

ADVANCED JAVA & SPRING BOOT

Marko Šutić, Marija Jurinec

WIFI

- Business club 5 Guest
- password: Bguest5!

PREREQUISITES

- IntelliJ community edition
- Exercises: <https://github.com/mithril-eu/zaba-java-advanced-code>

SETUP

- JAVA_HOME environment variable
- Java 17
 - java -version

LEARNING OBJECTIVES

- Introduction to Functional Programming
- Work through advanced concepts in Java programming language
- Web Programming with Java
- Spring Framework and Spring Boot

JAVA I/O CONCEPTS

JAVA I/O API

- Tools your application needs to access information from the outside

JAVA I/O, JAVA NIO AND JAVA NIO2

- Java I/O API was created in the mid-90s along with the first versions of the JDK.
- 2002, with Java SE 1.4, Java NIO
- 2011, with Java SE 7, Java NIO2

ACCESSING A FILE

- two main concepts in Java I/O:
 - locating the resource you need to access (it can be a file or a network resource)
 - opening a stream to this resource
- two ways to access files in the Java I/O API:
 - File class
 - Path interface

UNDERSTANDING I/O STREAMS

- I/O Stream represents an input source or an output destination
- Stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and object
- I/O streams are a different concept than the streams from the Stream API introduced in Java SE 8 !

UNDERSTANDING I/O STREAMS 2

- Java I/O API defines two kinds of content for a resource:
 - character content (text file, XML, JSON...)
 - byte content (image, video...)
- two operations:
 - reading
 - writing

UNDERSTANDING I/O STREAMS 3

Type	Reading	Writing
Streams of character	Reader	Writer
Streams of bytes	InputStream	OutputStream

WORKING WITH PATHS API

CREATING PATH OBJECTS IN JAVA

- Starting with Java 11: Use `Path.of()`
- Pre Java-11: Use `Paths.get()`

```
1 Path path = Path.of("c:\\\\folder\\\\child-folder\\\\readme.txt");
2 /* OR */
3 Path path = Paths.get("c:/folder/child-folder/readme.txt"); // Java < 11
```

DIFFERENT WAYS TO CREATE PATHS

```
1 /* Multiple string parameters */
2 path = Path.of("c:", "folder", "child-folder", "readme.txt");
3
4 /* Using resolve() for child paths */
5 path = Path.of("c:", "folder", "child-folder").resolve("readme.txt");
6
7 /* Using URI */
8 path = Path.of(new URI("file:///c:/folder/child-folder/readme.txt"));
```

IMPORTANT CONSIDERATIONS

- Path object creation does **not** verify file existence
 - It's just a reference to a **potential** file
 - Separate verification needed
- Version Compatibility
 - Java 11+: Use `Path.of()`
 - Java 7-10: Use `Paths.get()`
 - `Paths.get()` redirects to `Path.of()` in Java 11+

COMMON FILE OPERATIONS

BASIC FILE OPERATIONS

- Main class: `java.nio.file.Files`
- Contains static utility methods for file operations

```
1 /* Check if file exists */
2 boolean exists = Files.exists(path);
3
4 /* Get last modified date */
5 FileTime lastModifiedTime = Files.getLastModifiedTime(path);
6
7 /* Compare files (Java 12+) */
8 long mismatch = Files.mismatch(path1, path2);
```

FILE METADATA OPERATIONS

```
1 /* Get file owner */
2 UserPrincipal owner = Files.getOwner(path);
3
4 /* Get POSIX permissions (Unix-like systems only) */
5 Set<PosixFilePermission> permissions = Files.getPosixFilePermissions(path);
```

WORKING WITH TEMPORARY FILES

```
1 /* Create temp file in default temp directory */
2 Path tempFile1 = Files.createTempFile("prefix", ".jpg");
3
4 /* Create temp file in custom directory */
5 Path tempFile2 = Files.createTempFile(
6     parentPath, "prefix", ".jpg");
7
8 /* Create temporary directory */
9 Path tmpDirectory = Files.createTempDirectory("prefix");
```

CREATING FILES AND DIRECTORIES

```
1 /* Create complete directory structure */
2 Path newDirectory = Files.createDirectories(path.resolve("some/new/dir"));
3
4 /* Create new empty file */
5 Path newFile = Files.createFile(newDirectory.resolve("empty.txt"));
```

WRITING & READING FILES

WRITING STRINGS (JAVA 11+)

```
1 /* Write UTF-8 string (default) */
2 Path utfFile = Files.createTempFile("some", ".txt");
3 Files.writeString(utfFile, "this is my string č č ž");
4
5 /* Write with specific charset */
6 Path isoFile = Files.createTempFile("some", ".txt");
7 Files.writeString(isoFile, "this is my string č č ž", Charset.forName("windows-1250"));
```

WRITING BYTES & ADVANCED OPTIONS

```
1 /* Write bytes directly */
2 Files.write(path, "content".getBytes(StandardCharsets.UTF_8));
3
4 /* Write with specific options */
5 Files.writeString(path, "content", StandardCharsets.UTF_8,
6     StandardOpenOption.CREATE,
7     StandardOpenOption.TRUNCATE_EXISTING,
8     StandardOpenOption.WRITE);
```

USING WRITERS AND STREAMS

```
1 /* Using BufferedWriter */
2 try (BufferedWriter writer = Files.newBufferedWriter(path)) {
3     /* handle writer */
4 }
5
6 /* Using OutputStream */
7 try (OutputStream os = Files.newOutputStream(path)) {
8     /* handle outputstream */
9 }
```

READING FILES (JAVA 11+)

```
1 /* Read as UTF-8 string (default) */
2 String content = Files.readString(path);
3
4 /* Read with specific charset */
5 content = Files.readString(path, Charset.forName("windows-1250"));
6
7 /* Read as bytes */
8 content = new String(Files.readAllBytes(path),
9         StandardCharsets.UTF_8);
```

READING WITH READERS AND STREAMS

```
1 /* Using BufferedReader */
2 try (BufferedReader reader = Files.newBufferedReader(path)) {
3     /* handle reader */
4 }
5
6 /* Using InputStream */
7 try (InputStream is = Files.newInputStream(path)) {
8     /* handle inputstream */
9 }
```

IMPORTANT: FILE ENCODINGS

- **Always** use explicit encodings
- New Java 11 methods default to UTF-8
- Older methods use platform-specific encoding

MOVING, DELETING & LISTING FILES

MOVING FILES - COMMON PITFALLS

```
1 /* WRONG - Can't move directly to directory */
2 Files.move(utfFile, Path.of("c:\\folder")); /* throws exception */
3
4 /* CORRECT - Must specify full target path */
5 Files.move(utfFile, Path.of("c:\\folder").resolve(utfFile.getFileName().toString()));
```

MOVING FILES - OPTIONS

```
1 /* Replace existing file if present */
2 Files.move(sourceFile, targetPath, StandardCopyOption.REPLACE_EXISTING);
3
4 /* Ensure atomic move operation */
5 Files.move(sourceFile, targetPath, StandardCopyOption.ATOMIC_MOVE);
```

DELETING FILES AND DIRECTORIES

```
1 /* Simple file or empty directory deletion */
2 Files.delete(path);
3
4 /* Delete non-empty directory recursively */
5 try (Stream<Path> walk = Files.walk(tmpDir)) {
6     walk.sorted(Comparator.reverseOrder())
7         .forEach(path -> {
8             try {
9                 Files.delete(path);
10            } catch (IOException e) {
11                /* Handle deletion error */
12            }
13        });
14 }
```

LISTING FILES - SAME DIRECTORY LEVEL

```
1 /* List all files */
2 try (var files = Files.list(tmpDirectory)) {
3     files.forEach(System.out::println);
4 }
5
6 /* List with pattern matching */
7 try (var files = Files.newDirectoryStream(tmpDirectory, "*.txt")) {
8     files.forEach(System.out::println);
9 }
```

RECURSIVE FILE LISTING

```
1 /* Walk entire directory tree */
2 try (var files = Files.walk(tmpDirectory)) {
3     files.forEach(System.out::println);
4 }
```

WATCHING FILES & DIRECTORIES

JAVA WATCHSERVICE (JAVA 7+)

- Native file system events (except MacOS)
- Low-level API
- Part of `java.nio.file` package

JAVA WATCHSERVICE

```
1 /* Create WatchService */
2 WatchService watcher = FileSystems.getDefault().newWatchService();
3
4 /* Register directory for events */
5 Path dir = Path.of("c:\\\\someDir\\\\");
6 dir.register(watcher,
7     ENTRY_CREATE,
8     ENTRY_DELETE,
9     ENTRY_MODIFY);
10
11 /* Process events */
12 for (;;) {
13     WatchKey key = watcher.take();
14
15     for (WatchEvent<?> event: key.pollEvents()) {
16         /* Handle event */
17         Path filename = ev.context();
18         Path changedFile = dir.resolve(filename);
19     }
20
21     if (!key.reset()) break;
22 }
```

WATCHSERVICE CONSIDERATIONS

- Multiple events for single operation:
 - Content update
 - Timestamp update
- Complex file operations:
 - IDEs use temporary files
 - Multiple updates per save
 - File deletions and recreations

APACHE COMMONS-IO ALTERNATIVE

```
1 /* Create file monitor */
2 FileAlterationObserver observer =
3     new FileAlterationObserver(folder);
4 FileAlterationMonitor monitor =
5     new FileAlterationMonitor(pollingInterval);
6
7 /* Configure listener */
8 FileAlterationListener listener =
9     new FileAlterationListenerAdaptor() {
10     @Override
11     public void onFileCreate(File file) {
12         /* Handle file creation */
13     }
14
15     @Override
16     public void onFileDelete(File file) {
17         /* Handle file deletion */
18     }
19 };
```

COMMONS-IO VS WATCHSERVICE

Table 1. Comparing File Watching Options

Feature	WatchService	Commons-IO
Implementation	Native events*	Polling based
API Style	Low-level	High-level
File API	java.nio.file	java.io
Platform Support	All**	All

IN-MEMORY FILE SYSTEMS

WHAT ARE IN-MEMORY FILE SYSTEMS?

- Virtual file systems that operate purely in memory
- No disk I/O involved
- Perfect for testing scenarios
- Faster than physical file operations

MEMORY FILE SYSTEM LIBRARY

```
1 /* Create OS-specific in-memory file system */
2 try (FileSystem fileSystem = MemoryFileSystemBuilder.newMacOs().build()) {
3
4     /* Create and write to in-memory file */
5     Path inMemoryFile = fileSystem.getPath("/somefile.txt");
6     Files.writeString(inMemoryFile, "Hello World");
7
8     /* Read from in-memory file */
9     System.out.println(Files.readString(inMemoryFile));
10 }
```

KEY DIFFERENCES FROM PHYSICAL FILES

- Path creation:
 - Use `fileSystem.getPath()` instead of `Path.of()`
 - Don't use `Paths.get()`
- Available OS-specific builders:
 - `newWindows()`
 - `newLinux()`
 - `newMacOs()`

BENEFITS FOR TESTING

- No filesystem cleanup needed
- Faster test execution
- Isolated test environment
- No risk of interfering with real files

LAMBDA EXPRESSIONS

LAMBDA EXPRESSION

```
1 (parameter list) -> { body of the lambda expression }
```

```
1 BiFunction<Integer, Integer, Boolean> isDivisible = (x, y) -> x % y == 0;
```

```
1 /* if it has only one argument "()" are optional */
2 Function<Integer, Integer> adder1 = x -> x + 1;
3
4 /* with type inference */
5 Function<Integer, Integer> mult2 = (Integer x) -> x * 2;
6
7 /* with multiple statements */
8 Function<Integer, Integer> adder5 = (x) -> {
9     x += 2;
10    x += 3;
11    return x;
12 };
```

INVOKING LAMBDA EXPRESSIONS

```
1 BiFunction<Integer, Integer, Boolean> isDivisible = (x, y) -> x % y == 0;  
2 boolean result4Div2 = isDivisible.apply(4, 2); /* true */  
3 boolean result3Div5 = isDivisible.apply(3, 5); /* false */
```

PASSING LAMBDA EXPRESSIONS TO METHODS

```
1 private static void printResultOfLambda(Function<String, Integer> function) {  
2     System.out.println(function.apply("HAPPY NEW YEAR 3000!"));  
3 }
```

```
1 /* it returns the length of a string */  
2 Function<String, Integer> f = s -> s.length();  
3 printResultOfLambda(f); /* it prints 20 */
```

PASSING LAMBDA EXPRESSIONS TO METHODS

```
1 /* It prints the number of digits: 4 */
2 printResultOfLambda(s -> {
3     int count = 0;
4     for (char c : s.toCharArray()) {
5         if (Character.isDigit(c)) {
6             count++;
7         }
8     }
9     return count;
10});
```

CLOSURES

```
1 final String hello = "Hello, ";
2 Function<String, String> helloFunction = (name) -> hello + name;
3
4 System.out.println(helloFunction.apply("John"));
5 System.out.println(helloFunction.apply("Anastasia"));
```

```
1 Hello, John
2 Hello, Anastasia
```

```
1 int constant = 100;
2 Function<Integer, Integer> adder100 = x -> x + constant;
3
4 System.out.println(adder100.apply(200)); /* 300 */
5 System.out.println(adder100.apply(300)); /* 400 */
```

FUNCTIONAL INTERFACES

```
1 @FunctionalInterface  
2 interface Func<T, R> {  
3     R apply(T val); /* single abstract method */  
4 }
```

FUNCTIONAL INTERFACES

```
1 @FunctionalInterface
2 interface Func<T, R> {
3     R apply(T val);
4
5     static void doNothingStatic() { }
6
7     default void doNothingByDefault() { }
8 }
```

IMPLEMENTING FUNCTIONAL INTERFACES

1. Anonymous classes

```
1 Func<Long, Long> square = new Func<Long, Long>() {
2     @Override
3     public Long apply(Long val) {
4         return val * val;
5     }
6 };
7
8 long val = square.apply(10L); /* the result is 100L */
```

IMPLEMENTING FUNCTIONAL INTERFACES

2.Lambda expressions

```
1 Func<Long, Long> square = val -> val * val; /* the lambda expression */
2
3 long val = square.apply(10L); /* the result is 100L */
```

IMPLEMENTING FUNCTIONAL INTERFACES

3. Method references

```
1 class Functions {  
2  
3     public static long square(long val) {  
4         return val * val;  
5     }  
6 }
```

```
1 Func<Long, Long> square = Functions::square;
```

METHOD REFERENCES

```
1 objectOrClass :: methodName
```

```
1 BiFunction<Integer, Integer, Integer> max = Integer::max;
```

```
1 System.out.println(max.apply(50, 70)); // 70
```

```
1 BiFunction<Integer, Integer, Integer> max = (x, y) -> Integer.max(x, y);
```

KINDS OF METHOD REFERENCES

1. Reference to a static method

```
1 ClassName :: staticMethodName
```

```
1 Function<Double, Double> sqrt = Math::sqrt;
```

```
1 sqrt.apply(100.0d); // the result is 10.0d
```

```
1 Function<Double, Double> sqrt = x -> Math.sqrt(x);
```

KINDS OF METHOD REFERENCES

2. Reference to an instance method of an object

```
1  objectName :: instanceMethodName
```

```
1  String whatsGoingOnText = "What's going on here?";  
2  
3  Function<String, Integer> indexWithinWhatsGoingOnText = whatsGoingOnText::indexOf;
```

```
1  System.out.println(indexWithinWhatsGoingOnText.apply("going")); /* 7 */  
2  System.out.println(indexWithinWhatsGoingOnText.apply("Hi")); /* -1 */
```

```
1  Function<String, Integer> indexWithinWhatsGoingOnText = string -> whatsGoingOnText.indexOf(string);
```

