- Spring's RowMapper and BeanPropertyRowMapper helps us to map the underlying resultset object to retrieve the desired values.

```
public class Emp {  
    String ename, desig;  
    int empid;  
    //set and get methods  
}
```

```
public class EmpMapper implements RowMapper<Emp>{  
    public Emp mapRow(ResultSet rs, int rownum) {  
        Emp emp = new Emp();  
        emp.setEmpid(rs.getInt("empid"));  
        emp.setEname(rs.getString(2));  
        emp.setDesig(rs.getString(3));  
        return emp;  
    }  
}
```

```
public Emp getEmpById(int id) {  
    String SQL = "select * from employee where empid=?";  
    Emp emp = (Emp) jdbcTemplate.queryForObject(SQL, new Object[] {id}, new EmpMapper());  
    return emp;  
}
```

//Retrieving single object

```
public List<Emp> getAllEmps() {  
    String SQL = "select * from employee";  
    List<Emp> emps = jdbcTemplate.query(SQL, new EmpMapper());  
    return emps;  
}
```

//Retrieving multiple objects



# RowMapper

- BeanPropertyRowMapper

- If bean properties and column names are the same, then simplest option is to use BeanPropertyRowMapper .

```
public List<Emp> getAllEmps(){  
    String sql = "SELECT * FROM EMPLOYEE";  
    List<Emp> emps = jdbcTemplate.query(sql, new BeanPropertyRowMapper(Emp.class));  
    return emps;  
}
```

- Create RowMapper implementations anonymously

```
List<Contact> listContact = jdbcTemplate.query(sqlSelect, new RowMapper<Contact>(){  
  
    public Contact mapRow(ResultSet result, int rowNum) throws SQLException {  
        Contact contact = new Contact();  
        contact.setName(result.getString("name"));  
        contact.setEmail(result.getString("email"));  
        contact.setAddress(result.getString("address"));  
        contact.setPhone(result.getString("telephone"));  
        return contact;  
    }  
});
```

# NamedParameterJdbcTemplate class

- Wraps a JdbcTemplate to provide more convenient usage with named parameters instead of the traditional JDBC "?" place holders.

```
@Repository("namedParameterDemo")
public class NamedParameterJTDemo {

    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

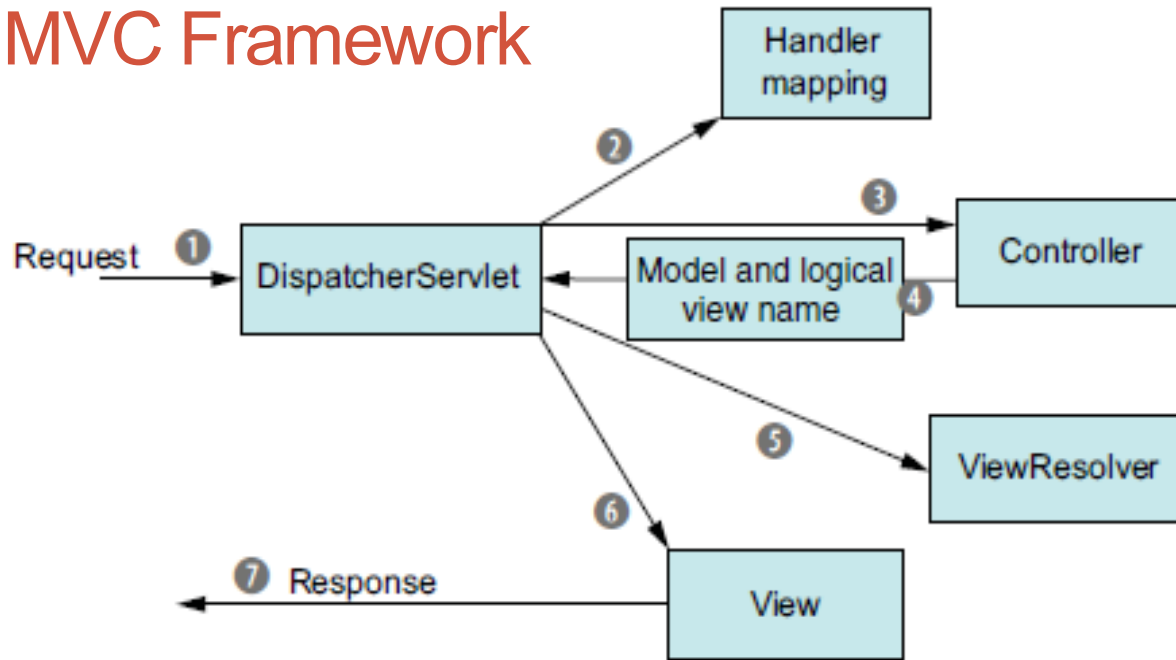
    public void addEmp(Emp emp) {
        System.out.println(namedParameterJdbcTemplate);
        String sql = "INSERT INTO employee "
            + "(empid, ename, desig, sal) VALUES (:empid, :empname, :empjob, :empsal)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("empid", emp.getEmpid());
        parameters.put("empname", emp.getEname());
        parameters.put("empjob", emp.getDesig());
        parameters.put("empsal", emp.getSal());

        namedParameterJdbcTemplate.update(sql, parameters);
    }

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("emp1.xml");
        NamedParameterJTDemo demo = ctx.getBean("namedParameterDemo", NamedParameterJTDemo.class);
        Emp emp = new Emp("Harry", "CLERK", 191, 2300);
        demo.addEmp(emp);
    }
}
```

# Spring MVC Framework



**<!-- web.xml -->**

```
<servlet>
  <servlet-name>basicspring</servlet-name>
  <servlet-class> org.springframework.web.servlet.DispatcherServlet
</servlet>

<servlet-mapping>
  <servlet-name>basicspring</servlet-name>
  <url-pattern>*.obj</url-pattern>
</servlet-mapping>
```

WEB-INF

lib

- commons-logging-1.2.jar
- jstl.jar
- spring-aop-5.0.3.RELEASE.jar
- spring-beans-5.0.3.RELEASE.jar
- spring-context-5.0.3.RELEASE.jar
- spring-core-5.0.3.RELEASE.jar
- spring-expression-5.0.3.RELEASE.jar
- spring-web-5.0.3.RELEASE.jar
- spring-webmvc-5.0.3.RELEASE.jar
- standard.jar
- app-servlet.xml
- c.tld
- web.xml

# Building a MVC app

- Steps to build a homepage in Spring MVC:
  1. Write the controller class that performs the logic behind the homepage
  2. Configure controller in the DispatcherServlet's context configuration file
  3. Configure a view resolver to tie the controller to the JSP
  4. Write the JSP that will render the homepage to the user
- Step 1 : Controller handles request and performs business logic

```
@Controller
public class HelloController{
    @RequestMapping("/helloWorld")
    public ModelAndView handleRequest(HttpServletRequest request,
                                    HttpServletResponse response){
        String today= new java.util.Date().toString();
        return new ModelAndView("hello", "now", today);
    }
}
```

- Step 2: Configure the controller in the DispatcherServlet's context configuration file

# Building a Homepage in Spring MVC

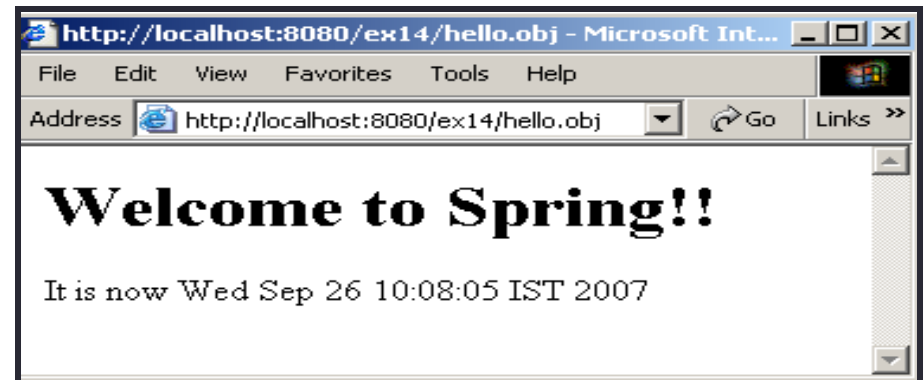
- Step 3 : Configure a view resolver to bind controller to the JSP

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
      <value>org.springframework.web.servlet.view.JstlView </value>
    </property>
    <property name="prefix"><value>/</value></property>
    <property name="suffix"><value>.jsp</value></property>
  </bean>
```

} /hello.jsp

- Step 4 : Write the JSP that will render the homepage to the user

```
<html>
  <body>
    <h1>Welcome to Spring!! </h1>
    Now it is ${now}
  </body>
</html>
```



# The configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">

<context:component-scan base-package="com.trg.mvc" />
<mvc:annotation-driven/>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass">
<value>org.springframework.web.servlet.view.JstlView</value>
</property>
<property name="prefix">
<value>/</value>
</property>
<property name="suffix">
<value>.jsp</value>
</property>
</bean>

</beans>
```

# Example

@Controller

```
public class HelloController2 {  
    @RequestMapping("/helloWorld")  
    public String handleMyRequest(Map<String, Object> model){  
        String now = new java.util.Date().toString();  
        String msg="Hi there";  
        model.put("now", now);  
        model.put("greet",msg);  
        return "hello"; //returning a string  
    }  
}
```

```
<html>  
<body>  
    <h1>Welcome to Spring!!</h1>  
    Now it is ${now}  
    <br>  
    Message is : ${greet}  
</body>  
</html>
```

@Controller

```
public class HelloWorld3 {  
    @RequestMapping("/helloWorld3")  
    public String handleMyRequest(Model model) {  
        String now = new java.util.Date().toString();  
        String msg="Hi there";  
        model.addAttribute("now", now);  
        model.addAttribute("greet",msg);  
        return "hello";  
    }  
}
```

## Welcome to Spring!!

Now it is Wed Jul 03 14:20:31 IST 2019  
Message is : Hi there

# Partial list of valid argument types in handler method

- `HttpServletRequest` or `HttpServletResponse`
- Request parameters of arbitrary type, annotated with `@RequestParam`
- `Map` or `Model`, for the handler method to add attributes to the model
- `HttpSession` : Session object; An argument of this type enforces the presence of a corresponding session.
- `java.io.InputStream` / `java.io.Reader` for access to the request's content
- `java.io.OutputStream` / `java.io.Writer` for generating the response's content.
- `@PathVariable` annotated parameters for access to URI template variables
- Model attributes of arbitrary type, annotated with `@ModelAttribute`
- Cookie values included in an incoming request, annotated with `@CookieValue`
- `Errors` or `BindingResult`, for the handler method to access the binding and validation result for the command object
- `@RequestBody` annotated parameters for access to the HTTP request body. Parameter values are converted to the declared method argument type using `HttpMessageConverters`



# Example

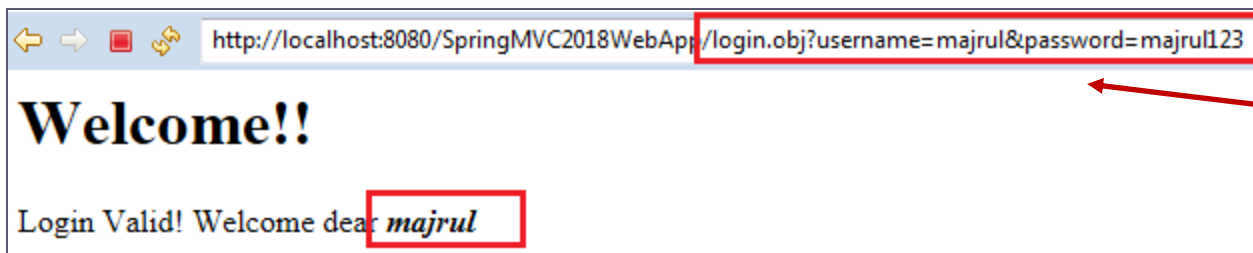
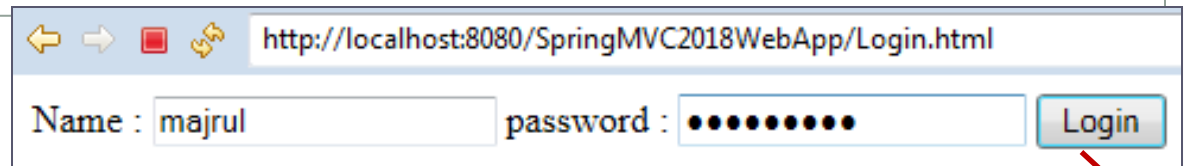
```
<form action="login.obj">
  Name : <input name="username">
  Password : <input type="password" name="password">
  <input type="submit" value="Login">
</form>
```

## //success.jsp

```
<body>
  <h1>Welcome!!</h1>
  Login Valid!
  Welcome ${username}
</body>
```

@Controller

```
public class LoginFormController {
  @RequestMapping(value = "/login", method = RequestMethod.GET)
  public String onSubmit(@RequestParam("username") String username,
    @RequestParam("password") String password, Model model) {
    model.addAttribute("username", username);
    if (username.equals("majrul") && password.equals("majrul123"))
      return "success";
    else return "failure";
  }
}
```



# @RequestMapping Variants; ver 4.3 onwards

- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
  - For eg, this code:

```
@RequestMapping(method = RequestMethod.GET)
public Map<String, Appointment> get() {
    return appointmentBook.getAppointmentsForToday();
}
```

- Can be written as:

```
@GetMapping
public Map<String, Appointment> get() {
    return appointmentBook.getAppointmentsForToday();
}
```

- Another eg:

```
@GetMapping
public String setupForm(@RequestParam("petId") int petId, ModelMap model)
```

# URI Template Patterns

- Use the `@PathVariable` annotation on a method argument to bind it to the value of a URI template variable:

```
@GetMapping("/owners/{ownerId}")
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

```
@RequestMapping(method=RequestMethod.GET, value="/{id}")
public String getEmployee(@PathVariable int id) {
    //do something with id
}
```

- A method can have any number of `@PathVariable` annotations:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public String findPet(@PathVariable String ownerId,
                     @PathVariable String petId )
```

# Example

```
@Repository("empDao")
public class EmpDao {
    List<Emp> emps = new ArrayList<>();

    public EmpDao() {
        emps.add(new Emp(101, "Soha", "SSE", 30000));
        emps.add(new Emp(102, "Arnav", "SE", 25000));
        emps.add(new Emp(103, "Shrilata", "Manager", 70000));
    }

    public Emp getEmpById(int id) {
        Emp getEmp = null;
        for (Emp emp : emps) {
            if (emp.getEmpId() == id) {
                getEmp = emp;
                break;
            }
        }
        return getEmp;
    }

    public List<Emp> getEmps(){
        System.out.println("in dao");
        return emps;
    }
}
```

```
public class Emp {
    int empId;
    String empName, desig;
    double sal;
    //appropriate constructors
    //get/set methods
}
```

```
@Service("empService")
public class EmpServiceImpl implements EmpService{

    @Autowired
    EmpDao empDao;

    public Emp getEmpWithId(int id) {
        return empDao.getEmpById(id);
    }

    public List<Emp> getAllEmps() {
        System.out.println("in service");
        return empDao.getEmps();
    }
}
```