KINDS OF METHOD REFERENCES

3. Reference to an instance method of an object of a particular type

```
1 ClassName :: instanceMethodName
```

```
1 Function<Long, Double> converter = Long::doubleValue;
```

```
1 converter.apply(100L); /* the result is 100.0d */
2 converter.apply(200L); /* the result is 200.0d */
```

```
1 Function<Long, Double> converter = val -> val.doubleValue();
```

KINDS OF METHOD REFERENCES

4. Reference to a constructor

```
1 ClassName :: new
```

```
1 class Person {  
2     String name;  
3  
4     public Person(String name) {  
5         this.name = name;  
6     }  
7 }
```

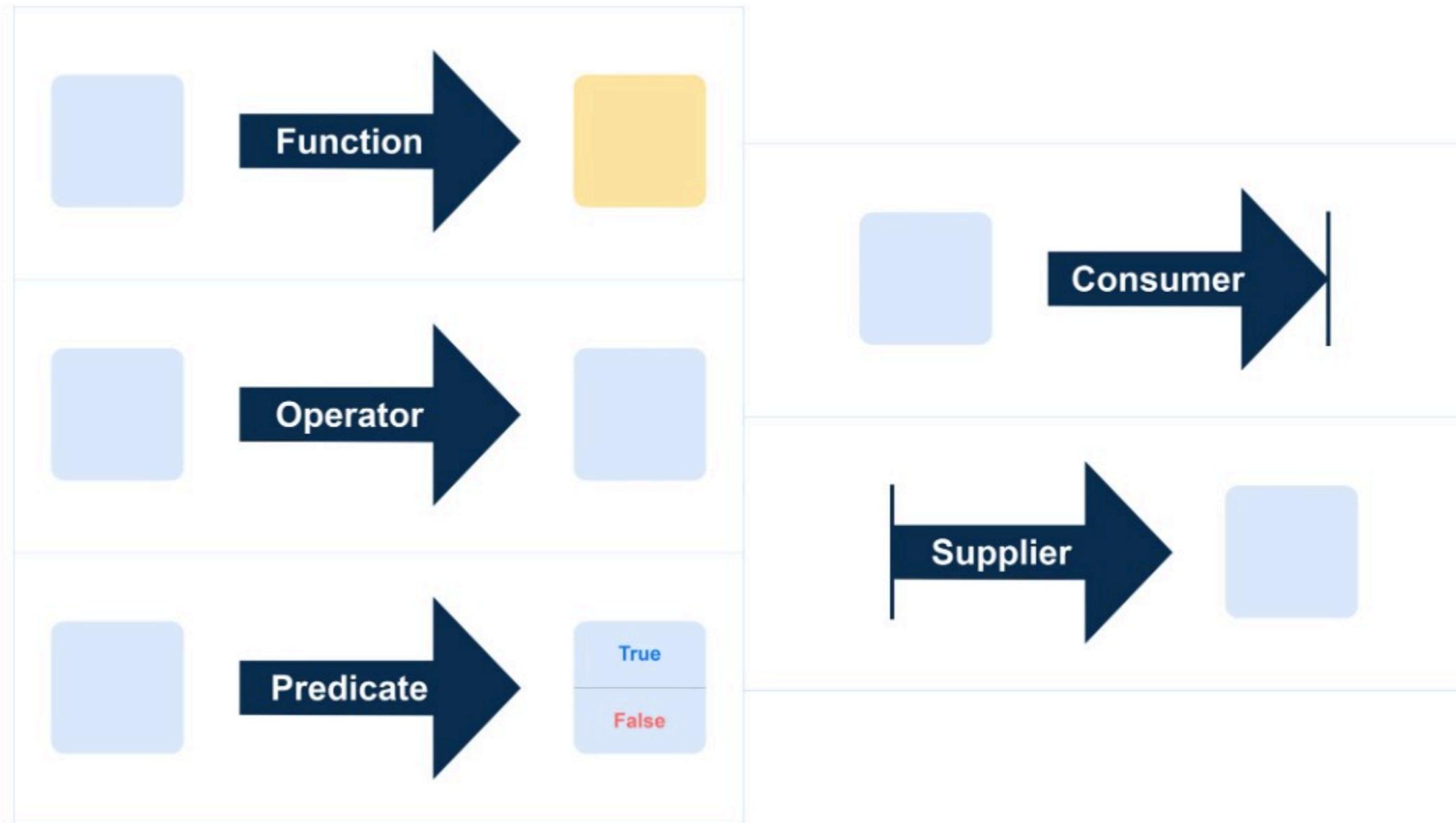
```
1 Function<String, Person> personGenerator = Person::new;
```

```
1 Person johnFoster = personGenerator.apply("John Foster"); // we have a John Foster object
```

```
1 Function<String, Person> personGenerator = name -> new Person(name);
```

FUNCTIONS

FUNCTIONAL INTERFACES



STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Functions

```
1 /* String to Integer function */
2 Function<String, Integer> converter = Integer::parseInt;
3 converter.apply("1000"); /* the result is 1000 (Integer) */
4
5 /* String to int function */
6ToIntFunction<String> anotherConverter = Integer::parseInt;
7 anotherConverter.applyAsInt("2000"); /* the result is 2000 (int) */
8
9 /* (Integer, Integer) to Integer function */
10 BiFunction<Integer, Integer, Integer> sumFunction = (a, b) -> a + b;
11 sumFunction.apply(2, 3); /* it returns 5 (Integer) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Operators

```
1 /* Long to Long multiplier */
2 UnaryOperator<Long> longMultiplier = val -> 100_000 * val;
3 longMultiplier.apply(2L); /* the result is 200_000L (Long) */
4
5 /* int to int operator */
6 IntUnaryOperator intMultiplier = val -> 100 * val;
7 intMultiplier.applyAsInt(10); /* the result is 1000 (int) */
8
9 /* (String, String) to String operator */
10 BinaryOperator<String> appender = (str1, str2) -> str1 + str2;
11 appender.apply("str1", "str2"); /* the result is "str1str2" */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Predicates

```
1 /* Character to boolean predicate */
2 Predicate<Character> isDigit = Character::isDigit;
3 isDigit.test('h'); /* the result is false (boolean) */
4
5 /* int to boolean predicate */
6 IntPredicate isEven = val -> val % 2 == 0;
7 isEven.test(10); /* the result is true (boolean) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Suppliers

```
1 Supplier<String> stringSupplier = () -> "Hello";
2 stringSupplier.get(); /* the result is "Hello" (String) */
3
4 BooleanSupplier booleanSupplier = () -> true;
5 booleanSupplier.getAsBoolean(); /* the result is true (boolean) */
6
7 IntSupplier intSupplier = () -> 33;
8 intSupplier.getAsInt(); /* the result is 33 (int) */
```

STANDARD FUNCTIONAL INTERFACES WITH EXAMPLES

Consumers

```
1 /* it prints a given string */
2 Consumer<String> printer = System.out::println;
3 printer.accept("!!!"); /* It prints "!!!" */
```

FUNCTION COMPOSITION

Two default methods:

- `f.compose(g).apply(x)` is the same as `f(g(x))`
- `f.andThen(g).apply(x)` is the same as `g(f(x))`

```
1 Function<Integer, Integer> adder = x -> x + 10;
2 Function<Integer, Integer> multiplier = x -> x * 5;
3
4 /* compose: adder(multiplier(5)) */
5 System.out.println("result: " + adder.compose(multiplier).apply(5));
6
7 /* andThen: multiplier(add(5)) */
8 System.out.println("result: " + adder.andThen(multiplier).apply(5));
```

```
1 result: 35
2 result: 75
```

COMPOSING PREDICATES

```
1 IntPredicate isOdd = n -> n % 2 != 0; /* it's true for 1, 3, 5, 7, 9, 11 and so on */
2
3 System.out.println(isOdd.test(10)); /* prints "false" */
4 System.out.println(isOdd.test(11)); /* prints "true" */
5
6 IntPredicate lessThan11 = n -> n < 11; /* it's true for all numbers < 11 */
7
8 System.out.println(lessThan11.test(10)); /* prints "true" */
9 System.out.println(lessThan11.test(11)); /* prints "false" */
```

```
1 IntPredicate isEven = isOdd.negate(); /* it's true for 0, 2, 4, 6, 8, 10 and so on */
2 System.out.println(isEven.test(10)); /* prints "true" */
3 System.out.println(isEven.test(11)); /* prints "false" */
```

COMPOSING PREDICATES

```
1 IntPredicate isOddOrLessThan11 = isOdd.or(lessThan11);
2
3 System.out.println(isOddOrLessThan11.test(10)); /* prints "true" */
4 System.out.println(isOddOrLessThan11.test(11)); /* prints "true" */
5 System.out.println(isOddOrLessThan11.test(12)); /* prints "false" */
6 System.out.println(isOddOrLessThan11.test(13)); /* prints "true" */
7
8 IntPredicate isOddAndLessThan11 = isOdd.and(lessThan11);
9
10 System.out.println(isOddAndLessThan11.test(8)); /* prints "false" */
11 System.out.println(isOddAndLessThan11.test(9)); /* prints "true" */
12 System.out.println(isOddAndLessThan11.test(10)); /* prints "false" */
13 System.out.println(isOddAndLessThan11.test(11)); /* prints "false" */
```

COMPOSING CONSUMERS

```
1 Consumer<String> consumer = System.out::println;  
2 Consumer<String> doubleConsumer = consumer.andThen(System.out::println);  
3 doubleConsumer.accept("Hi!");
```

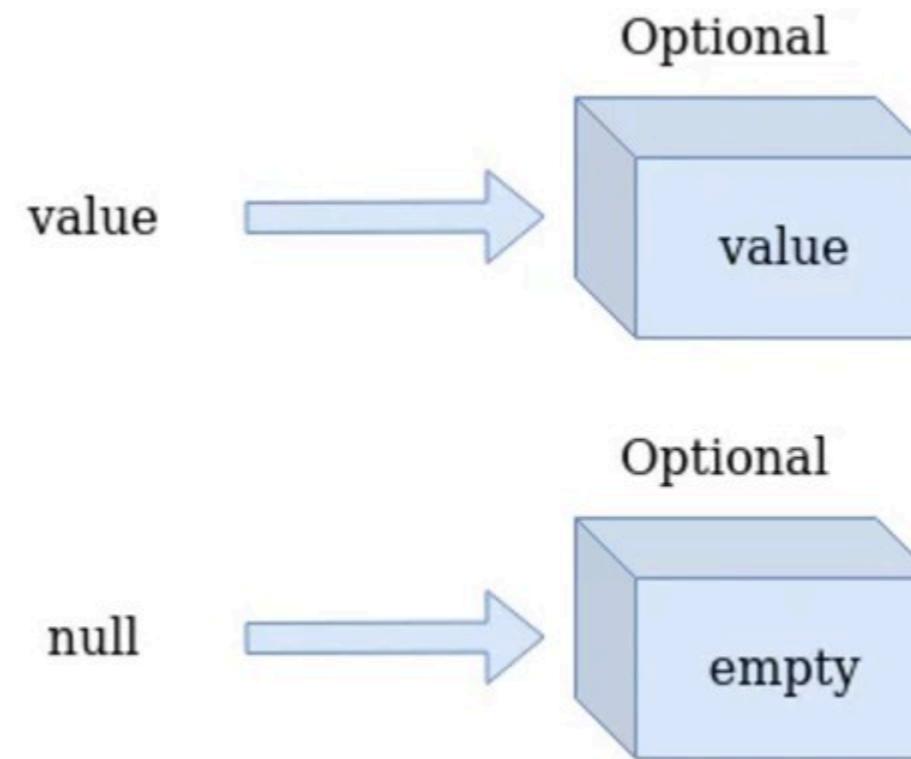
```
1 Hi!  
2 Hi!
```

OPTIONAL

```
1 Optional<String> absent = Optional.empty();  
2 Optional<String> present = Optional.of("Hello");
```

```
1 System.out.println(absent.isPresent()); /* false */  
2 System.out.println(present.isPresent()); /* true */
```

OPTIONALS AND NULLABLE OBJECTS



```
1 String message = getRandomMessage(); /* it may be null */
2
3 Optional<String> optMessage = Optional.ofNullable(message);
4
5 System.out.println(optMessage.isPresent()); /* true or false */
```

GETTING THE VALUE FROM AN OPTIONAL

get - returns the value of the instance if it's present, otherwise throws an exception

```
1 Optional<String> optName = Optional.of("John");  
2 String name = optName.get(); // "John"
```

```
1 Optional<String> optName = Optional.ofNullable(null);  
2 String name = optName.get(); // throws NoSuchElementException
```

GETTING THE VALUE FROM AN OPTIONAL

- `orElse` - used to extract the value wrapped inside an Optional object or return some default value when the Optional is empty

```
1 String nullableName = null;
2 String name = Optional.ofNullable(nullableName).orElse("unknown");
3
4 System.out.println(name); // unknown
```

- `orElseGet` - takes a supplier function to produce a result instead of taking some value to return

```
1 String name = Optional
2     .ofNullable(nullableName)
3     .orElseGet(SomeClass::getDefaultResult);
```

CONDITIONAL ACTIONS

- `ifPresent` -performs the given action with the value, otherwise does nothing
- `ifPresentOrElse` -performs the given action with the value, otherwise performs the given empty-based action

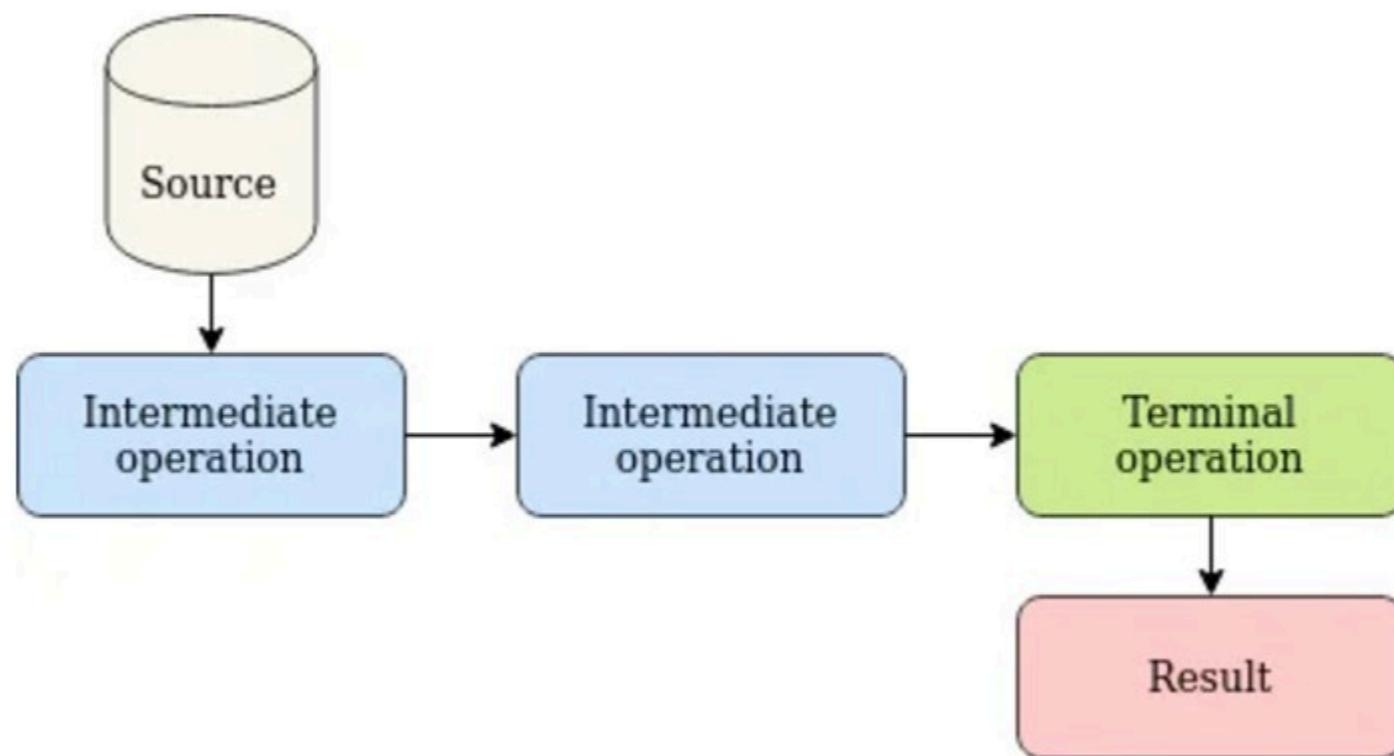
```
1 Optional<String> companyName = Optional.of("Google");
2 companyName.ifPresent((name) -> System.out.println(name.length())); // 6
```

```
1 Optional<String> noName = Optional.empty();
2 noName.ifPresent((name) -> System.out.println(name.length()));
```

```
1 Optional<String> optName = Optional.ofNullable(/* some value goes here */);
2
3 optName.ifPresentOrElse(
4     (name) -> System.out.println(name.length()),
5     () -> System.out.println(0)
6 );
```

FUNCTIONAL STREAMS

FUNCTIONAL DATA PROCESSING



A LOOP VS. A STREAM EXAMPLE

```
1 List<Integer> numbers = List.of(1, 4, 7, 6, 2, 9, 7, 8);
```

```
1 long count = 0;
2 for (int number : numbers) {
3     if (number > 5) {
4         count++;
5     }
6 }
7 System.out.println(count); /* 5 */
```

```
1 long count = numbers.stream()
2         .filter(number -> number > 5)
3         .count(); /* 5 */
```

```
1 long count = numbers.stream()
2         .skip(4) /* skip 1, 4, 7, 6 */
3         .filter(number -> number > 5)
4         .count(); /* 3 */
```

CREATING STREAMS

- from a collection:

```
1 List<Integer> famousNumbers = List.of(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55);
2 Stream<Integer> numbersStream = famousNumbers.stream();
3
4 Set<String> usefulConcepts = Set.of("functions", "lazy", "immutability");
5 Stream<String> conceptsStream = usefulConcepts.stream();
```

- from an array:

```
1 Stream<Double> doubleStream = Arrays.stream(new Double[]{ 1.01, 1d, 0.99, 1.02, 1d, 0.99 })
```

- directly from some values:

```
1 Stream<String> persons = Stream.of("John", "Demetra", "Cleopatra");
```

CREATING STREAMS

- concatenate streams together:

```
1 Stream<String> stream1 = Stream.of(/* some values */);
2 Stream<String> stream2 = Stream.of(/* some values */);
3 Stream<String> resultStream = Stream.concat(stream1, stream2);
```

- empty streams:

```
1 Stream<Integer> empty1 = Stream.of();
2 Stream<Integer> empty2 = Stream.empty();
```

INTERMEDIATE OPERATIONS

- `filter` returns a new stream that includes the elements that match a **predicate**;
- `limit` returns a new stream that consists of the first `n` elements of this stream;
- `skip` returns a new stream without the first `n` elements of this stream;
- `distinct` returns a new stream consisting of only unique elements according to the results of `equals`;
- `sorted` returns a new stream that includes elements sorted according to the natural order or a given **comparator**;
- `peek` returns the same stream of elements but allows observing the current elements of the stream for debugging;
- `map` returns a new stream that consists of the elements that were obtained by applying a function (i.e. transforming each element).

TERMINAL OPERATIONS

- `count` returns the number of elements in the stream as a `long` value;
- `max` / `min` returns an `Optional` maximum/minimum element of the stream according to the given comparator;
- `reduce` combines values from the stream into a single value (an aggregate value);
- `findFirst` / `findAny` returns the first / any element of the stream as an `Optional` ;
- `anyMatch` returns `true` if at least one element matches a predicate (see also:
`allMatch` , `noneMatch`);
- `forEach` takes a **consumer** and applies it to each element of the stream (for example, printing it);
- `collect` returns a collection of the values in the stream;
- `toArray` returns an array of the values in a stream.

AN EXAMPLE

```
1 List<String> companies = List.of(  
2     "Google", "Amazon", "Samsung",  
3     "GOOGLE", "amazon", "Oracle"  
4 );  
5  
6 companies.stream()  
7     .map(String::toUpperCase) /* transform each name to the upper case*/  
8     .distinct() /* intermediate operation: keep only unique words*/  
9     .forEach(System.out::println); /* print every company*/
```

```
1 GOOGLE  
2 AMAZON  
3 SAMSUNG  
4 ORACLE
```

THE MAP OPERATION

```
1 List<Double> numbers = List.of(6.28, 5.42, 84.0, 26.0);
```

```
1 List<Double> famousNumbers = numbers.stream()  
2     .map(number -> number / 2) /* divide each number in the stream by 2 */  
3     .collect(Collectors.toList()); /* collect transformed numbers in a new list */
```

```
1 [3.14, 2.71, 42.0, 13.0]
```

THE MAP OPERATION

```
1 public class Job {  
2     private String title;  
3     private String description;  
4     private double salary;  
5  
6     /* getters and setters */  
7 }
```

```
1 List<String> titles = jobs.stream()  
2         .map(Job::getTitle) /* get title of each job */  
3         .collect(Collectors.toList()); /* collect titles in a new list */
```

THE MAP OPERATION

```
1 class User {  
2     private long id;  
3     private String firstName;  
4     private String lastName;  
5 }  
6  
7 class Account {  
8     private long id;  
9     private boolean isLocked;  
10    private User owner;  
11 }  
12  
13 class AccountInfo {  
14     private long id;  
15     private String ownerFullName;  
16 }
```

```
1 List<AccountInfo> infoList = accounts.stream()  
2     .map(acc -> {  
3         AccountInfo info = new AccountInfo();  
4         info.setId(acc.getId());  
5         String ownerFirstName = acc.getOwner().getFirstName();  
6         String ownerLastName = acc.getOwner().getLastName();  
7         info.setOwnerFullName(ownerFirstName + " " + ownerLastName);  
8         return info;  
9     }).collect(Collectors.toList());
```

PRIMITIVE-SPECIALIZED TYPES OF THE MAP OPERATION

```
1 class Planet {  
2     private String name;  
3     private int orderFromSun;  
4  
5     public Planet(int orderFromSun) {  
6         this.orderFromSun = orderFromSun;  
7     }  
8 }
```

```
1 List<Planet> planets = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8)  
2         .mapToObj(Planet::new)  
3         .collect(Collectors.toList());
```

THE FLATMAP OPERATION

```
1 List<Book> javaBooks = List.of(  
2     new Book("Java EE 7 Essentials", 2013, List.of("Arun Gupta")),  
3     new Book("Algorithms", 2011, List.of("Robert Sedgewick", "Kevin Wayne")),  
4     new Book("Clean code", 2014, List.of("Robert Martin"))  
5 );
```

```
1 List<String> authors = javaBooks.stream()  
2     .flatMap(book -> book.getAuthors().stream())  
3     .collect(Collectors.toList());
```

```
1 ["Arun Gupta", "Robert Sedgewick", "Kevin Wayne", "Robert Martin"]
```

THE REDUCE OPERATION

- reduce is a method of a Stream class that combines elements of a stream into a single value. The result can be a value of a primitive type or a complex object.

```
1 List<Integer> transactions = List.of(20, 40, -60, 5);  
2 transactions.stream().reduce((sum, transaction) -> sum + transaction);
```

iteration	sum	transaction
0	20	40
1	$20 + 40 =$ 60	-60
2	$60 + -60 =$ 0	5
return	$0 + 5 =$ 5	

OTHER REDUCTION OPERATIONS

```
1 transactions.stream().reduce((t1, t2) -> t2 > t1 ? t2 : t1)
```

```
1 transactions.stream().max(Integer::compareTo);
```

```
1 IntStream.of(20, 40, -60, 5).max();
```

STREAM FILTERING

```
1 List<Integer> primeNumbers = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);  
  
1 List<Integer> filteredPrimeNumbers = primeNumbers.stream() /* create a stream from the l  
2     .filter(n -> n >= 11 && n <= 23) /* filter elements */  
3     .collect(Collectors.toList()); /* collect elements in a new list */  
  
1 [11, 13, 17, 19, 23]  
  
1 Predicate<Integer> between11and23 = n -> n >= 11 && n <= 23; /* instantiate the predicit  
2  
3 List<Integer> filteredPrimeNumbers = primeNumbers.stream() /* create a stream from the l  
4     .filter(between11and23) /* pass the predicate to the filter method */  
5     .collect(Collectors.toList()); /* collect elements in a new list */
```

USING MULTIPLE FILTERS

```
1 List<String> programmingLanguages = Arrays.asList("Java", "", "scala", "Kotlin", "", "cl
```

```
1 long count = programmingLanguages.stream()
  2     .filter(lang -> lang.length() > 0) /* consider only non-empty strings */
  3     .filter(lang -> Character.isUpperCase(lang.charAt(0)))
  4     .count(); /* count suitable languages */
```

```
1 filter(lang -> lang.length() > 0 && Character.isUpperCase(lang.charAt(0)))
```

COLLECTORS

```
1 public class Account {  
2     private long balance;  
3     private String number;  
4  
5     /* getters and setters */  
6 }
```

```
1 List<Account> accounts = accountStream.collect(Collectors.toList());
```

```
1 Set<Account> accounts = accountStream.collect(Collectors.toSet());
```

```
1 LinkedList<Account> accounts = accountStream.collect(Collectors.toCollection(LinkedList::
```

PRODUCING VALUES

- `summingInt`, `summingLong`, `summingDouble`
- `averagingInt`, `averagingLong`, `averagingDouble`
- `maxBy`, `minBy`
- `counting`

```
1 long summary = accounts.stream()  
2     .collect(summingLong(Account::getBalance));
```

```
1 double average = accounts.stream()  
2     .collect(averagingLong(Account::getBalance));
```

```
1 String megaNumber = accountStream.collect(Collectors.reducing("",  
2     account -> account.getNumber(),  
3     (numbers, number) -> numbers.concat(number)  
4 ));
```

STREAM PIPELINES

```
1 List<String> words = List.of("JAR", "Java", "Kotlin", "JDK", "jakarta");
2
3 long numberOfWords = words.stream()
4     .map(String::toUpperCase)          /* convert all words to upper case */
5     .filter(s -> s.startsWith("JA"))  /* filter words using a prefix */
6     .count();                         /* count the suitable words */
7
8 System.out.println(numberOfWords); /* 3 */
```

height:200

THE ORDER OF EXECUTION

```
1 long numberOfWords = words.stream()
2     .map(String::toUpperCase)
3     .peek(System.out::println)
4     .filter(s -> s.startsWith("JA"))
5     .peek(System.out::println)
6     .count();
```

Output:

```
1 JAR
2 JAR
3 JAVA
4 JAVA
5 KOTLIN
6 JDK
7 JAKARTA
8 JAKARTA
```

STREAMS WITH CUSTOM CLASSES

Let's assume that we have the `Event` class that represents a public event, such as a conference, a film premiere, or a concert. It has two fields:

- `beginning (LocalDate)` is a date when the event happens;
- `name (String)` that is the name of the event (for instance, "JavaOne – 2017").

Also, the class has getters and setters for each field with the corresponding names.

We also have a list of instances named `events`

```
1 LocalDate after = LocalDate.of(2017, 12, 29);
2 LocalDate before = LocalDate.of(2018, 1, 1);
3
4 List<String> suitableEvents = events.stream()
5     .filter(e -> e.getBeginning().isAfter(after) && e.getBeginning().isBefore(before))
6     .map(Event::getName)
7     .collect(Collectors.toList());
```

MAPPING AND REDUCING FUNCTIONS

```
1 public static IntPredicate negateEachAndConjunctAll(Collection<IntPredicate> predicates)
2     return predicates.stream()
3         .map(IntPredicate::negate)
4         .reduce(n -> true, IntPredicate::and);
5 }
```

⇒ the input predicates $P_1(x), P_2(x), \dots, P_n(x)$ will be reduced into one predicate $Q(x) = \text{not } P_1(x) \text{ and not } P_2(x) \text{ and ... and not } P_n(x)$

TAKING ELEMENTS

THE TAKEWHILE METHOD

```
1 List<Integer> numbers =  
2     Stream.of(3, 5, 1, 2, 0, 4, 5)  
3         .takeWhile(n -> n > 0)  
4         .collect(Collectors.toList());  
5  
6 System.out.println(numbers); // [3, 5, 1, 2]
```

THE DROPWHILE METHOD

```
1 List<Integer> numbers =  
2     Stream.of(3, 5, 1, 2, 0, 4, 5)  
3         .dropWhile(n -> n > 0)  
4         .collect(Collectors.toList());  
5  
6 System.out.println(numbers); // [0, 4, 5]
```

THE CASE OF UNORDERED STREAMS

```
1 Set<String> conferences = Set.of(  
2     "JokerConf", "JavaZone",  
3     "KotlinConf", "JFokus"  
4 );  
5  
6 conferences.stream()  
7     .takeWhile(word -> word.startsWith("J"))  
8     .forEach(System.out::println);
```

GROUPING COLLECTORS

PARTITIONING

```
1 List<Account> accounts = List.of(  
2     new Account(3333, "530012"),  
3     new Account(15000, "771843"),  
4     new Account(0, "681891")  
5 );
```

```
1 Map<Boolean, List<Account>> accountsByBalance = accounts.stream()  
2     .collect(Collectors.partitioningBy(account -> account.getBalance() >= 10000));
```

```
1 {  
2     false=[Account{balance=3333, number='530012'}, Account{balance=0, number='681891'}],  
3     true=[Account{balance=15000, number='771843'}]  
4 }
```

GROUPING

```
1 enum Status {  
2     ACTIVE,  
3     BLOCKED,  
4     REMOVED  
5 }  
6 public class Account {  
7     private long balance;  
8     private String number;  
9     private Status status;  
10    /* constructors, getters and setters */  
11 }
```

```
1 List<Account> accounts = List.of(  
2         new Account(3333L, "530012", Status.REMOVED),  
3         new Account(15000L, "771843", Status.ACTIVE),  
4         new Account(0L, "681891", Status.BLOCKED)  
5 );
```

```
1 Map<Status, List<Account>> accountsByStatus = accounts.stream()  
2     .collect(Collectors.groupingBy(Account::getStatus));
```

```
1 {  
2     BLOCKED=[Account{balance=0, number='681891'}],  
3     REMOVED=[Account{balance=3333, number='530012'}],  
4     ACTIVE=[Account{balance=15000, number='771843'}]  
5 }
```

DOWNSTREAM COLLECTORS

```
1 List<Account> accounts = List.of(  
2     new Account(3333L, "530012", Status.ACTIVE),  
3     new Account(15000L, "771843", Status.BLOCKED),  
4     new Account(15000L, "234465", Status.ACTIVE),  
5     new Account(8800L, "110011", Status.ACTIVE),  
6     new Account(45000L, "462181", Status.BLOCKED),  
7     new Account(0L, "681891", Status.REMOVED)  
8 );
```