## Example continued

displayOneEmp.jsp

```
@Controller
@RequestMapping("/employees")
public class EmpController {
```

```
    @Autowired
    EmpService empService;
```

```
    @RequestMapping("/{id}")
    public String getEmpWithId(@PathVariable int id, Model model){
        model.addAttribute("emp", empService.getEmpWithId(id));
        return "displayOneEmp";
    }
```

```
    @GetMapping(value = "/getAllEmployees")
    public String getAllEmployees(Model model){
        System.out.println("in method");
        model.addAttribute("employees", empService.getAllEmps());
        return "employeesDisplay";
    }
}
```

```
<body>
<h2>Employee in System</h2>

    Employee Id : ${emp.empId} <br>
    Employee Name : ${emp.empName} <br>
    Employee Designation : ${emp.desig} <br>
    Employee Salary : ${emp.sal} <br>

</body>
```

http://localhost:8080/SpringMVC2018WebApp/employees/102.obj

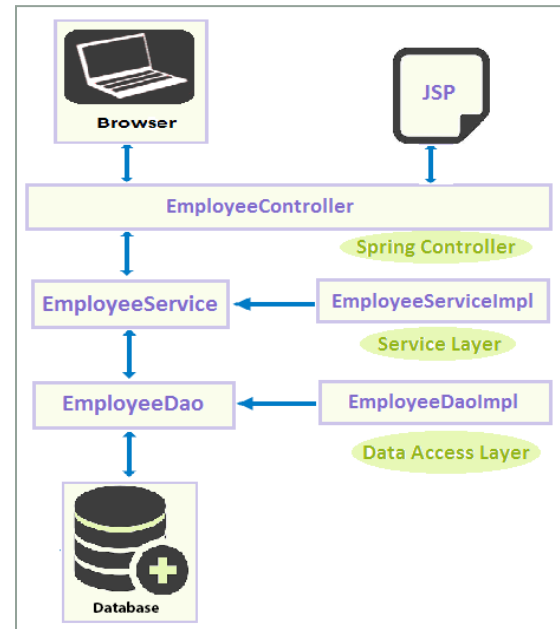
### Employee in System

Employee Id : 102  
Employee Name : Arnav  
Employee Designation : SE  
Employee Salary : 25000.0

# Example continued

```
<body>
<h2>All Employees in System</h2>
<table border="1">
  <tr>
    <th>Employee Id</th>
    <th>Employee Name</th>
    <th>Employee Designation</th>
    <th>Employee Salary</th>
  </tr>
  <c:forEach items="${employees}" var="employee">
    <tr>
      <td>${employee.empId}</td>
      <td>${employee.empName}</td>
      <td>${employee.desig}</td>
      <td>${employee.sal}</td>
    </tr>
  </c:forEach>
</table>
</body>
```

employeesDisplay.jsp



http://localhost:8080/SpringMVC2018WebApp/employees/getAllEmployees.obj

## All Employees in System

Employee Id	Employee Name	Employee Designation	Employee Salary
101	Soha	SSE	30000.0
102	Arnav	SE	25000.0
103	Shrilata	Manager	70000.0

# Supported method return types in a handler method

- The `@RequestMapping` annotated methods can have return values, some of which have been described below:

Return type	Description
<code>ModelAndView</code>	This holds Model and View information
<code>String</code>	This represents the View name
<code>View</code>	This represents the View object
<code>Model/Map</code>	This contains data exposed by a view; view is determined implicitly by the <code>RequestToViewNameTranslator</code> class
<code>Void</code>	This specifies that a view can be handled by the invoked method internally or can be determined implicitly by the <code>RequestToViewNameTranslator</code> class

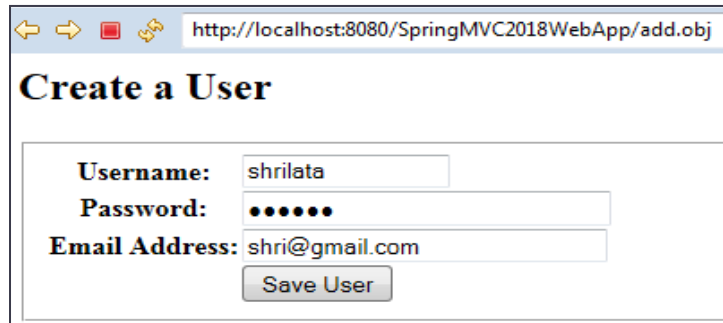
# Processing forms : Example

```
@Controller
public class UserController {

    @GetMapping(value = "/add")
    public String showForm(Model model) {
        model.addAttribute("userobj", new User());
        return "addForm";
    }

    @PostMapping(value = "/process")
    public String processForm(@ModelAttribute("userobj") User user, Model model) {
        model.addAttribute("username", user.getUsername());
        return "success";
    }
}
```

```
public class User {
    String username, password, email;
    //constructor
    //getter/setter methods
}
```



http://localhost:8080/SpringMVC2018WebApp/add.obj

### Create a User

Username:	<input type="text" value="shrilata"/>
Password:	<input type="password" value="....."/>
Email Address:	<input type="text" value="shri@gmail.com"/>



# Processing forms : Example

addForm.jsp

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="userobj" action="/process">
<table cellpadding="0">
  <tr>
    <th><sf:label path="username">Username:</sf:label></th>
    <td><sf:input path="username" size="15" maxlength="15"/>
  </tr>
  <tr>
    <th><sf:label path="password">Password:</sf:label></th>
    <td><sf:password path="password" size="30" showPassword="true"/> </td>
  </tr>
  ...
  <tr><th></th>
    <td><input name="commit" type="submit" value="Save User" /></td></tr>
</sf:form></div>
```

```
<body>
<h1>Welcome!!</h1>
  Login Valid!
  Welcome dear <i><b>${username}</b></i>
</body>
```

success.jsp

# Processing forms with validation : Example

```
@Controller
public class UserValidationController {
    @GetMapping(value = "/addUser")
    public String showForm(Model model){
        model.addAttribute("userobj",new UserVal());
        return "addUser";
    }
    @PostMapping(value = "/processForm")
    public String processForm(@Valid @ModelAttribute("userobj") UserVal user,
                             BindingResult bindingResult, Model model){
        if(bindingResult.hasErrors())
            return "addUser";
        else { // some logic to persist user
            model.addAttribute("username",user.getUsername());
            return "success";
        }
    }
}
```

addUser.jsp

## Create a User

Username:	<input type="text" value="jo"/>	Username must be between 3 and 20 characters long.
Password:	<input type="password" value="..."/>	The password must be at least 6 characters long.
Email Address:	<input type="text" value="shri"/>	Invalid email address.
<input type="button" value="Save User"/>		

# Processing forms : The JSP