```
1 Map<Status, Long> sumByStatuses = accounts.stream()  
2     .collect(Collectors.groupingBy(  
3         Account::getStatus,  
4         Collectors.summingLong(Account::getBalance))  
5     );
```

```
1 { REMOVED=0, ACTIVE=27133, BLOCKED=60000 }
```

TEEING COLLECTOR

```
1 long[ ] countAndSum = accounts
2     .stream()
3     .filter(account -> account.getStatus() == Status.BLOCKED)
4     .collect(Collectors.teeing(
5         Collectors.counting(),
6         Collectors.summingLong(Account::getBalance),
7         (count, sum) -> new long[ ]{count, sum}
8     );
```

STRATEGY PATTERN EVOLUTION

- Traditional: Interface + Multiple Implementation Classes
- Modern: Interface + Lambda Expressions/Method References
 - Reduced boilerplate code
 - Increased code clarity

TRADITIONAL STRATEGY PATTERN

```
1 /* Strategy interface */
2 public interface ValidationStrategy {
3     boolean validate(String text);
4 }
5 /* Concrete strategy implementations */
6 public class NumericValidationStrategy implements ValidationStrategy {
7     public boolean validate(String text) {
8         return text.matches("[0-9]+"); /* Check if string contains only numbers */
9     }
10 }
11 public class AlphabeticValidationStrategy implements ValidationStrategy {
12     public boolean validate(String text) {
13         return text.matches("[a-zA-Z]+"); /* Check if string contains only letters */
14     }
15 }
```

TRADITIONAL STRATEGY USAGE

```
1 public class TextValidator {  
2     private ValidationStrategy strategy;  
3     public TextValidator(ValidationStrategy strategy) {  
4         this.strategy = strategy; /* Set validation strategy */  
5     }  
6     public boolean validateText(String text) {  
7         return strategy.validate(text); /* Delegate to strategy implementation */  
8     }  
9 }  
10  
11 /* Client code */  
12 TextValidator numericValidator = new TextValidator(new NumericValidationStrategy());  
13 TextValidator alphabeticValidator = new TextValidator(new AlphabeticValidationStrategy())
```

LAMBDA-BASED STRATEGY PATTERN

```
1 /* Function interface replacing strategy interface */
2 public class TextValidator {
3     private Function<String, Boolean> validationStrategy;
4     public TextValidator(Function<String, Boolean> strategy) {
5         this.validationStrategy = strategy; /* Set validation strategy using lambda */
6     }
7     public boolean validateText(String text) {
8         return validationStrategy.apply(text);/* Delegate to lambda implementation */
9     }
10 }
```

LAMBDA STRATEGY USAGE

```
1  /* Define strategies as lambdas */
2  TextValidator numericValidator = new TextValidator(text -> text.matches("[0-9]+"));
3  TextValidator alphabeticValidator = new TextValidator(text -> text.matches("[a-zA-Z]+"))
4  /* Or using method references */
5  public class ValidationStrategies {
6      public static boolean isNumeric(String text) {
7          return text.matches("[0-9]+");
8      }
9      public static boolean isAlphabetic(String text) {
10         return text.matches("[a-zA-Z]+");
11     }
12 }
13
14 TextValidator numericValidator = new TextValidator(ValidationStrategies::isNumeric);
```

STRATEGY LIBRARY APPROACH

- Create utility class with static methods
- Use method references as strategies
- Build reusable strategy libraries

```
1 public class ValidationStrategies {  
2     /* Library of validation strategies */  
3     public static final Function<String, Boolean> NUMERIC = text -> text.matches("[0-9]+");  
4     public static final Function<String, Boolean> ALPHABETIC = text -> text.matches("[a-zA-Z]+");  
5     public static final Function<String, Boolean> ALPHANUMERIC = text -> text.matches("[a-zA-Z0-9]+");  
6 }
```

DECORATOR PATTERN EVOLUTION

- Traditional: Component + Multiple Decorator Classes
- Modern: Function Interface + Lambda Composition
- Simplified decoration through composition
- Enhanced flexibility and readability

TRADITIONAL DECORATOR PATTERN

```
1  /* Component interface */
2  public interface TextProcessor {
3      String process(String text);
4  }
5
6  /* Concrete component */
7  public class BasicTextProcessor implements TextProcessor {
8      public String process(String text) {
9          /* Basic text processing */
10         return text.trim();
11     }
12 }
13
14 /* Abstract decorator */
15 public abstract class TextDecorator implements TextProcessor {
16     protected TextProcessor processor;
17
18     public TextDecorator(TextProcessor processor) {
19         /* Store decorated component */
20         this.processor = processor;
21     }
22 }
```

TRADITIONAL DECORATORS

```
1 /* Concrete decorators */
2 public class UpperCaseDecorator extends TextDecorator {
3     public UpperCaseDecorator(TextProcessor processor) {
4         super(processor);
5     }
6
7     public String process(String text) {
8         /* Apply uppercase decoration */
9         return processor.process(text).toUpperCase();
10    }
11 }
12
13 /* Usage example */
14 TextProcessor processor = new UpperCaseDecorator(
15     new TrimDecorator(
16         new BasicTextProcessor()
17     )
18 );
```

LAMBDA-BASED DECORATOR PATTERN

```
1 public class TextProcessor {  
2     private Function<String, String> processor;  
3  
4     public TextProcessor(Function<String, String> processor) {  
5         /* Store processing function */  
6         this.processor = processor;  
7     }  
8  
9     public String process(String text) {  
10        /* Apply processing function */  
11        return processor.apply(text);  
12    }  
13  
14    public TextProcessor decorate(Function<String, String> decoration) {  
15        /* Add new decoration using composition */  
16        return new TextProcessor(processor.andThen(decoration));  
17    }  
18 }
```

FUNCTION COMPOSITION

```
1 /* Define processing functions */
2 Function<String, String> trim = String::trim;
3 Function<String, String> toUpperCase = String::toUpperCase;
4 Function<String, String> removeSpaces = s -> s.replace(" ", "");
5
6 /* Create and decorate processor */
7 TextProcessor processor = new TextProcessor(trim)
8     .decorate(toUpperCase)
9     .decorate(removeSpaces);
10
11 /* Direct function composition */
12 Function<String, String> directProcessor = trim
13     .andThen(toUpperCase)
14     .andThen(removeSpaces);
```

DECORATOR LIBRARY APPROACH

```
1 public class TextDecorators {
2     /* Library of text decorations */
3     public static final Function<String, String> TRIM = String::trim;
4     public static final Function<String, String> UPPERCASE = String::toUpperCase;
5     public static final Function<String, String> NO_SPACES =
6         s -> s.replace(" ", "");
7
8     /* Factory method for parameterized decorators */
9     public static Function<String, String> prefix(String prefix) {
10         return text -> prefix + text;
11     }
12 }
```

COMPARING APPROACHES

- Traditional Pattern:
 - Class hierarchy
 - Inheritance-based
 - Fixed structure
- Lambda-based Pattern:
 - Function composition
 - Interface-based
 - Flexible structure

TEMPLATE METHOD PATTERN EVOLUTION

- Traditional: Abstract Class + Concrete Subclasses
- Modern: Base Class + Lambda Injection
- Shift from inheritance to composition
- More flexible behavior customization

TRADITIONAL TEMPLATE METHOD

```
1  /* Abstract base class with template method */
2  public abstract class DataMiner {
3      /* Template method defining the algorithm */
4      public final void mine() {
5          String data = extractData();
6          String processed = parseData(data);
7          String analyzed = analyzeData(processed);
8          sendReport(analyzed);
9      }
10
11     /* Abstract methods to be implemented by subclasses */
12     protected abstract String extractData();
13     protected abstract String parseData(String data);
14
15     /* Common method for all implementations */
16     protected String analyzeData(String data) {
17         return "Analyzed: " + data;
18     }
19
20     protected void sendReport(String report) {
21         System.out.println(report);
22     }
23 }
```

TRADITIONAL IMPLEMENTATION

```
1  /* Concrete implementation for CSV files */
2  public class CSVDataMiner extends DataMiner {
3      protected String extractData() {
4          /* CSV specific extraction */
5          return "data from CSV";
6      }
7
8      protected String parseData(String data) {
9          /* CSV specific parsing */
10         return "parsed " + data;
11     }
12 }
13
14 /* Concrete implementation for PDF files */
15 public class PDFDataMiner extends DataMiner {
16     protected String extractData() {
17         /* PDF specific extraction */
18         return "data from PDF";
19     }
20
21     protected String parseData(String data) {
22         /* PDF specific parsing */
23         return "parsed " + data;
24     }
25 }
```

LAMBDA-BASED TEMPLATE METHOD

```
1 /* Single class with injected behaviors */
2 public class DataMiner {
3     private final Function<Void, String> extractData;
4     private final Function<String, String> parseData;
5
6     public DataMiner(
7         Function<Void, String> extractData,
8         Function<String, String> parseData) {
9         this.extractData = extractData; /* Store injected behaviors */
10        this.parseData = parseData;
11    }
12
13    /* Template method using injected behaviors */
14    public final void mine() {
15        String data = extractData.apply(null);
16        String processed = parseData.apply(data);
17        String analyzed = analyzeData(processed);
18        sendReport(analyzed);
19    }
20    /* Common methods remain in base class */
21    protected String analyzeData(String data) {
22        return "Analyzed: " + data;
23    }
24
25    protected void sendReport(String report) {
26        System.out.println(report);
27    }
```

LAMBDA-BASED USAGE

```
1 /* Create miners with different behaviors */
2 DataMiner csvMiner = new DataMiner(
3     unused -> "data from CSV",
4     data -> "parsed " + data
5 );
6
7 DataMiner pdfMiner = new DataMiner(
8     unused -> "data from PDF",
9     data -> "parsed " + data
10 );
11
12 /* Create miner with complex behavior */
13 DataMiner complexMiner = new DataMiner(
14     unused -> "complex data", /* Complex extraction logic */
15     data -> "complex " + data /* Complex parsing logic */
16 );
```

BEHAVIOR LIBRARY APPROACH

```
1 public class DataMinerBehaviors {
2     /* Library of extraction behaviors */
3     public static Function<Void, String> csvExtractor =
4         unused -> "data from CSV";
5     public static Function<Void, String> pdfExtractor =
6         unused -> "data from PDF";
7
8     /* Library of parsing behaviors */
9     public static Function<String, String> simpleParser =
10        data -> "parsed " + data;
11     public static Function<String, String> complexParser =
12        data -> "complex parsed " + data;
13
14     /* Factory method for parameterized behavior */
15     public static Function<Void, String> fileExtractor(String path) {
16         return unused -> "data from " + path;
17     }
18 }
19
20 /* Usage with behavior library */
21 DataMiner miner = new DataMiner(
22     DataMinerBehaviors.csvExtractor,
23     DataMinerBehaviors.complexParser
24 );
```

COMBINING BEHAVIORS

```
1 /* Creating composite behaviors */
2 Function<Void, String> loggingExtractor =
3     unused -> {
4         System.out.println("Extracting data...");
5         return DataMinerBehaviors.csvExtractor.apply(null);
6     };
7
8 Function<String, String> validatingParser =
9     data -> {
10     if (data == null) throw new IllegalArgumentException();
11     return DataMinerBehaviors.simpleParser.apply(data);
12 };
13
14 /* Usage with composite behaviors */
15 DataMiner miner = new DataMiner(
16     loggingExtractor,
17     validatingParser
18 );
```

JAVA CONCURRENCY API

THREADS AND RUNNABLES

- All modern operating systems support concurrency both via processes and threads.
- Processes are instances of programs which typically run independent to each other
- Inside those processes we can utilize threads to execute code concurrently
- Java supports threads since JDK 1.0

BASIC THREAD CREATION

```
1 Runnable task = () -> {  
2     String threadName = Thread.currentThread().getName();  
3     System.out.println("Hello " + threadName);  
4 };  
5  
6 task.run();  
7  
8 Thread thread = new Thread(task);  
9 thread.start();  
10  
11 System.out.println("Done!");
```

Output might look like:

```
1 Hello main
2 Hello Thread-0
3 Done!
```

or:

```
1 Hello main
2 Done!
3 Hello Thread-0
```

THREAD SLEEP

```
1 Runnable runnable = () -> {
2     try {
3         String name = Thread.currentThread().getName();
4         System.out.println("Foo " + name);
5         TimeUnit.SECONDS.sleep(1);
6         System.out.println("Bar " + name);
7     }
8     catch (InterruptedException e) {
9         e.printStackTrace();
10    }
11 };
12 Thread thread = new Thread(runnable);
13 thread.start();
```

EXECUTORS

- ExecutorService as a higher level replacement for working with threads directly
- capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually
- All threads of the internal pool will be reused under the hood for tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

```
1 ExecutorService executor = Executors.newSingleThreadExecutor();
2 executor.submit(() -> {
3     String threadName = Thread.currentThread().getName();
4     System.out.println("Hello " + threadName);
5 });
6
7 // => Hello pool-1-thread-1
```

EXECUTORS

- Executors have to be stopped explicitly - otherwise they keep listening for new tasks:

```
1 try {
2     System.out.println("attempt to shutdown executor");
3     executor.shutdown();
4     executor.awaitTermination(5, TimeUnit.SECONDS);
5 }
6 catch (InterruptedException e) {
7     System.err.println("tasks interrupted");
8 }
9 finally {
10    if (!executor.isTerminated()) {
11        System.err.println("cancel non-finished tasks");
12    }
13    executor.shutdownNow();
14    System.out.println("shutdown finished");
15 }
```

CALLABLES AND FUTURES

- in addition to Runnable executors support another kind of task named Callable
- Callables are functional interfaces just like Runnables but instead of being void they return a value.

CALLABLE

```
1 Callable<Integer> task = () -> {
2     try {
3         TimeUnit.SECONDS.sleep(1);
4         return 123;
5     }
6     catch (InterruptedException e) {
7         throw new IllegalStateException("task interrupted", e);
8     }
9 };
```

FUTURE

```
1 ExecutorService executor = Executors.newFixedThreadPool(1);
2 Future<Integer> future = executor.submit(task);
3
4 System.out.println("future done? " + future.isDone());
5 Integer result = future.get();
6
7 System.out.println("future done? " + future.isDone());
8 System.out.print("result: " + result);
```

TIMEOUTS

- Any call to `future.get()` will block and wait until the underlying callable has been terminated
- In the worst case a callable runs forever - thus making your application unresponsive. You can simply counteract this by passing a timeout:

```
1 ExecutorService executor = Executors.newFixedThreadPool(1);
2
3 Future<Integer> future = executor.submit(() -> {
4     try {
5         TimeUnit.SECONDS.sleep(2);
6         return 123;
7     }
8     catch (InterruptedException e) {
9         throw new IllegalStateException("task interrupted", e);
10    }
11 });
12
13 future.get(1, TimeUnit.SECONDS);
```

INVOKEALL

```
1 ExecutorService executor = Executors.newWorkStealingPool();
2 List<Callable<String>> callables = Arrays.asList(
3     () -> "task1",
4     () -> "task2",
5     () -> "task3");
6 executor.invokeAll(callables)
7     .stream()
8     .map(future -> {
9         try {
10             return future.get();
11         }
12         catch (Exception e) {
13             throw new IllegalStateException(e);
14         }
15     })
16     .forEach(System.out::println);
```

INVOKEANY

```
1 Callable<String> callable(String result, long sleepSeconds) {  
2     return () -> {  
3         TimeUnit.SECONDS.sleep(sleepSeconds);  
4         return result;  
5     };  
6 }
```