```
<!-- addUser.jsp -->
```

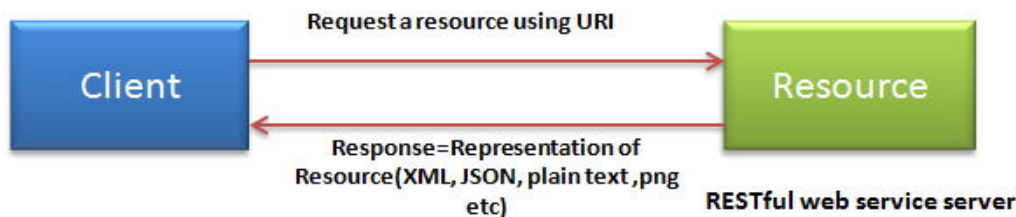
```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="userobj" action="processForm.obj">
<table cellpadding="0">
  <tr>
    <th><sf:label path="username">Username:</sf:label></th>
    <td><sf:input path="username" size="15" maxlength="15"/>
      <a style="color:red"><sf:errors path="username"/></a></td>
  </tr>
  <tr>
    <th><sf:label path="password">Password:</sf:label></th>
    <td><sf:password path="password" size="30" />
      <a style="color:red"><sf:errors path="password"/> </a>
    </td>
  </tr>
  ...
  <tr><td><input name="commit" type="submit" value="Save User" /></td> </tr>
</table>
</fieldset>
</sf:form>
```

# Validating input : declaring validation rules

```
public class User {  
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20  characters long.")  
    @Pattern(regexp = "[a-zA-Z0-9]+$", message = "Username must be alphanumeric with no spaces")  
    private String username;  
  
    @Size(min = 6, max = 20, message = "The password must be at least 6 characters long.")  
    private String password;  
  
    @Pattern(regexp = "[A-Za-z0-9]+@[A-Za-z0-9.-]+[.][A-Za-z]{2,4}", message = "Invalid email address.")  
    // @Email – alternate choice  
    private String email;  
  
    //getter and setter methods for all these properties  
}
```

# Spring MVC Framework and REST

- REST stands for REpresentational State Transfer .
  - It is an architectural principle based on top of HTTP to represent resources by doing operations on them.
  - It's a way of representing data and manipulating a resource that resides on the server.
  - REST web services solely depend on the HTTP methods; For each method, respective operations on a resource take place.



HTTP verb	Meaning in CRUD terms
POST	Create a new resource from the request data
GET	Read a resource
PUT	Update a resource from the request data
DELETE	Delete a resource

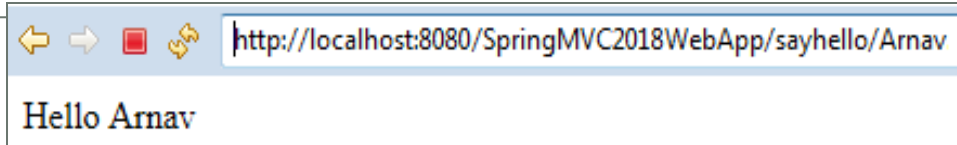
# Using the @RestController Annotation

- @RestController is used to define the RESTful web services.
  - It serves JSON, XML and custom response
  - Its a specialized version of the Controller ; does nothing more than add the @Controller and @ResponseBody annotations.

```
@RestController = @Controller + @ResponseBody
```

- Example:

```
@RestController
@RequestMapping("sayhello")
public class PathVariableDemo {
    @GetMapping(value = "/{name}")
    public String hello(@PathVariable String name) {
        return "Hello "+ name;
    }
}
```



# Difference between @Controller and @RestController

## @Controller

@RequestMapping("employees")

**public class EmpController{**

Emp emp = **new Emp();**

@GetMapping(value =("/{name}", produces = "application/json")

**public @ResponseBody Emp getEmpInJSON(@PathVariable String name){**

emp.setName(name);

emp.setEmail("emp1@myorg.com");

**return emp;**

}

@GetMapping(value =("/{name}.xml", produces = "application/xml")

**public @ResponseBody Emp getEmpInXML(@PathVariable String name){**

emp.setName(name);

emp.setEmail("emp1@myorg.com");

**return emp;**

}

}

localhost:8080/SpringRESTApp2018/employees/Shrilata.xml

```
<Employee>
  <email>emp1@myorg.com</email>
  <name>Shrilata</name>
</Employee>
```

localhost:8080/SpringRESTApp2018/employees/Shrilata

Apps Google PDF to Word conver... Converting File - Lu...

{"name":"Shrilata","email":"emp1@myorg.com"}

# Difference between @Controller and @RestController

## @RestController

@RequestMapping("restemployees")

```
public class EmpRestController {  
    Emp emp = new Emp();
```

@GetMapping(value =("/{name}", produces = "application/json")

```
public Emp getEmpInJSON(@PathVariable String name) {
```

```
    emp.setName(name);
```

```
    emp.setEmail("emp1@myorg.com");
```

```
    return emp;
```

```
}
```

@GetMapping(value =("/{name}.xml", produces = "application/xml")

```
public Emp getEmpInXML(@PathVariable String name) {
```

```
    emp.setName(name);
```

```
    emp.setEmail("emp1@myorg.com");
```

```
    return emp;
```

```
}
```

```
}
```

localhost:8080/SpringRESTApp2018/restemployees/Shrilata.xml

```
<Employee>  
  <email>emp1@myorg.com</email>  
  <name>Shrilata</name>  
</Employee>
```

localhost:8080/SpringRESTApp2018/restemployees/Shrilata

Apps Google PDF to Word conver... Converting File - Lu... Java

```
{"name": "Shrilata", "email": "emp1@myorg.com"}
```

@XmlElement(name = "Employee")

```
public class Emp {  
    String name, email;
```

```
    ...
```

```
}
```

- In order to serve JSON, we use Jackson library [jackson-databind.jar].
- For XML, we use Jackson XML extension [jackson-dataformat-xml.jar].
  - Mere presence of these libraries in classpath will trigger Spring to convert the output in required format.



@Repository

```
public class PersonDao {  
    List<Person> personlist = new ArrayList<>();
```

```
    public PersonDao() {  
        personlist.add(new Person("Anita", 50));  
        personlist.add(new Person("Sunita", 45));  
        personlist.add(new Person("Kavita", 30));  
    }
```

```
    public List<Person> getPersons() {  
        return personlist;  
    }
```

```
    public Person getPersonByName(String name) {  
        Person returnperson = null;  
        for (Person p : personlist) {  
            if (p.getPname().equals(name)) {  
                returnperson = p; break;  
            }  
        }  
        return returnperson;  
    }  
    public Person createPerson(Person p) {  
        personlist.add(p); return p;  
    }  
}
```

```
public class Person {  
    String pname;  
    int age;  
    //constructore  
    //get/set methods  
}
```

localhost:8080/SpringRESTApp2018/persons/

Apps Google PDF to Word conver... Converting File - Lu... Java spring integration

[{"pname": "Anita", "age": 50}, {"pname": "Sunita", "age": 45}, {"pname": "Kavita", "age": 30}]

localhost:8080/SpringRESTApp2018/persons/Sunita

Apps Google PDF to Word conver... Converting File - Lu...

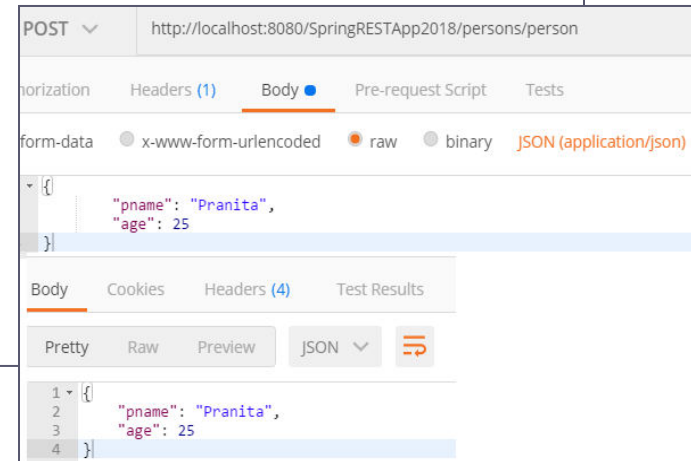
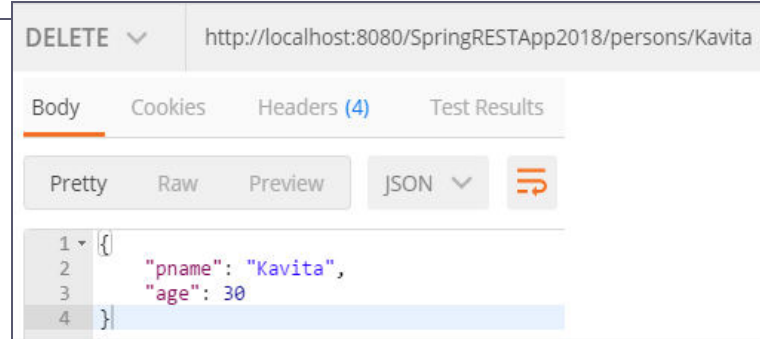
{"pname": "Sunita", "age": 45}

```
@RestController  
@RequestMapping("/persons")  
public class PersonController {  
    @Autowired  
    PersonDao personDao;  
  
    @GetMapping(value = "/", produces = "application/json")  
    public List<Person> getAll() {  
        return personDao.getPersons();  
    }  
  
    @GetMapping(value =("/{name}", produces = "application/json")  
    public Person getPersonByName(@PathVariable String name)  
        return personDao.getPersonByName(name);  
    }  
    ...  
}
```

```

@RestController
@RequestMapping("/persons")
public class PersonController {
    @PostMapping(value="/person", produces="application/json")
    public Person addPerson(@RequestBody Person person){
        return personDao.createPerson(person);
    }
    @DeleteMapping(value="/{name}", produces="application/json")
    public Person delPerson(@PathVariable String name){
        return personDao.deletePersonByName(name);
    }
    @PutMapping(value="/person", produces="application/json")
    public Person updatePerson(@RequestBody Person person){
        return personDao.updatePerson(person);
    }
}

```

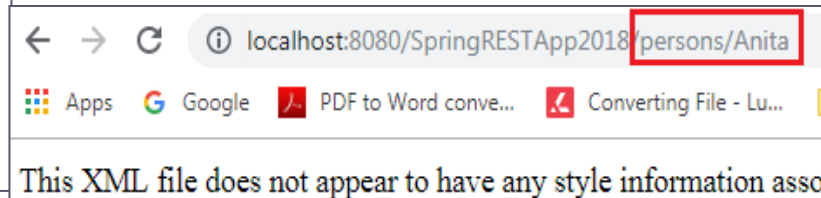


## Returning multiple formats

```

@GetMapping(value = "{name}", produces={
    MediaType.APPLICATION_JSON_VALUE,
    MediaType.APPLICATION_XML_VALUE
})
public Person getPersonByName(@PathVariable String name) {
    return personDao.getPersonByName(name);
}

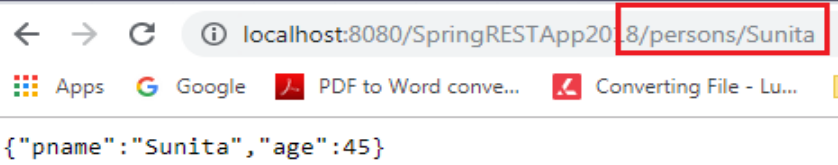
```



```

<person>
  <age>50</age>
  <pname>Anita</pname>
</person>

```



# SPRING BOOT (2.1.6)

---

# SpringBoot introduction

- From the Spring Boot website

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run."  
We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss.

- **Spring Boot is not a framework**

- It is a way to ease to create a stand-alone application with minimal or zero configurations.
- It provides defaults for code and annotation configuration to quick start new spring projects within no time.

# Spring Boot

- **Spring without Boot**

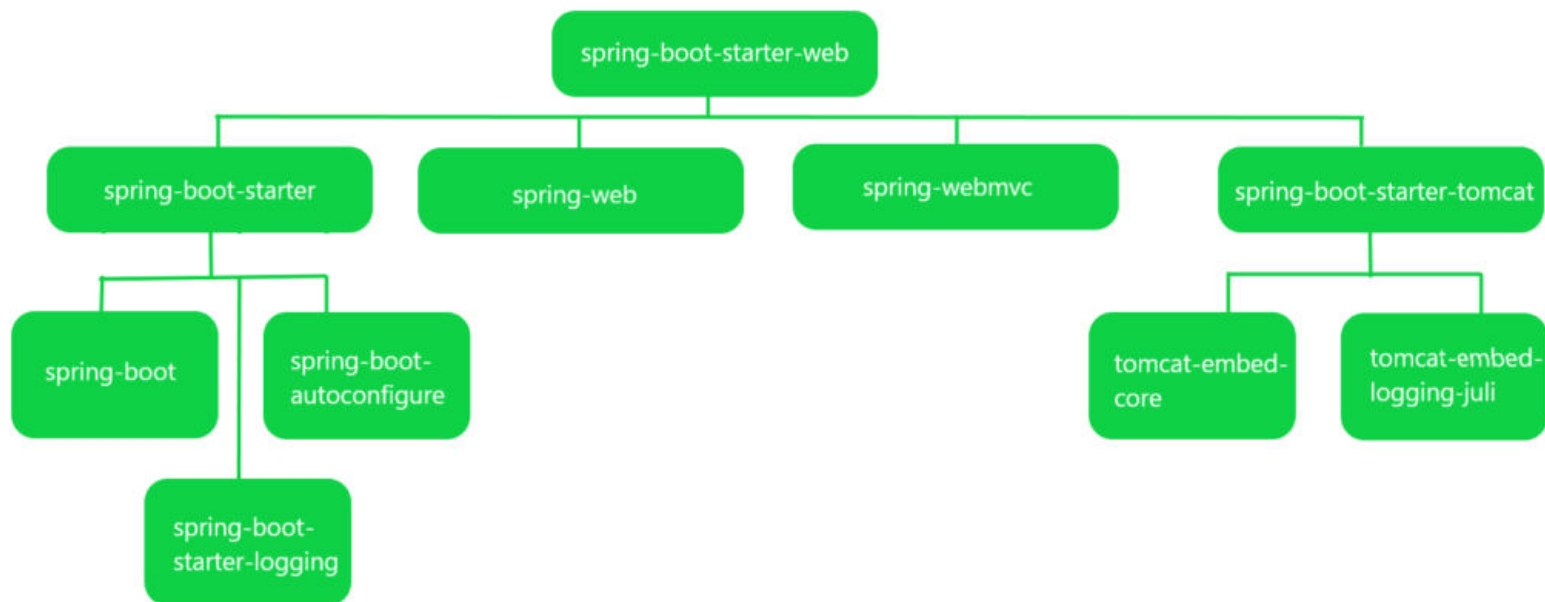
- Need to hunt for all the compatible libraries for the specific Spring version and add them
- Most of the time we configure the DataSource, JdbcTemplate, TransactionManager, DispatcherServlet, HandlerMapping, ViewResolver etc beans in same way
- We should always deploy in external container
- Problem with Spring component-scanning and autowiring is that its hard to see how all of the components are wired together

- **Spring with Boot**

- Spring Boot lets us develop Spring based application with very less configuration.
  - It provides a set of Starter Pom's or gradle build files which one can use to add required dependencies and also facilitate auto configuration.
  - Auto configuration for most of the commonly used beans like DataSource, JdbcTemplate, TransactionTemplate, DispatcherServlet, HandlerMapping, ViewResolver using customizable properties
  - Embedded Servlet container : spring-boot-starter-web pulls the spring-boot-starter-tomcat automatically which starts Tomcat as a embedded container; no external Tomcat server
  - By default, [Tomcat](#), [Jetty](#) and [Undertow](#) application servers come built-in with Boot
  - Spring Boot Actuators : actuators let us look inside of our running Boot application.

# Starters

- A project's dependencies are taken care of by the starters.
  - It does so by aggregating commonly used dependencies under a banner with a name
  - The developer names it in the dependency section of their project and the build specification transitively resolves the required dependencies issues.
  - All the required dependencies are automatically imported under that banner.



# Spring Boot AutoConfigurator

- Many Spring Boot developers always have their main class annotated with the following annotations:
  - `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
  - `@ComponentScan`: enable `@Component` scan on the package where the application is located
  - `@Configuration`: allow to register extra beans in the context or import additional configuration classes
- Since these annotations are frequently used together, Spring Boot provides a convenient `@SpringBootApplication` alternative.