```
1 ExecutorService executor = Executors.newWorkStealingPool();
2
3 List<Callable<String>> callables = Arrays.asList(
4     callable("task1", 2),
5     callable("task2", 1),
6     callable("task3", 3));
7
8 String result = executor.invokeAny(callables);
9 System.out.println(result);
10
11 // => task2
```

SCHEDULED EXECUTORS

- ScheduledExecutorService allows periodic task execution
- Tasks can run once after delay or periodically
- Two types of scheduling available
 - Fixed rate execution
 - Fixed delay execution

BASIC SCHEDULING

```
1 ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
2
3 Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
4 ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);
5
6 TimeUnit.MILLISECONDS.sleep(1337);
7
8 long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
9 System.out.printf("Remaining Delay: %sms", remainingDelay);
```

FIXED RATE EXECUTION

- Executes tasks at fixed time intervals
- Uses `scheduleAtFixedRate()`
- Doesn't account for task duration

```
1 ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
2
3 Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
4
5 int initialDelay = 0;
6 int period = 1;
7 executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

FIXED DELAY EXECUTION

- Delay starts after task completion
- Better for unpredictable task duration
- Uses scheduleWithFixedDelay()

```
1 ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
2
3 Runnable task = () -> {
4     try {
5         TimeUnit.SECONDS.sleep(2);
6         System.out.println("Scheduling: " + System.nanoTime());
7     }
8     catch (InterruptedException e) {
9         System.err.println("task interrupted");
10    }
11 };
12
13 executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

SYNCHRONIZATION AND LOCKS

SYNCHRONIZED

- Thread synchronization is crucial for shared mutable state
- Race conditions occur without proper synchronization
- Java provides the `synchronized` keyword
- Also supports explicit locks via Lock interface

RACE CONDITION EXAMPLE

```
1 int count = 0;
2
3 void increment() {
4     count = count + 1;
5 }
6
7 ExecutorService executor = Executors.newFixedThreadPool(2);
8 IntStream.range(0, 10000)
9     .forEach(i -> executor.submit(this::increment));
10
11 stop(executor);
12 System.out.println(count); // 9965
```

SYNCHRONIZED SOLUTION

```
1 synchronized void incrementSync() {  
2     count = count + 1;  
3 }  
4  
5 /* Or as a block statement */  
6 void incrementSyncBlock() {  
7     synchronized (this) {  
8         count = count + 1;  
9     }  
10 }
```

REENTRANTLOCK

- Mutual exclusion lock with extended capabilities
- More expressive than implicit monitors
- Supports reentrant characteristics

```
1 ReentrantLock lock = new ReentrantLock();
2 int count = 0;
3
4 void increment() {
5     lock.lock();
6     try {
7         count++;
8     } finally {
9         lock.unlock();
10    }
11 }
```

REENTRANTLOCK FEATURES

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 ReentrantLock lock = new ReentrantLock();
3 executor.submit(() -> {
4     lock.lock();
5     try { sleep(1); } finally { lock.unlock(); }
6 });
7
8 executor.submit(() -> {
9     System.out.println("Locked: " + lock.isLocked());
10    System.out.println("Held by me: " + lock.isHeldByCurrentThread());
11    boolean locked = lock.tryLock();
12    System.out.println("Lock acquired: " + locked);
13 });
14
15 stop(executor);
```

READWRITELOCK

- Maintains separate locks for read and write access
- Multiple threads can hold read lock simultaneously
- Only one thread can hold write lock
- Improves performance for read-heavy scenarios

READWRITELOCK EXAMPLE

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 Map<String, String> map = new HashMap<>();
3 ReadWriteLock lock = new ReentrantReadWriteLock();
4
5 executor.submit(() -> {
6     lock.writeLock().lock();
7     try {
8         sleep(1);
9         map.put("foo", "bar");
10    } finally {
11        lock.writeLock().unlock();
12    }
13 } );
```

READ TASKS

```
1 Runnable readTask = () -> {
2     lock.readLock().lock();
3     try {
4         System.out.println(map.get("foo"));
5         sleep(1);
6     } finally {
7         lock.readLock().unlock();
8     }
9 };
10 executor.submit(readTask);
11 executor.submit(readTask);
12 stop(executor);
```

STAMPEDLOCK

- New in Java 8
- Supports read/write locks with stamps
- Adds optimistic reading mode
- Not reentrant - be careful of deadlocks

STAMPEDLOCK EXAMPLE

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 Map<String, String> map = new HashMap<>();
3 StampedLock lock = new StampedLock();
4
5 executor.submit(() -> {
6     long stamp = lock.writeLock();
7     try {
8         sleep(1);
9         map.put("foo", "bar");
10    } finally {
11        lock.unlockWrite(stamp);
12    }
13 }) ;
```

```
1 Runnable readTask = () -> {
2     long stamp = lock.readLock();
3     try {
4         System.out.println(map.get("foo"));
5         sleep(1);
6     } finally {
7         lock.unlockRead(stamp);
8     }
9 };
10
11 executor.submit(readTask);
12 executor.submit(readTask);
13 stop(executor);
```

OPTIMISTIC LOCKING

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2 StampedLock lock = new StampedLock();
3
4 executor.submit(() -> {
5     long stamp = lock.tryOptimisticRead();
6     try {
7         System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
8         sleep(1);
9         System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
10        sleep(2);
11        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
12    } finally {
13        lock.unlock(stamp);
14    }
15 }) ;
```

```
1 executor.submit(() -> {
2     long stamp = lock.writeLock();
3     try {
4         System.out.println("Write Lock acquired");
5         sleep(2);
6     } finally {
7         lock.unlock(stamp);
8         System.out.println("Write done");
9     }
10 });
11 stop(executor);
```

```
1 Optimistic Lock Valid: true
2 Write Lock acquired
3 Optimistic Lock Valid: false
4 Write done
5 Optimistic Lock Valid: false
```

SEMAPHORES

- Maintains a set of permits
- Limits concurrent access to resources
- Useful for throttling operations

SEMAPHORE EXAMPLE

```
1 ExecutorService executor = Executors.newFixedThreadPool(10);
2 Semaphore semaphore = new Semaphore(5);
3
4 Runnable longRunningTask = () -> {
5     boolean permit = false;
6     try {
7         permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
8         if (permit) {
9             System.out.println("Semaphore acquired");
10            sleep(5);
11        } else {
12            System.out.println("Could not acquire semaphore");
13        }
14    } catch (InterruptedException e) {
15        throw new IllegalStateException(e);
16    } finally {
17        if (permit) {
18            semaphore.release();
19        }
20    }
21 }
```

```
1 Semaphore acquired
2 Semaphore acquired
3 Semaphore acquired
4 Semaphore acquired
5 Semaphore acquired
6 Could not acquire semaphore
7 Could not acquire semaphore
8 Could not acquire semaphore
9 Could not acquire semaphore
10 Could not acquire semaphore
```

ATOMIC VARIABLES AND CONCURRENTMAP

ATOMICINTEGER

- Package `java.concurrent.atomic` for thread-safe operations
- Uses CPU-level Compare-and-Swap (CAS)
- Faster than locks for single variable changes
- Multiple atomic classes available: `AtomicInteger`, `AtomicBoolean`, `AtomicLong`, `AtomicReference`

BASIC ATOMICINTEGER USAGE

```
1 AtomicInteger atomicInt = new AtomicInteger(0);
2
3 ExecutorService executor = Executors.newFixedThreadPool(2);
4
5 IntStream.range(0, 1000)
6     .forEach(i -> executor.submit(atomicInt::incrementAndGet));
7
8 stop(executor);
9
10 /* Prints 1000 */
11 System.out.println(atomicInt.get());
```

UPDATEANDGET OPERATION

```
1 AtomicInteger atomicInt = new AtomicInteger(0);
2
3 ExecutorService executor = Executors.newFixedThreadPool(2);
4
5 IntStream.range(0, 1000)
6     .forEach(i -> {
7         Runnable task = () ->
8             atomicInt.updateAndGet(n -> n + 2);
9         executor.submit(task);
10    });
11
12 stop(executor);
13
14 /* Prints 2000 */
15 System.out.println(atomicInt.get());
```

ACCUMULATEANDGET OPERATION

```
1 AtomicInteger atomicInt = new AtomicInteger(0);
2
3 ExecutorService executor = Executors.newFixedThreadPool(2);
4
5 IntStream.range(0, 1000)
6     .forEach(i -> {
7         Runnable task = () ->
8             atomicInt.accumulateAndGet(i, (n, m) -> n + m);
9         executor.submit(task);
10    });
11
12 stop(executor);
13
14 /* Prints 499500 */
15 System.out.println(atomicInt.get());
```

LONGADDER

- Alternative to AtomicLong
- Optimized for concurrent additions
- Maintains internal set of variables
- Better performance for write-heavy scenarios

LONGADDER EXAMPLE

```
1 ExecutorService executor = Executors.newFixedThreadPool(2);
2
3 IntStream.range(0, 1000)
4     .forEach(i -> executor.submit(adder::increment));
5
6 stop(executor);
7
8 /* Prints 1000 */
9 System.out.println(adder.sumThenReset());
```

LONGACCUMULATOR

- More generalized version of LongAdder
- Supports custom accumulation operations
- Uses LongBinaryOperator for calculations
- Also maintains internal variable set

LONGACCUMULATOR EXAMPLE

```
1 LongBinaryOperator op = (x, y) -> 2 * x + y;
2 LongAccumulator accumulator = new LongAccumulator(op, 1L);
3
4 ExecutorService executor = Executors.newFixedThreadPool(2);
5
6 IntStream.range(0, 10)
7     .forEach(i -> executor.submit(() -> accumulator.accumulate(i)));
8
9 stop(executor);
10
11 /* Prints 2539 */
12 System.out.println(accumulator.getThenReset());
```

CONCURRENTMAP

- Extends Map interface
- Thread-safe collection
- Enhanced in Java 8 with functional methods
- Key implementation: ConcurrentHashMap

SAMPLE MAP SETUP

```
1 ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();  
2 map.put("foo", "bar");  
3 map.put("han", "solo");  
4 map.put("r2", "d2");  
5 map.put("c3", "p0");
```

FOREACH OPERATION

```
1 map.forEach((key, value) -> System.out.printf("%s = %s\n", key, value));  
2 /* Sequential iteration on current thread */
```

PUTIFABSENT AND GETORDEFAULT

```
1 /* Thread-safe put if key doesn't exist */
2 String value = map.putIfAbsent("c3", "p1");
3 System.out.println(value);      /* p0 */
4
5 /* Returns default if key not found */
6 String value = map.getOrDefault("hi", "there");
7 System.out.println(value);      /* there */
```

REPLACE OPERATIONS

```
1 /* Replace all matching entries */
2 map.replaceAll((key, value) ->
3     "r2".equals(key) ? "d3" : value);
4
5 /* Transform single entry */
6 map.compute("foo", (key, value) -> value + value);
7
8 /* Merge values */
9 map.merge("foo", "boo",
10     (oldVal, newVal) -> newVal + " was " + oldVal);
```

CONCURRENTHASHMAP PARALLEL OPERATIONS

- Uses ForkJoinPool for parallel execution
- Three types of operations:
 - forEach - parallel iteration
 - search - parallel search
 - reduce - parallel reduction
- Controlled by parallelismThreshold

PARALLEL FOREACH

```
1 map.forEach(1, (key, value) ->
2     System.out.printf("key: %s; value: %s; thread: %s\n",
3         key, value, Thread.currentThread().getName()));
4
5 /* Output shows parallel execution:
6 key: r2; value: d2; thread: main
7 key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1
8 key: han; value: solo; thread: ForkJoinPool.commonPool-worker-2
9 key: c3; value: p0; thread: main */
```

PARALLEL REDUCE

```
1 String result = map.reduce(1,
2     (key, value) -> {
3         /* Transform step */
4         return key + "=" + value;
5     },
6     (s1, s2) -> {
7         /* Combine step */
8         return s1 + ", " + s2;
9     });
10
11 /* Result combines all entries */
```

JAVA GENERICS

- Provide type safety at compile time
- Eliminate code duplication
- Eliminate explicit casting
- Enable type-flexible classes and methods
- Support bounded type parameters

PROBLEM WITHOUT GENERICS

```
1 class IntegerPrinter {
2     private Integer thingToPrint;
3
4     public IntegerPrinter(Integer thingToPrint) {
5         this.thingToPrint = thingToPrint;
6     }
7
8     public void print() {
9         System.out.println(thingToPrint);
10    }
11 }
12
13 /* Would need separate classes for each type */
14 class DoublePrinter { ... }
15 class StringPrinter { ... }
```

GENERIC SOLUTION

```
1 class Printer<T> {
2     private T thingToPrint;
3
4     public Printer(T thingToPrint) {
5         this.thingToPrint = thingToPrint;
6     }
7
8     public void print() {
9         System.out.println(thingToPrint);
10    }
11 }
```

USING GENERIC CLASS

```
1 /* Integer printer */
2 Printer<Integer> intPrinter = new Printer<>(23);
3 intPrinter.print();
4
5 /* Double printer */
6 Printer<Double> doublePrinter = new Printer<>(33.5);
7 doublePrinter.print();
8
9 /* String printer */
10 Printer<String> stringPrinter = new Printer<>("Hello");
11 stringPrinter.print();
```

TYPE PARAMETER NAMING CONVENTIONS

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V - Additional types

MULTIPLE TYPE PARAMETERS

```
1 public interface Pair<K, V> {
2     K getKey();
3     V getValue();
4 }
5 public class OrderedPair<K, V> implements Pair<K, V> {
6     private K key;
7     private V value;
8
9     public OrderedPair(K key, V value) {
10        this.key = key;
11        this.value = value;
12    }
13    public K getKey() { return key; }
14    public V getValue() { return value; }
15 }
```

```
1 Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
2 Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
3
4 /* using diamond operator */
5 OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
6 OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
7
8 /* type parameter (that is, K or V) can also be parameterized type, like Box<Integer> */
9 OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...))
```

GENERIC METHODS

```
1 public class Util {  
2     public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
3         return p1.getKey().equals(p2.getKey()) &&  
4             p1.getValue().equals(p2.getValue());  
5     }  
6 }  
7  
8 public class Pair<K, V> {  
9     private K key;  
10    private V value;  
11    public Pair(K key, V value) {  
12        this.key = key;  
13        this.value = value;  
14    }  
15    public void setKey(K key) { this.key = key; }  
16    public void setValue(V value) { this.value = value; }  
17    public K getKey() { return key; }  
18    public V getValue() { return value; }  
19 }
```

GENERIC METHODS INVOCATION

```
1 /* Explicit type specification */
2 Pair<Integer, String> p1 = new Pair<>(1, "apple");
3 Pair<Integer, String> p2 = new Pair<>(2, "pear");
4 boolean same = Util.<Integer, String>compare(p1, p2);
5 /* with type inference (diamond operator) */
6 Pair<Integer, String> p1 = new Pair<>(1, "apple");
7 Pair<Integer, String> p2 = new Pair<>(2, "pear");
8 boolean same = Util.compare(p1, p2);
```

BOUNDED TYPE PARAMETERS

```
1 public class Box<T> {
2     private T t;
3     public void set(T t) { this.t = t; }
4     public T get() { return t; }
5
6     public <U extends Number> void inspect(U u){
7         System.out.println("T: " + t.getClass().getName());
8         System.out.println("U: " + u.getClass().getName());
9     }
10    public static void main(String[] args) {
11        Box<Integer> integerBox = new Box<Integer>();
12        integerBox.set(new Integer(10));
13        /* compilation fails, since our invocation of inspect still includes a String: */
14        integerBox.inspect("some text");
15    }
16 }
```

METHODS DEFINED IN BOUNDS