```
@SpringBootApplication = @Configuration +  
                          @ComponentScan +  
                          @EnableAutoConfiguration
```

## `@SpringBootApplication`

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

This is the entry point to the entire Spring/Spring Boot system.

# Getting started with Spring Boot

- A Spring Boot project is just a regular Spring project that happens to leverage Spring Boot starters and auto-configuration.
- There are three approaches to start opinionated approach to create Spring Boot Applications:
  - Using the Spring Boot CLI Tool
    - Spring Boot CLI is a command line tool used for executing the groovy scripts
  - Using Spring Initializr (Website <http://start.spring.io/>)
    - It is an online Spring Boot application generator that generates Spring Boot projects.
  - Using your favorite IDE and Maven



```
class HelloWorld{
    public String sayHello(){
        return "Hello Spring Boot!";
    }
}
```

```
@SpringBootApplication
public class AppExample {

    @Bean
    public HelloWorld hello(){
        return new HelloWorld();
    }

    public static void main(String[] args) {
        ApplicationContext ctx =
            SpringApplication.run(AppExample.class, args);
        HelloWorld hw = ctx.getBean("hello", HelloWorld.class);
        System.out.println(hw.sayHello());
    }
}
```

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>
```

```

:: Spring Boot ::                (v2.1.6.RELEASE)

2019-07-23 23:52:20.120 INFO 22116 --- [
2019-07-23 23:52:20.122 INFO 22116 --- [
2019-07-23 23:52:20.588 INFO 22116 --- [
Hello Spring Boot!
```

# ApplicationRunner and CommandLineRunner

- Spring Boot allows you to execute code before your application starts.
  - It provides the ApplicationRunner and the CommandLineRunner interfaces that expose the run methods

```
@SpringBootApplication
public class RunDemo implements CommandLineRunner, ApplicationRunner {
    static final Logger log = LoggerFactory.getLogger(RunDemo.class);

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

    @Bean
    String info() { return "Simple String bean"; }

    @Autowired String info;

    @Override
    public void run(ApplicationArguments args) throws Exception {
        log.info("## > ApplicationRunner Implementation...");
        log.info("Accessing the Info bean: " + info);
    }

    @Override
    public void run(String... args) throws Exception {
        log.info("## > CommandLineRunner Implementation...");
        log.info("Accessing the Info bean: " + info);
        for (String arg : args) log.info(arg);
    }
}
```

```
## > ApplicationRunner Implementation...
Accessing the Info bean: Simple String bean
## > CommandLineRunner Implementation...
Accessing the Info bean: Simple String bean
```

# Configuring your Boot application

- Spring Boot applications can be configured in several ways: For eg, to set context root:

1. Using application.properties file : server.servlet.contextPath=/myapp

2. Via command line properties

\$ java -jar javadevjournal.jar --server.servlet.context-path=/myapp

3. Using Java System Property

System.setProperty("server.servlet.context-path", "/myapp")

4. Via Java code

```
@Component
public class AppCustomizer implements WebServerFactoryCustomizer {
    @Override
    public void customize(ConfigurableServletWebServerFactory factory) {
        factory.setContextPath("/javadevjournal");
    }
}
```

5. Using application.yml :

```
server:
  servlet:
    contextPath:/myapp
```

# application.properties

- Spring boot's "application.properties" file is auto detected without any spring based configurations.
  - Typically found inside "src/main/resources" directory.
  - There is no need to explicitly register a PropertySource, or provide a path to a property file.
  - Common Spring Boot properties and references to underlying classes that consume them:

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

- Examples of configuration in application.properties:
  - To change default context for Boot web application from "/" to "/myapp":  
`server.servlet.context-path=/myapp`
  - To configure the log file for your application : `logging.file=logs/my-log-file.log`
  - To change default port of embedded Tomcat : `server.port=9080`
  - To define your own custom properties: `person.name=Shrilata`
  - To update banner mode: `spring.main.banner-mode=off`

## Example-2: Stand alone app

```
@Component
public class Country {

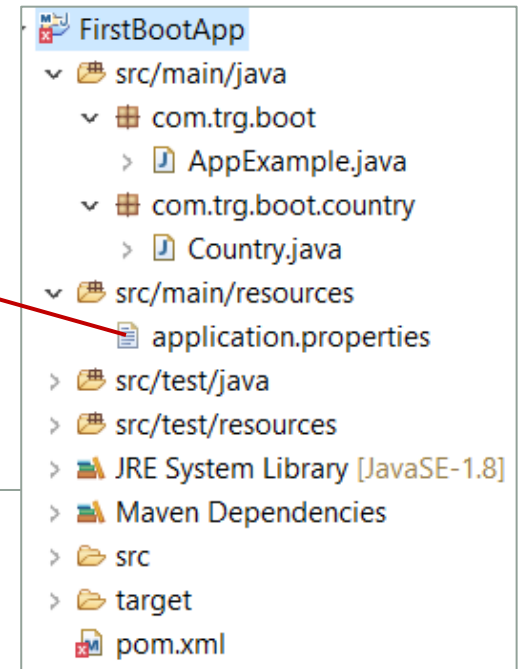
    @Value("${cname}")
    String countryName;

    //setter, getter methods
}
```

```
@SpringBootApplication
public class AppExample {

    public static void main(String[] args) {
        ApplicationContext ctx =
            SpringApplication.run(AppExample.class, args);
        HelloWorld hw = ctx.getBean("hello", HelloWorld.class);
        System.out.println(hw.sayHello());
        Country c = ctx.getBean("country", Country.class);
        System.out.println(c.getCountryName());
    }
}
```

```
#application.properties
cname=India
```



```

  ____
 /  _ \
| |_) |
|  _ <
| |_) |
|  __/
|_____|

:: Spring Boot ::

2019-07-25 01:14:54.515
2019-07-25 01:14:54.517
2019-07-25 01:14:58.527
Hello Spring Boot!
India

```

# Configuring with YAML

- [YAML](#) is a human-friendly data serialization standard but is mainly used for configuration files.
  - The SpringApplication class will automatically support YAML as an alternative to properties and it uses SnakeYAML library as the YAML parser.
  - Instead of having an application.properties in Spring, we can use the *application.yml* as our configuration file.

```
#application.yml
user:
  uname:
    shrilata
```

```
@Component
class Hello{

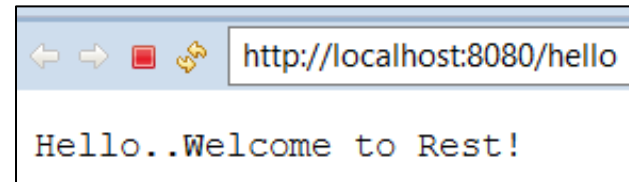
    @Value("${user.uname}")
    String msg;
    public void sayHello(){
        System.out.println("Hello Boot! " + msg);
    }
}
```

# Spring Boot Web Project : RestController : Example-1

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

```
@RestController
public class HelloWorldController {

    @RequestMapping("/hello")
    public String handler(){
        return "Hello..Welcome to Rest!";
    }
}
```



```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- The spring-boot-starter-web starter by default configures DispatcherServlet to the URL pattern "/" and adds Tomcat as the embedded servlet container, which runs on port 8080.
- The spring-boot-starter-web ensures all required dependencies to create Spring Web application (including REST) are included in your classpath, **it will also add tomcat-starter** as default server to run web application

# Spring Boot Web Project : RestController : Example-2

```
public class Course {  
    int id;  
    String name, desc;  
  
    // appropriate cons,getter,setter
```

```
@RestController  
public class CourseController {  
  
    @RequestMapping("/courses")  
    public List<Course> getTopics() {  
        return Arrays.asList(  
            new Course(101, "Spring", "Spring quickstart"),  
            new Course(102, "Java", "Java fundamentals"),  
            new Course(103, "NodeJS", "Node essentials"));  
    }  
}
```

```
SecondBootApplication  
└─ src/main/java  
    └─ com.trg.boot  
        ├── Application.java  
        └─ com.trg.boot.web  
            ├── HelloWorldController.java  
            ├── Topic.java  
            └─ TopicController.java  
└─ src/main/resources  
└─ src/test/java  
└─ src/test/resources  
└─ JRE System Library [JavaSE-1.8]  
└─ Maven Dependencies  
└─ src  
└─ target  
    └─ pom.xml
```

← → ↻ ⓘ localhost:8081/courses

```
[{"id":101,"name":"Spring","desc":"Spring quickstart"},  
{"id":102,"name":"Java","desc":"Java fundamentals"},  
{"id":103,"name":"NodeJS","desc":"Node essentials"}]
```



# RestController : Example-3

```
@Service
public class CourseService {

    public CourseService() {
        courses.add(new Course(101, "Spring", "Spring quickstart"));
        courses.add(new Course(102, "Java", "Java fundamentals"));
        courses.add(new Course(103, "NodeJS", "Node essentials"));
    }

    List<Course> courses = new ArrayList<>();

    public List<Course> getCourses() {
        return courses;
    }

    public Course getCourseById(int id){
        for(Course c : courses){
            if(c.getId()==id)
                return c;
        }
        return null;
    }

    public boolean addCourse(Course course){
        return courses.add(course);
    }
}
```

```
public void updateCourse(int id, Course course){
    System.out.println(id);
    for(int i=0; i< courses.size() ;i++){
        if(courses.get(i).getId()==id){
            courses.set(i, course);
            break;
        }
    }
}

public void deleteCourse(int id){

    Iterator<Course> it = courses.iterator();
    while(it.hasNext()){
        if(it.next().getId()==id)
            it.remove();
    }
}
```

# RestController : Example-3

```
@RestController
public class CourseController {

    @Autowired
    CourseService courseService;

    @GetMapping("/courses")
    public List<Course> getCourses() {
        return courseService.getCourses();
    }

    @GetMapping("/courses/{id}")
    public Course getById(@PathVariable int id) {
        return courseService.getCourseById(id);
    }

    @PostMapping("/courses")
    public void addCourse(@RequestBody Course course) {
        courseService.addCourse(course);
    }

    @PutMapping("/courses/{id}")
    public void updateCourse(@PathVariable int id,
        @RequestBody Course course){
        courseService.updateCourse(id, course);
    }

    @DeleteMapping("/courses/{id}")
    public void delCourse(@PathVariable int id) {
        courseService.deleteCourse(id);
    }
}
```

POST http://localhost:8080/courses

Params Authorization Headers (9) Body Pre-re

● none ● form-data ● x-www-form-urlencoded ● raw

```
1 {
2   "id": 104,
3   "name": "JSON",
4   "desc": "json quickstart fundamentals"
5 }
```

GET http://localhost:8080/courses

Pretty Raw Preview JSON

```
1 [
2   {
3     "id": 101,
4     "name": "Spring",
5     "desc": "Spring quickstart"
6   },
7   {
8     "id": 102,
9     "name": "Java",
10    "desc": "Java fundamentals"
11  },
12  {
13    "id": 103,
14    "name": "NodeJS",
15    "desc": "Node essentials"
16  }
17 ]
```

# Using Spring Initializr

- Spring Initializr is an online Spring Boot application generator.
- Goto <https://start.spring.io/>

The screenshot shows the Spring Initializr web application generator interface. The interface is divided into several sections:

- Project:** Includes tabs for **Maven Project** and **Gradle Project**.
- Language:** Includes tabs for **Java**, **Kotlin**, and **Groovy**.
- Spring Boot:** Includes version selection tabs: **2.2.0 M4**, **2.2.0 (SNAPSHOT)**, **2.1.7 (SNAPSHOT)**, **2.1.6** (selected), and **1.5.21**.
- Project Metadata:** Includes input fields for **Group** (containing `com.trg.boot`) and **Artifact** (containing `BootInitializrDemo`).
- Options:** A section for additional project options.
- Dependencies:** Includes a search bar and a list of selected dependencies. The selected dependency is **Spring Web Starter**, which includes RESTful applications using Spring MVC and uses Apache Tomcat as the default embedded container.

At the bottom of the interface, there are two buttons: **Generate the project - Ctrl + G** and **Explore the project - Ctrl + Space**.

# Using JdbcTemplate with Spring Boot

- Create a Spring Boot Maven-based project and add the spring-boot-starter-jdbc module

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

- With this module, you get the following autoconfiguration features:
  - It transitively pulls tomcat-jdbc-{version}.jar, which is used to configure the **DataSource bean**.
  - If you have not defined a DataSource bean explicitly and if you have any embedded database drivers in the classpath, such as H2, HSQL, or Derby, then Spring Boot will automatically register the DataSource bean using the in-memory database settings.
  - If you haven't registered any of the following beans, then Spring Boot will register them automatically : PlatformTransactionManager (DataSourceTransactionManager), **JdbcTemplate, NamedParameterJdbcTemplate**
  - You can have the schema.sql and data.sql files in the root classpath, which Spring Boot will automatically use to initialize the database.

# Example : JdbcTemplate

```
FourthDBBootApp
├── src/main/java
│   ├── com.trg.boot
│   │   ├── Application.java
│   │   ├── com.trg.boot.db.book
│   │   │   ├── Book.java
│   │   │   ├── com.trg.boot.db.controller
│   │   │   ├── com.trg.boot.db.dao
│   │   │   │   ├── BookDao.java
│   │   │   │   ├── BookDaoImpl.java
│   │   │   │   └── BookMapper.java
│   │   └── src/main/resources
│   │       ├── application.properties
│   │       ├── data.sql
│   │       └── schema.sql
│   ├── src/test/java
│   ├── src/test/resources
│   ├── JRE System Library [JavaSE-1.8]
│   ├── Maven Dependencies
│   └── lib
│       ├── ojdbc6.jar
│       ├── src
│       ├── target
│       └── pom.xml
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <!-- Exclude the default Tomcat connection pool -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
      <exclusion>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-jdbc</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0</version>
    <scope>system</scope>
    <systemPath>${basedir}/lib/ojdbc6.jar</systemPath>
  </dependency>

  <!-- Common DBCP2 connection pool -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
  </dependency>
</dependencies>
```

## Example : JdbcTemplate : Dao

- Spring Boot will register a JdbcTemplate bean automatically, just inject it into your bean.