```
1 public class NaturalNumber<T extends Integer> {  
2  
3     private T n;  
4  
5     public NaturalNumber(T n) { this.n = n; }  
6  
7     public boolean isEven() {  
8         return n.intValue() % 2 == 0; /* intValue() method defined in the Integer class */  
9     }  
10 }
```

MULTIPLE BOUNDS

```
1 Class A { /* ... */ }
2 interface B { /* ... */ }
3 interface C { /* ... */ }
4
5 class D <T extends A & B & C> { /* ... */ }
6 class D <T extends B & A & C> { /* ... */ } // compile-time error
```

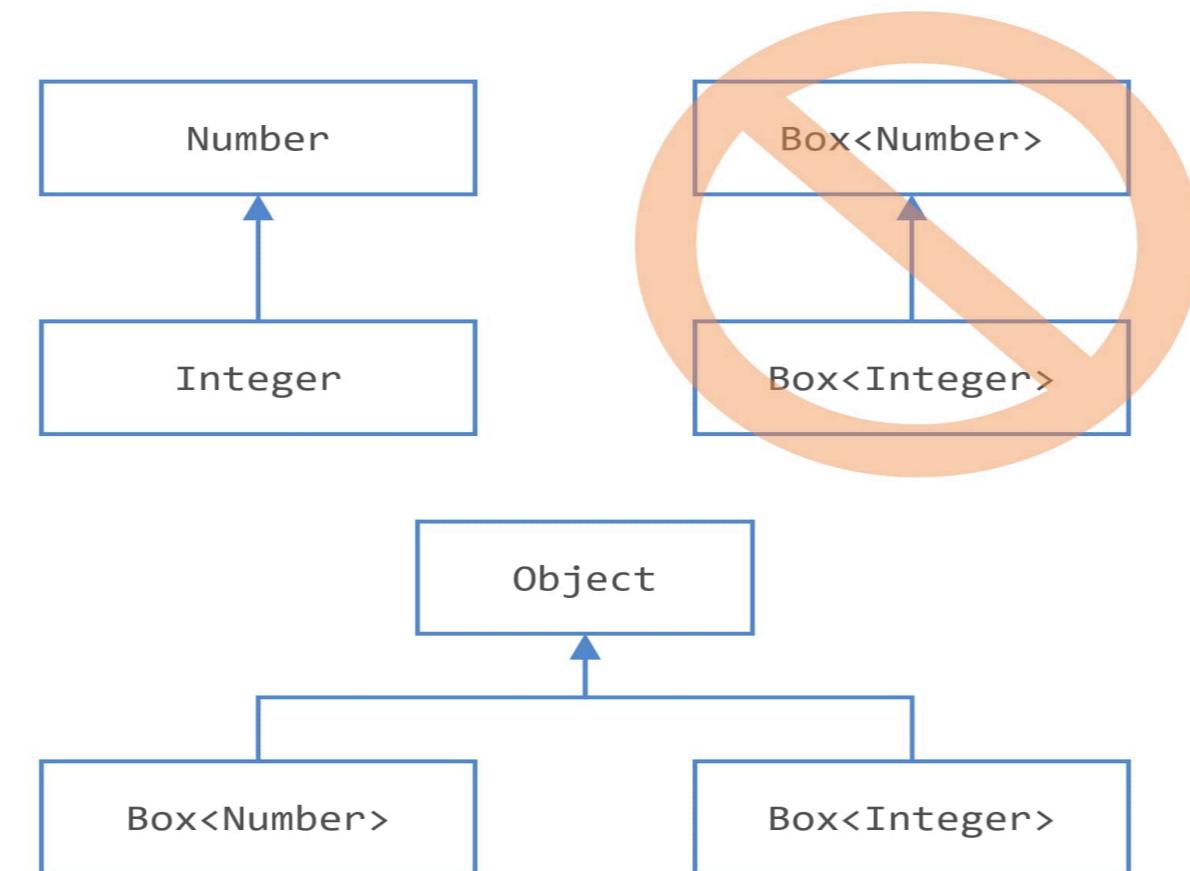
GENERICSS AND OBJECT INHERITANCE

```
1 /* Basic object inheritance */
2 Object someObject = new Object();
3 Integer someInteger = new Integer(10);
4 someObject = someInteger; /* OK - Integer "is a" Object */
5
6 /* Number inheritance example */
7 public void someMethod(Number n) { /* ... */ }
8 someMethod(new Integer(10)); /* OK */
9 someMethod(new Double(10.1)); /* OK */
```

GENERIC TYPE INHERITANCE

```
1 /* Generic Box with Number */
2 Box<Number> box = new Box<Number>();
3 box.add(new Integer(10));      /* OK */
4 box.add(new Double(10.1));    /* OK */
5
6 /* Method accepting Box<Number> */
7 public void boxTest(Box<Number> n) { /* ... */ }
```

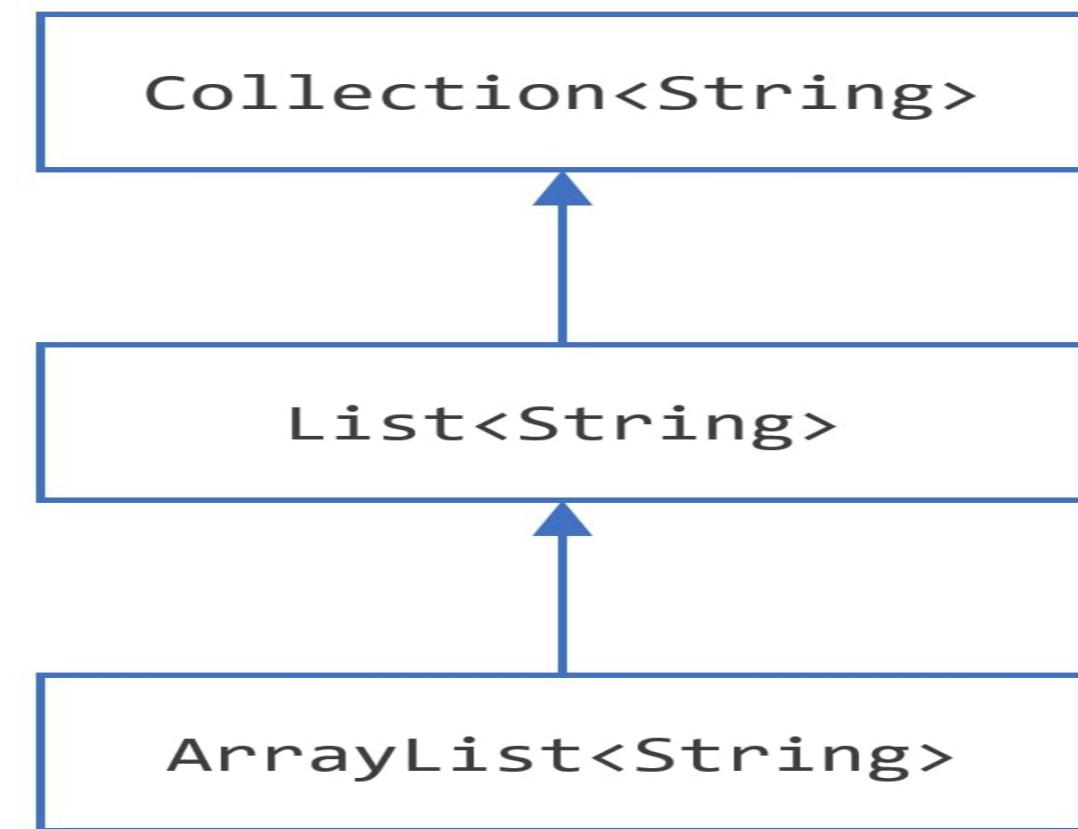
INVARIANCE



SUBTYPING RULES FOR GENERIC TYPES

```
1 /* MyClass<A> has no relationship to MyClass<B> */
2 MyClass<Number> numberBox;
3 MyClass<Integer> integerBox;
4 /* numberBox = integerBox;      // NOT allowed */
5 /* integerBox = numberBox;     // NOT allowed */
```

COLLECTIONS HIERARCHY



COLLECTIONS HIERARCHY EXAMPLE

```
1 ArrayList<String> arrayList = new ArrayList<>();
2 List<String> list = arrayList;          /* OK */
3 Collection<String> collection = list;  /* OK */
4
5 /* Type argument must remain the same */
6 ArrayList<Integer> intList = new ArrayList<>();
7 List<String> stringList = intList;      /* NOT allowed */
```

CUSTOM GENERIC INTERFACE INHERITANCE

```
1 interface PayloadList<E,P> extends List<E> {
2     void setPayload(int index, P val);
3 }
4
5 /* Valid subtypes of List<String>: */
6 PayloadList<String,String> p1;      /* OK */
7 PayloadList<String,Integer> p2;    /* OK */
8 PayloadList<String,Exception> p3;  /* OK */
```

UPPER BOUNDED WILDCARDS

```
1 /* Generic method with upper bounded wildcard */
2 public static void process(List<? extends Foo> list) {
3     for (Foo elem : list) {
4         /* Can use any method from Foo class on elem */
5     }
6 }
```

NUMERIC LIST EXAMPLE

```
1 public static double sumOfList(List<? extends Number> list) {  
2     double s = 0.0;  
3     for (Number n : list)  
4         s += n.doubleValue();  
5     return s;  
6 }  
7  
8 /* Usage with Integer list */  
9 List<Integer> li = Arrays.asList(1, 2, 3);  
10 System.out.println("sum = " + sumOfList(li));      /* prints sum = 6.0 */  
11  
12 /* Usage with Double list */  
13 List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);  
14 System.out.println("sum = " + sumOfList(ld));      /* prints sum = 7.0 */
```

KEY POINTS ABOUT UPPER BOUNDS

```
1 /* More restrictive */
2 public static void processNumbers(List<Number> list) { /* ... */ }
3
4 /* More flexible */
5 public static void processNumbers(List<? extends Number> list) { /* ... */ }
6
7 /* Accessing elements safely */
8 public static void processElements(List<? extends Foo> list) {
9     for (Foo elem : list) {
10         elem.someMethod();      /* OK: can call Foo's methods */
11     }
12 }
```

UNBOUNDED WILDCARDS INTRODUCTION

```
1 /* Limited version - only works with List<Object> */
2 public static void printList(List<Object> list) {
3     for (Object elem : list)
4         System.out.println(elem + " ");
5     System.out.println();
6 }
```

GENERIC PRINTING WITH WILDCARDS

```
1 /* Generic version - works with any List type */
2 public static void printList(List<?> list) {
3     for (Object elem: list)
4         System.out.print(elem + " ");
5     System.out.println();
6 }
7
8 /* Usage with different types */
9 List<Integer> li = Arrays.asList(1, 2, 3);
10 List<String> ls = Arrays.asList("one", "two", "three");
11 printList(li);      /* Works with Integer list */
12 printList(ls);      /* Works with String list */
```

LIST<OBJECT> VS LIST<?>

```
1 /* List<Object> allows adding any object */
2 List<Object> objectList = new ArrayList<>();
3 objectList.add(new Object());           /* OK */
4 objectList.add("some string");         /* OK */
5 objectList.add(42);                   /* OK */
6
7 /* List<?> only allows null */
8 List<?> wildcardList = new ArrayList<>();
9 wildcardList.add(null);               /* OK */
10 wildcardList.add(new Object());        /* Compile error */
11 wildcardList.add("string");          /* Compile error */
```

LOWER BOUNDED WILDCARDS

```
1 /* Method accepting lists of Integer or any supertype */
2 public static void addNumbers(List<? super Integer> list) {
3     for (int i = 1; i <= 10; i++) {
4         list.add(i);
5     }
6 }
7
8 /* Valid uses */
9 List<Integer> intList = new ArrayList<>();
10 List<Number> numList = new ArrayList<>();
11 List<Object> objList = new ArrayList<>();
12 addNumbers(intList);      /* Works with Integer */
13 addNumbers(numList);      /* Works with Number */
14 addNumbers(objList);      /* Works with Object */
```

PRODUCER VS CONSUMER GUIDELINES

```
1 /* Producer - uses extends */
2 public static void readElements(List<? extends Number> list) {
3     for (Number n : list) {
4         /* Can read elements as Number */
5         System.out.println(n.doubleValue());
6     }
7 }
8
9 /* Consumer - uses super */
10 public static void addElements(List<? super Integer> list) {
11     list.add(42);      /* Can add Integers safely */
12     list.add(123);    /* Can add Integers safely */
13 }
```

WILDCARD GUIDELINES SUMMARY

```
1 /* Producer (extends) - can read but can't add */
2 List<? extends Number> numbers = new ArrayList<>();
3 Number n = numbers.get(0);                      /* OK */
4 numbers.add(new Integer(42));                   /* Compile error */
5
6 /* Consumer (super) - can add but reading is limited */
7 List<? super Integer> integers = new ArrayList<>();
8 integers.add(42);                             /* OK */
9 integers.add(new Integer(123));                /* OK */
10 Integer n = integers.get(0);                  /* Compile error */
11 Object obj = integers.get(0);                 /* OK */
```

JAVA REFLECTION API

DEFINITION

- Examines/modifies runtime behavior
- Advanced feature for experienced developers
- Enables otherwise impossible operations
- Requires careful consideration

USES OF REFLECTION

EXAMPLE: BASIC REFLECTION

```
1 /* Get class information at runtime */
2 Class<?> clazz = Class.forName("com.example.MyClass");
3
4 /* Create new instance */
5 Object object = clazz.getDeclaredConstructor().newInstance();
6
7 /* Access methods */
8 Method method = clazz.getDeclaredMethod("myMethod");
9 method.setAccessible(true);
10 method.invoke(object);
```

DRAWBACKS: PERFORMANCE

- Dynamic type resolution limits JVM optimizations
- Slower than non-reflective operations
- Avoid in performance-critical code sections

DRAWBACKS: SECURITY

- Requires runtime permissions
- May be restricted under security managers
- Limited access to standard Java API classes
- Requires consideration in restricted contexts

DRAWBACKS: CODE RISKS

- Can access private fields/methods
- May cause unexpected side effects
- Could break encapsulation
- May affect code portability
- Behavior might change with platform updates

BEST PRACTICES

- Use only when necessary
- Prefer non-reflective alternatives
- Consider performance impact
- Be aware of security implications
- Maintain careful documentation
- Test thoroughly

RETRIEVING CLASSES IN REFLECTION

- Entry point is `java.lang.Class`
- No public constructors in `java.lang.reflect`
- Multiple ways to get `Class` objects
- Works with both reference and primitive types

GETTING CLASS FROM OBJECT

```
1 /* From String object */
2 Class<?> c = "foo".getClass();
3
4 /* From Console */
5 Class<?> c = System.console().getClass();
6
7 /* From Enum */
8 enum Days {SATURDAY, SUNDAY}
9 Class<?> c = Days.SATURDAY.getClass();
10
11 /* From Array */
12 byte[ ] bytes = new byte[1024];
13 Class<?> c = bytes.getClass();
```

GETTING CLASS FROM RUNTIME TYPES

```
1 /* From Set implementation */
2 Set<String> s = new HashSet<String>();
3 Class<?> c = s.getClass();
4
5 /* From Lambda */
6 Supplier<String> supplier = () -> "Hello";
7 Class<?> c = supplier.getClass();
8
9 /* From Anonymous Class */
10 Object key = new Object() {};
11 Class<?> c = key.getClass();
```

USING .CLASS SYNTAX