```
@Repository("bookDao")
public class BookDaoImpl implements BookDao {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public Book getBookById(int id) {
        String sql = "select * from book where bookid=?";
        Book book = (Book) jdbcTemplate.queryForObject(sql,
            new Object[] { id }, new BookMapper());

        return book;
    }

    public List<Book> getBooks() {
        String SQL = "select * from book";
        List<Book> books = jdbcTemplate.query(SQL, new BookMapper());
        return books;
    }
}
```

```
public class Book {
    int id;
    String bname, author;
    double price;

    //appropriate cons, getter, setter
}
```

# Example : JdbcTemplate : Database Initialization

- Spring Boot automatically creates the schema of an embedded DataSource.
  - `spring.datasource.initialization-mode=always`
  - If set to “always”, Spring Boot will use the `schema.sql` and `data.sql` files in the root classpath to initialize the database.

```
CREATE TABLE Book(  
  BOOKID NUMBER(4) NOT NULL,  
  BOOKNAME VARCHAR2(20) NOT NULL,  
  AUTHOR VARCHAR2(20) NOT NULL,  
  PRICE NUMBER(6,2) NOT NULL,  
  CONSTRAINT BOOK_PK PRIMARY KEY (BOOKID)  
);
```

schema.sql

data.sql

```
insert into book(bookid, bookname, author, price) values (101, 'Core Java', 'Cay Horstman', 450.0);  
insert into book(bookid, bookname, author, price) values (102, 'Core Servlets', 'Chris Brown', 400.0);  
insert into book(bookid, bookname, author, price) values (103, 'Spring Boot', 'Alex Antonov', 300.0);
```

# Example : JdbcTemplate : Database Initialization

- Configure Oracle and dbcp2 settings.

Application.properties

```
# Set true for first time db initialization.
#To always initialize the DataSource regardless of its type
# After running first time, comment this line
spring.datasource.initialization-mode=always

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=oracle
spring.datasource.password=oracle123
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

# dbcp2 settings
# spring.datasource.dbcp2.*

spring.datasource.dbcp2.initial-size=7
spring.datasource.dbcp2.max-total=20
spring.datasource.dbcp2.pool-prepared-statements=true
```



# Example : JdbcTemplate : The Application class

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    DataSource dataSource;

    @Autowired
    BookDao bookDao;

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

    @Override
    public void run(String... args) throws Exception {

        System.out.println("DATASOURCE = " + dataSource);
        System.out.println("Display all books...");
        List<Book> list = bookDao.getBooks();
        list.forEach(x -> System.out.println(x));

        System.out.println("Display book with id 102");
        Book b = bookDao.getBookById(102);
        System.out.println(b);
    }
}
```

```
Display all books...
Book [id=101, bname=Core Java, author=Cay Horstman, price=450.0]
Book [id=102, bname=Core Servlets, author=Chris Brown, price=400.0]
Book [id=103, bname=Spring Boot, author=Alex Antonov, price=300.0]
Display book with id 102
Book [id=102, bname=Core Servlets, author=Chris Brown, price=400.0]
```

# Example – Spring Boot MVC Project

```
@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public String sayHello(){
        return "hello";
    }
}
```

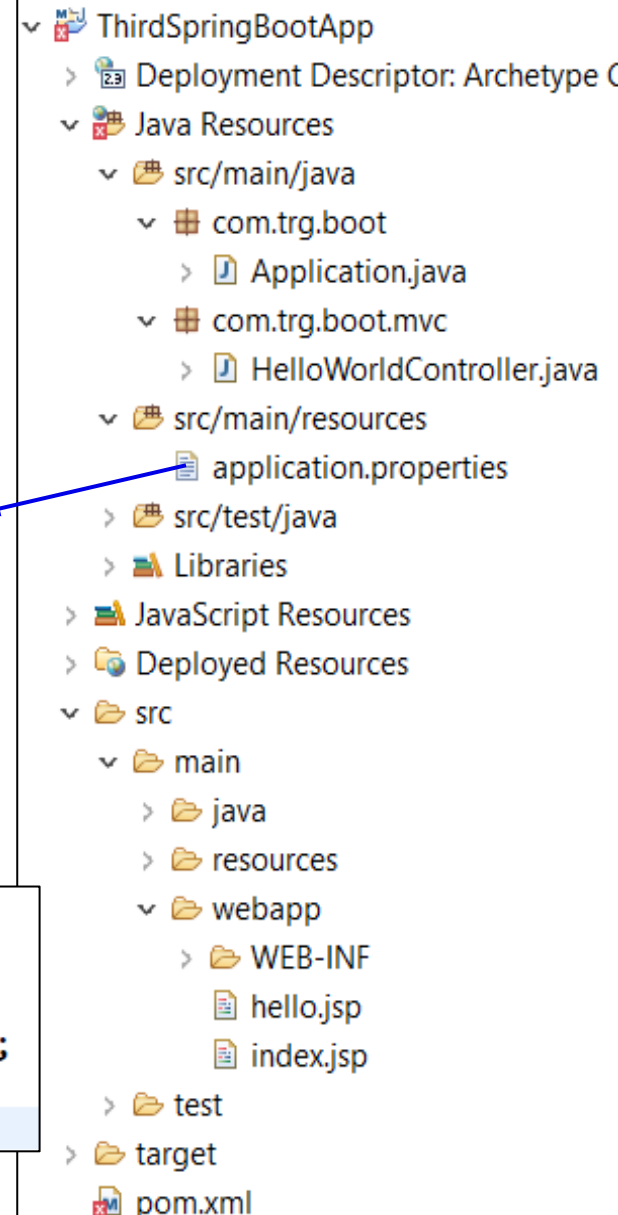
```
<!-- hello.jsp -->
<!DOCTYPE html>
<html>
<body>
<h1> Welcome to Spring Boot MVC!</h1>
</body>
</html>
```

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

```
#application.properties
server.port=8081
spring.mvc.view.prefix: /
spring.mvc.view.suffix: .jsp
```

http://localhost:8081/hello

**Welcome to Boot MVC!**



# Passing model data to JSP

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
```

```
<!-- JSTL for JSP -->
```

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

```
<!-- For JSP compilation -->
```

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>
</dependencies>
```

```
<!-- hello.jsp -->
<!DOCTYPE html>
<html>
<body>
<h1> ${message}</h1>
</body>
</html>
```

```
@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public String sayHello(Model model){
        model.addAttribute("message", "Welcome to Spring Boot MVC!");
        return "hello";
    }
}
```

To add styling to web pages either:

1. Add the Bootstrap WebJars dependency
2. Create the styles.css stylesheet in the src/main/resources/static/css folder

```
<dependency>
<groupId>org.webjars.bower</groupId>
<artifactId>bootstrap</artifactId>
<version>3.3.7</version>
</dependency>
```

```
body{
background-color: #A7A5A4;
padding-top: 50px;
}
```