```
1 /* For primitive types */
2 Class<?> c = boolean.class;
3
4 /* For reference types */
5 Class<?> c = java.io.PrintStream.class;
6
7 /* For arrays */
8 Class<?> c = int[ ][ ][ ].class;
```

USING CLASS.FORNAME()

```
1 /* For regular classes */
2 Class<?> c = Class.forName("java.lang.String");
3
4 /* For arrays */
5 Class<?> simpleDoubleArray = Class.forName("[ D");
6 Class<?> bidiStringArray =
7     Class.forName("[[Ljava.lang.String;");
```

PRIMITIVE TYPES AND WRAPPERS

```
1 /* Using forPrimitiveName (Java SE 22+) */
2 Class<?> c = Class.forName("int");
3
4 /* Using wrapper TYPE field */
5 Class<?> c = Double.TYPE;
6 Class<?> v = Void.TYPE;
```

GETTING RELATED CLASSES

```
1 /* Get superclass chain */
2 Class<?> c = NullPointerException.class;
3 while (c != null) {
4     System.out.println(c);
5     c = c.getSuperclass();
6 }
7
8 /* Get implemented interfaces */
9 Class<?> c = String.class;
10 Class<?>[ ] interfaces = c.getInterfaces();
```

MEMBER CLASSES

```
1 /* Get public member classes */
2 Class<?> c = Character.class;
3 for (var memberClass: c.getClasses()) {
4     /* Returns public classes, interfaces,
5      records, and enums */
6     System.out.println(memberClass);
7 }
8
9 /* Get all declared member classes */
10 for (var memberClass: c.getDeclaredClasses()) {
11     /* Includes non-public members */
12     System.out.println(memberClass);
13 }
```

DECLARING AND ENCLOSING CLASSES

```
1 /* Get declaring class */
2 Class<?> c = Character.UnicodeBlock.class;
3 Class<?> declaringClass = c.getDeclaringClass();
4
5 /* Get enclosing class */
6 var key = new Object() {};
7 Class<?> anonymousClass = key.getClass();
8 Class<?> enclosingClass =
9     anonymousClass.getEnclosingClass();
```

BEST PRACTICES

- Choose appropriate method based on available information
- Consider performance implications
- Handle potential exceptions
- Use type-safe methods when possible
- Be aware of array class name syntax
- Consider security implications

READING CLASS NAMES

- Multiple methods to retrieve class names
- Each method serves different purposes
- Special handling for arrays
- Different behavior for anonymous/lambda classes

BASIC NAME RETRIEVAL

```
1 /* Getting fully qualified name */
2 Class<?> arrayList = Class.forName("java.util.ArrayList");
3 System.out.println(arrayList.getCanonicalName());
4
5 /* Different runtime classes */
6 List<String> strings1 = new ArrayList<>();
7 List<String> strings2 = Arrays.asList("one", "two");
8 List<String> strings3 = List.of("one", "two");
9
10 /* Prints different implementations */
11 System.out.println(strings1.getClass().getCanonicalName());
12 System.out.println(strings2.getClass().getCanonicalName());
13 System.out.println(strings3.getClass().getCanonicalName());
```

SIMPLE NAME

```
1 /* Regular class */
2 Class<?> c = String.class;
3 System.out.println(c.getSimpleName()); /* Prints: String */
4
5 /* Anonymous class - no simple name */
6 Object key = new Object() {};
7 System.out.println(key.getClass().getSimpleName()); /* Prints: "" */
8
9 /* Lambda class */
10 Supplier<String> supplier = () -> "Hello";
11 System.out.println(supplier.getClass().getSimpleName());
12 /* Prints: Main$$Lambda/... */
```

CANONICAL NAME

```
1 /* Regular class */
2 Class<?> c1 = String.class;
3 /* Prints: java.lang.String */
4 System.out.println(c1.getCanonicalName());
5
6 /* Anonymous class - no canonical name */
7 Object key = new Object() {};
8 /* Prints: null */
9 System.out.println(key.getClass().getCanonicalName());
10
11 /* Lambda - no canonical name */
12 Supplier<String> supplier = () -> "Hello";
13 /* Prints: null */
14 System.out.println(supplier.getClass().getCanonicalName());
```

TYPE NAME

```
1 /* Regular class */
2 Class<?> c1 = String.class;
3 /* Prints: java.lang.String */
4 System.out.println(c1.getTypeName());
5
6 /* Anonymous class */
7 Object key = new Object() {};
8 /* Prints: com.example.Main$1 */
9 System.out.println(key.getClass().getTypeName());
10
11 /* Lambda class */
12 Supplier<String> supplier = () -> "Hello";
13 /* Prints: com.example.Main$$Lambda/... */
14 System.out.println(supplier.getClass().getTypeName());
```

ARRAY NAMES

```
1 int[ ] array = {};
2 Class<?> c = array.getClass();
3
4 /* Different name formats */
5 System.out.println(c.getSimpleName());      /* int[ ] */
6 System.out.println(c.getTypeName());        /* int[ ] */
7 System.out.println(c.getCanonicalName());    /* int[ ] */
8 System.out.println(c.getName());            /* [ I */
```

ARRAY NAME CONVENTIONS

- Primitive Array Encoding:
- boolean → Z
- byte → B
- short → S
- int → I
- long → J
- char → C
- float → F
- double → D

COMPLEX ARRAY EXAMPLES

```
1 /* Multi-dimensional primitive array */
2 long[][] bidiLongs = new long[0][];
3 /* Prints: [[J */
4 System.out.println(bidiLongs.getClass().getName());
5
6 /* Multi-dimensional reference array */
7 String[][] bidiString = new String[0][];
8 /* Prints: [[Ljava.lang.String; */
9 System.out.println(bidiString.getClass().getName());
10
11 /* Getting array class from name */
12 Class<?> c = Class.forName("[[Ljava.lang.String;");
```

NAME METHOD SELECTION GUIDE

READING MODIFIERS

- Classes and members can have multiple modifiers
- Reflection API provides access to these modifiers
- Two main approaches:
 - `getModifiers()` method
 - `accessFlags()` method (Java SE 20+)

TYPES OF MODIFIERS

- Access modifiers
 - public, protected, private
- Inheritance modifiers
 - abstract, final, sealed, non-sealed
- Instance modifiers
 - static
- Behavior modifiers
 - strictfp
- Annotations

USING GETMODIFIERS()

```
1 /* Check String class modifiers */
2 Class<?> s = String.class;
3 int modifiers = s.getModifiers();
4 System.out.println(
5     Modifier.isFinal(modifiers));          /* true */
6 System.out.println(
7     Modifier.isAbstract(modifiers));      /* false */
8
9 /* Check Collection interface modifiers */
10 Class<?> c = Collection.class;
11 modifiers = c.getModifiers();
12 System.out.println(
13     Modifier.isFinal(modifiers));          /* false */
14 System.out.println(
15     Modifier.isAbstract(modifiers));      /* true */
```

MODIFIER CLASS METHODS

MEMBER INTERFACE

```
1 /* Getting modifiers from fields */
2 Field field = String.class.getDeclaredField("value");
3 int fieldMods = field.getModifiers();
4 System.out.println(
5     Modifier.isPrivate(fieldMods));      /* true */
6 System.out.println(
7     Modifier.isFinal(fieldMods));        /* true */
8
9 /* Getting modifiers from methods */
10 Method method = String.class.getMethod("length");
11 int methodMods = method.getModifiers();
12 System.out.println(
13     Modifier.isPublic(methodMods));    /* true */
```

ACCESS FLAGS (JAVA SE 20+)

```
1 /* Using new accessFlags() method */
2 Class<?> c = String.class;
3 Set<AccessFlag> accessFlags = c.accessFlags();
4 for (AccessFlag flag : accessFlags) {
5     System.out.println(flag);
6 }
7
8 /* Output:
9  PUBLIC
10 FINAL
11 SUPER */
```

BEST PRACTICES

- Use `accessFlags()` when available (Java SE 20+)
- Handle possible exceptions
- Check relevant modifiers for your use case
- Consider security implications
- Be aware of implicit modifiers
- interfaces are implicitly abstract
- interface methods are implicitly public

READING AND WRITING FIELDS

- Field objects provide access to:
- Field type information
- Field modifiers
- Field values
- Field updates

LOCATING FIELDS

```
1 /* Get specific declared field */
2 Field field = class.getDeclaredField("fieldName");
3
4 /* Get all declared fields */
5 Field[] fields = class.getDeclaredFields();
6
7 /* Get public fields (including inherited) */
8 Field[] publicFields = class.getFields();
```

FIELD LOCATION METHODS

OBTAINING FIELD TYPES

```
1 /* For non-generic fields */
2 Class<?> userClass = User.class;
3 Field nameField = userClass.getDeclaredField("name");
4 Class<?> type = nameField.getType();
5
6 /* For generic fields */
7 class Box<T> {
8     private T value;
9 }
10 Field valueField = Box.class.getDeclaredField("value");
11 Type genericType = valueField.getGenericType();
```

FIELD MODIFIERS

```
1 /* Using getModifiers */
2 Field field = userClass.getDeclaredField("name");
3 int modifiers = field.getModifiers();
4 System.out.println(Modifier.isPrivate(modifiers));
5 System.out.println(Modifier.isFinal(modifiers));
6
7 /* Using accessFlags (Java SE 20+) */
8 Set<AccessFlag> flags = field.accessFlags();
9 System.out.println(flags); /* [PRIVATE] */
```

GETTING FIELD VALUES

```
1 class User {  
2     private String name;  
3 }  
4  
5 User user = new User("John");  
6 Field nameField = User.class.getDeclaredField("name");  
7  
8 /* For non-accessible fields */  
9 nameField.setAccessible(true);  
10  
11 /* Get value */  
12 String name = (String) nameField.get(user);
```

SETTING FIELD VALUES

```
1 /* Setting field value */
2 nameField.setAccessible(true);
3 nameField.set(user, "Jane");
4
5 /* Setting final fields */
6 Field finalField = User.class.getDeclaredField("id");
7 finalField.setAccessible(true);
8 finalField.set(user, "newId"); /* Works even on final */
```

PRIMITIVE TYPE HANDLING

```
1 class User {  
2     private int age;  
3 }  
4  
5 User user = new User(25);  
6 Field ageField = User.class.getDeclaredField("age");  
7 ageField.setAccessible(true);  
8  
9 /* Using primitive-specific methods */  
10 int age = ageField.getInt(user);  
11 ageField.setInt(user, 26);  
12  
13 /* Using generic methods (with boxing) */  
14 Integer ageBoxed = (Integer) ageField.get(user);  
15 ageField.set(user, 27);
```

BEST PRACTICES

- Check field accessibility
- Handle security restrictions
- Use primitive-specific methods when possible
- Consider performance implications
- Handle potential exceptions:
 - NoSuchFieldException
 - IllegalAccessException
 - IllegalArgumentException

SECURITY CONSIDERATIONS

- `setAccessible(true)` bypasses access control
- Module system may restrict access
- Consider using proper encapsulation
- Document reflection usage
- Handle security manager restrictions

INVOKING METHODS WITH REFLECTION

- Method discovery and access
- Parameter and return type information
- Method invocation
- Exception handling
- Access control

LOCATING METHODS

```
1 /* Get specific method */
2 Method method = class.getMethod("methodName", paramTypes);
3
4 /* Get all public methods (including inherited) */
5 Method[] methods = class.getMethods();
6
7 /* Get all declared methods (including private) */
8 Method[] declared = class.getDeclaredMethods();
```

METHOD LOCATION SUMMARY

- `getMethod(name, paramTypes)`
- Public methods only
- Includes inherited methods
- `getDeclaredMethod(name, paramTypes)`
- All methods in class
- Includes private methods
- No inherited methods

GETTING RETURN TYPES

```
1 /* Basic return type */
2 Class<?> listClass = List.class;
3 Method getMethod = listClass.getMethod("get", int.class);
4 Class<?> returnType = getMethod.getReturnType();
5
6 /* Generic return type */
7 Type genericType = getMethod.getGenericReturnType();
```

METHOD PARAMETERS

```
1 /* Get parameter types */
2 Class<?>[ ] paramTypes = method.getParameterTypes();
3
4 /* Get parameter names (requires -parameters compiler flag) */
5 Parameter[ ] parameters = method.getParameters();
6 for (Parameter param : parameters) {
7     System.out.println(param.getName());
8 }
```

METHOD MODIFIERS

```
1 /* Using getModifiers */
2 Method method = String.class.getMethod("join",
3     CharSequence.class, Iterable.class);
4 int modifiers = method.getModifiers();
5
6 System.out.println("Is static: " +
7     Modifier.isStatic(modifiers));
8 System.out.println("Is public: " +
9     Modifier.isPublic(modifiers));
10
11 /* Using accessFlags (Java SE 20+) */
12 Set<AccessFlag> flags = method.accessFlags();
```

BASIC METHOD INVOCATION

```
1 /* No parameters */
2 String hello = "Hello";
3 Method lengthMethod =
4     String.class.getMethod("length");
5 int length = (int)lengthMethod.invoke(hello);
6
7 /* With parameters */
8 Method substringMethod =
9     String.class.getMethod("substring",
10         int.class, int.class);
11 String result = (String)substringMethod.invoke(
12     hello, 0, 5);
```

EXCEPTION HANDLING

```
1 try {
2     Method method = Files.class.getMethod(
3         "newBufferedWriter",
4         Path.class, OpenOption[].class);
5
6     Writer writer = (Writer)method.invoke(
7         null, path, new OpenOption[0]);
8 } catch (InvocationTargetException ite) {
9     /* Get the actual exception */
10    Throwable actual = ite.getTargetException();
11    actual.printStackTrace();
12 } catch (Exception e) {
13     /* Handle other reflection exceptions */
14     e.printStackTrace();
15 }
```

PRIVATE METHOD ACCESS

```
1 /* Access private method */
2 Method privateMethod =
3     class.getDeclaredMethod("privateName");
4
5 /* Enable access */
6 privateMethod.setAccessible(true);
7
8 /* Now can invoke */
9 Object result =
10    privateMethod.invoke(instance);
```

GENERICCS CONSIDERATIONS

```
1 /* Won't work due to type erasure */
2 List<String> list = new ArrayList<>();
3 Method addString = list.getClass()
4     .getMethod("add", String.class); /* Throws */
5
6 /* Correct approach */
7 Method addObject = list.getClass()
8     .getMethod("add", Object.class);
9 addObject.invoke(list, "string");
```

BEST PRACTICES

- Handle security restrictions
- Consider performance impact
- Cache Method objects for reuse
- Handle exceptions properly
- Be aware of type erasure
- Consider visibility rules
- Document reflection usage

READING ANNOTATIONS WITH REFLECTION

- Access runtime annotations
- Used extensively in frameworks
- Supported by AnnotatedElement interface
- Classes, methods, fields, and constructors

BASIC ANNOTATION STRUCTURE

```
1  /* Annotation definition */
2  @Target({ElementType.TYPE})
3  @Retention(RetentionPolicy.RUNTIME)
4  @interface Bean {}
5
6  @Target({ElementType.TYPE})
7  @Retention(RetentionPolicy.RUNTIME)
8  @interface Serialized {
9      SerializedFormat format() default SerializedFormat.JSON;
10 }
11
12 /* Using annotations */
13 @Serialized @Bean
14 public class Person { }
```

DISCOVERING ANNOTATIONS

```
1 Class<?> c = Person.class;
2
3 /* Check for specific annotation */
4 boolean isBean = c.isAnnotationPresent(Bean.class);
5
6 /* Get all annotations */
7 Annotation[ ] annotations = c.getAnnotations();
8
9 /* Get specific annotation */
10 Serialized serialized =
11     c.getAnnotation(Serialized.class);
12 SerializedFormat format = serialized.format();
```

REPEATING ANNOTATIONS

```
1 /* Container annotation */
2 @interface Validators {
3     Validator[] value();
4 }
5
6 /* Repeatable annotation */
7 @Repeatable(Validators.class)
8 @interface Validator {
9     ValidationRules value();
10 }
11
12 /* Usage */
13 public class Person {
14     @Validator(ValidationRules.NON_NULL)
15     @Validator(ValidationRules.NON_EMPTY)
16     private String name;
17 }
```

READING REPEATING ANNOTATIONS

```
1 /* Using getAnnotationsByType */
2 Class<?> c = Person.class;
3 Field nameField = c.getDeclaredField("name");
4
5 Validator[] validators =
6     nameField.getAnnotationsByType(Validator.class);
7
8 for (Validator validator : validators) {
9     System.out.println(validator.value());
10 }
```

INHERITED ANNOTATIONS

```
1 /* Inheritable annotation */
2 @Inherited
3 @interface Bean {}
4
5 /* Base class */
6 @Bean
7 class Person {}
8
9 /* Subclass inherits @Bean */
10 class User extends Person {}
11
12 /* Check inheritance */
13 Class<?> c = User.class;
14 System.out.println(c.getAnnotation(Bean.class) != null);
```

TYPE ANNOTATIONS

```
1 /* Type annotation definition */
2 @Target(ElementType.TYPE_USE)
3 @interface NonNull {}
4
5 /* Usage on type */
6 public class User
7     extends @NonNull Person
8     implements @NonNull Serializable {
9 }
10
11 /* Reading type annotations */
12 AnnotatedType superClass =
13     User.class.getAnnotatedSuperclass();
14 Annotation[] annotations =
15     superClass.getAnnotations();
```

PARAMETER ANNOTATIONS

```
1 public class Message {  
2     public Message(String name)  
3         throws @ReThrowAsRuntimeException EmptyStringException {  
4             // Implementation  
5         }  
6     }  
7  
8     /* Reading exception annotations */  
9     Constructor<?> constructor =  
10        Message.class.getConstructor(String.class);  
11     AnnotatedType[] exceptions =  
12        constructor.getAnnotatedExceptionTypes();
```

BEST PRACTICES

- Cache annotation queries for performance
- Handle missing annotations gracefully
- Consider security implications
- Document annotation requirements
- Test annotation processing
- Consider inheritance rules

ANNOTATION-BASED INTERCEPTORS

- Method interception pattern
- Uses runtime annotations
- Reflection for dynamic invocation
- Common in enterprise frameworks

INTERCEPTOR INTERFACE

```
1  /* Generic interceptor interface */
2  public interface Interceptor<T, R> {
3      R intercept(T interceptedObject,
4                  Method method,
5                  Object... arguments);
6  }
7
8  /* Interceptor annotation */
9  @Target(ElementType.METHOD)
10 @Retention(RetentionPolicy.RUNTIME)
11 public @interface Intercept {
12     Class<? extends Interceptor<?, ?>> value();
13 }
```

SERVICE IMPLEMENTATION

```
1 public class SomeInterceptedService {  
2  
3     @Intercept(MessageInterceptor.class)  
4     public String message(String input) {  
5         /* Original service method */  
6         return input.toUpperCase();  
7     }  
8 }  
9  
10 /* Non-intercepted service */  
11 public class SomeNonInterceptedService {  
12     public String message(String input) {  
13         return input.toUpperCase();  
14     }  
15 }
```

MESSAGE INTERCEPTOR

```
1 public class MessageInterceptor
2     implements Interceptor<SomeInterceptedService, String> {
3
4     @Override
5     public String intercept(
6         SomeInterceptedService service,
7         Method interceptedMethod,
8         Object... arguments) {
9         try {
10             /* Argument validation */
11             if (arguments.length != 1) {
12                 throw new IllegalArgumentException(
13                     "Exactly one argument required");
14             }
15
16             String input = (String) arguments[0];
17             Objects.requireNonNull(input, "Input is null");
18
19             /* Invoke original method */
20             String result = (String)interceptedMethod
21                 .invoke(service, arguments);
22
23             /* Modify result */
24             return result + " [was intercepted]";
25
26         } catch (Exception e) {
27             throw new RuntimeException(e);
28         }
29     }
30 }
```

SERVICE FACTORY

```
1 public class ServiceFactory {
2     public static <T, R> R invoke(
3         Class<? extends T> serviceClass,
4         String methodName,
5         Object... arguments) {
6     try {
7         /* Create service instance */
8         T service = serviceClass
9             .getConstructor()
10            .newInstance();
11
12         /* Find method */
13         Method method = serviceClass
14             .getDeclaredMethod(methodName,
15                 /* Get parameter types */
16                 Arrays.stream(arguments)
17                     .map(Object::getClass)
18                     .toArray(Class<?>[ ] ::new));
19
20         /* Check for interceptor */
21         if (method.isAnnotationPresent(
22             Intercept.class)) {
23
24             /* Get interceptor class */
25             Intercept intercept = method
26                 .getDeclaredAnnotation(
27                     Intercept.class);
```

USING THE INTERCEPTOR

```
1 /* Intercepted call */
2 String interceptedOutput = ServiceFactory
3     .invoke(SomeInterceptedService.class,
4         "message",
5         "Hello");
6
7 /* Non-intercepted call */
8 String normalOutput = ServiceFactory
9     .invoke(SomeNonInterceptedService.class,
10        "message",
11        "Hello");
12
13 /* Output:
14    HELLO [was intercepted]
15    HELLO */
```

INTERCEPTOR CAPABILITIES

- Pre/post method execution == Dependency Injection Framework
- Automated object initialization
- Manages object lifecycles
- Handles dependencies
- Uses reflection for discovery

BASIC BEAN FACTORY

```
1  /* Bean factory implementation */
2  public enum BeanFactory {
3      INSTANCE;
4
5      public <T> T getInstanceOf(
6          Class<T> beanClass,
7          Object... arguments) {
8          try {
9              /* Get parameter types */
10             Class<?>[ ] paramTypes = Arrays.stream(arguments)
11                 .map(Object::getClass)
12                 .toArray(Class<?>[ ]::new);
13
14             /* Find constructor */
15             Constructor<T> constructor =
16                 beanClass.getConstructor(paramTypes);
17
18             /* Create instance */
19             return constructor.newInstance(arguments);
20         } catch (Exception e) {
21             throw new RuntimeException(e);
22         }
23     }
24 }
```

SINGLETON SUPPORT

```
1 /* Singleton annotation */
2 @Target(ElementType.TYPE)
3 @Retention(RetentionPolicy.RUNTIME)
4 @interface Singleton {}
5
6 /* Example singleton bean */
7 @Singleton
8 public class DBService {
9     /* Implementation */
10 }
11
12 /* Usage */
13 DBService service1 = factory.getInstanceOf(DBService.class);
14 DBService service2 = factory.getInstanceOf(DBService.class);
15 /* service1 == service2 */
```

SINGLETON REGISTRY

```
1 public enum BeanFactory {
2     INSTANCE;
3
4     private final ConcurrentHashMap<Class<?>, Object>
5         registry = new ConcurrentHashMap<>();
6
7     public <T> T getInstanceOf(
8         Class<T> beanClass,
9         Object... arguments) {
10
11     /* Check for singleton annotation */
12     if (beanClass.isAnnotationPresent(
13         Singleton.class)) {
14
15         /* Return existing instance if present */
16         if (registry.containsKey(beanClass)) {
17             return (T) registry.get(beanClass);
18         }
19
20         /* Create new instance */
21         T bean = instantiateBeanClass(
22             beanClass, arguments);
23
24         /* Thread-safe registration */
25         registry.putIfAbsent(beanClass, bean);
26         return (T) registry.get(beanClass);
27     }
```

DEPENDENCY INJECTION

```
1 /* Injection annotation */
2 @Target(ElementType.FIELD)
3 @Retention(RetentionPolicy.RUNTIME)
4 @interface Inject {}
5
6 /* Example usage */
7 @Singleton
8 public class MyApplication {
9     @Inject
10    private DBService dbService;
11
12    public boolean isDBServiceSet() {
13        return dbService != null;
14    }
15 }
```

FIELD INJECTION LOGIC

```
1 private <T> T instantiateBeanClass(
2     Class<T> beanClass,
3     Object[ ] arguments) throws Exception {
4
5     /* Create instance */
6     T bean = createInstance(beanClass, arguments);
7
8     /* Get injectable fields */
9     Field[ ] fields = beanClass.getDeclaredFields();
10
11    /* Process @Inject fields */
12    for (Field field : fields) {
13        if (field.isAnnotationPresent(Inject.class)) {
14            /* Get field type */
15            Class<?> fieldType = field.getType();
16
17            /* Create dependency */
18            Object dependency =
19                getInstanceOf(fieldType);
20
21            /* Set field value */
22            field.setAccessible(true);
23            field.set(bean, dependency);
24        }
25    }
26
27    return bean;
```

USING THE FRAMEWORK

```
1 /* Create factory */
2 BeanFactory factory = BeanFactory.INSTANCE;
3
4 /* Get application instance */
5 MyApplication app =
6     factory.getInstanceOf(MyApplication.class);
7
8 /* Dependencies automatically injected */
9 System.out.println(
10     "DBService injected: " +
11     app.isDBServiceSet());
```

THREAD SAFETY CONSIDERATIONS

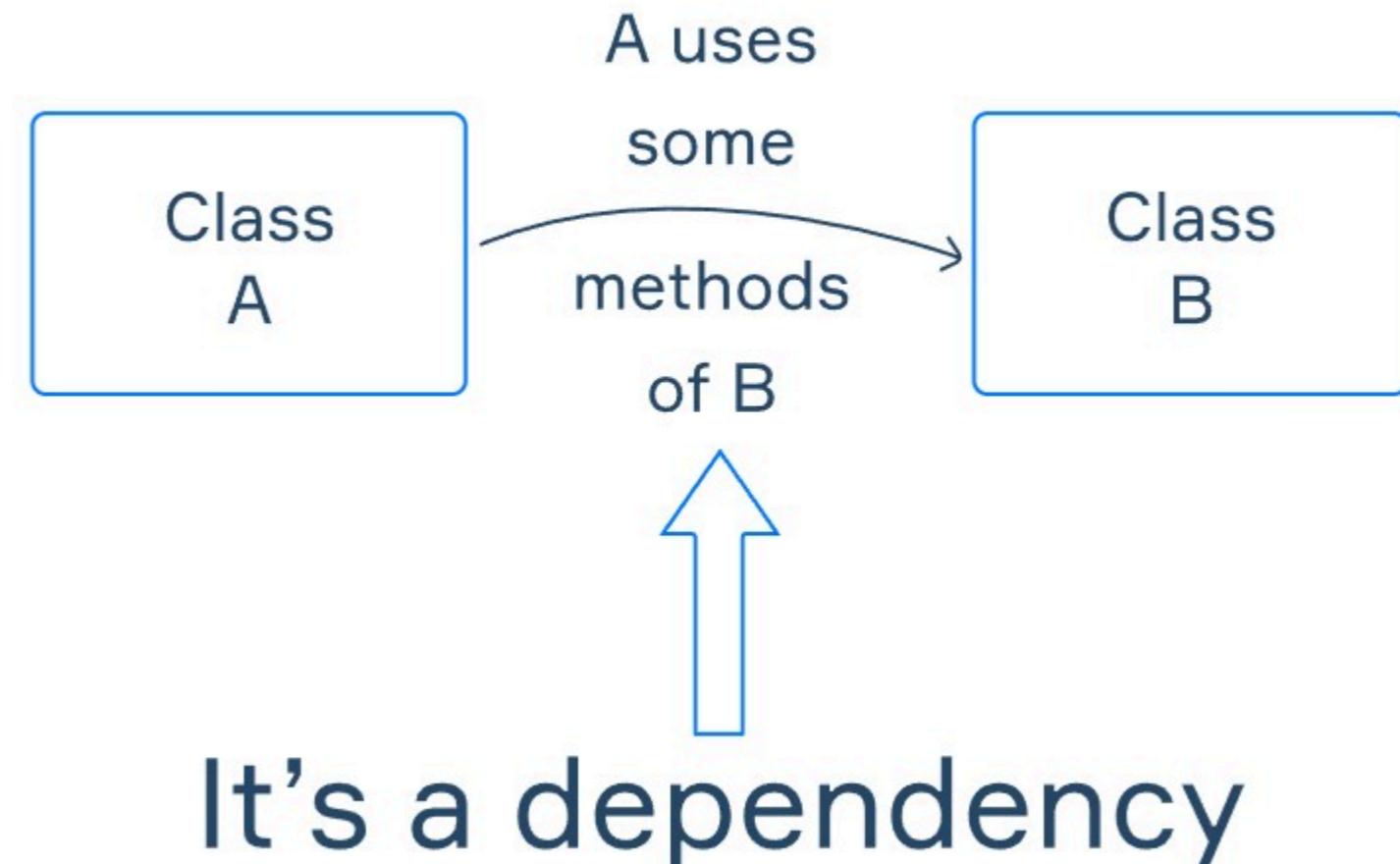
- Use ConcurrentHashMap for registry
- Atomic operations for singleton management
- Handle concurrent instantiation
- Consider double-check locking
- Proper synchronization for field injection

BEST PRACTICES

- Cache reflective lookups
 - Handle circular dependencies
 - Clear error messages
 - Document usage patterns
 - Consider lazy initialization
 - Proper exception handling
-

PREREQUISITES

DEPENDENCY INJECTION



EXAMPLE

```
1 class Email is
2     method sendMessage() is
3         ...
4
5 class EmailService is
6     Email gmail = new Email()
7
8     method sendMessage() is
9         gmail.sendMessage()
```

TYPES OF INJECTION (METHOD)

Method(Interface) Injection — dependencies are passed through methods that the class can access via interface or another class

```
1 interface Email is
2     method sendMessage()
3
4
5 class GmailService implements Email is
6     method sendMessage() is ...
7 ...
8
9
10 class EmailService is
11     method sendMessage(Email service) is
12         service.sendMessage()
```

```
1 EmailService gmail = new EmailService()
2 gmail.sendMessage(new GmailService())
```

TYPES OF INJECTION (PROPERTY)

Property(Setter) Injection — dependencies are passed through a separate setter method

```
1 class EmailService is
2   Email service
3
4   method getService() is
5     return service
6
7   method setService(Email service) is
8     this.service = service
9
10
11 class Main is
12   EmailService email = new EmailService()
13   email.setService(new GmailService())
14
15   Email gmail = email.getService()
16   gmail.sendMessage()
```

TYPES OF INJECTION (CONSTRUCTOR)

Constructor Injection — dependencies are provided through the class constructor

```
1 class EmailService is
2     Email email
3
4     constructor of EmailService(Email email) is
5         this.email = email
6
7     method sendMessage() is
8         email.sendMessage()
9
10
11 class Main is
12     EmailService gmail = new EmailService(new GmailService())
13     gmail.sendMessage()
```

COUPLING AND COHESION

COUPLING

Coupling is the degree to which software components depend on each other.

There are two basic levels of coupling: *tight* and *loose*.

TIGHT COUPLING EXAMPLE

```
1 class Engine is
2     method run() is
3
4 class Robot is
5     Engine eng = new Engine()
6     eng.run()
```

LOOSE COUPLING EXAMPLE

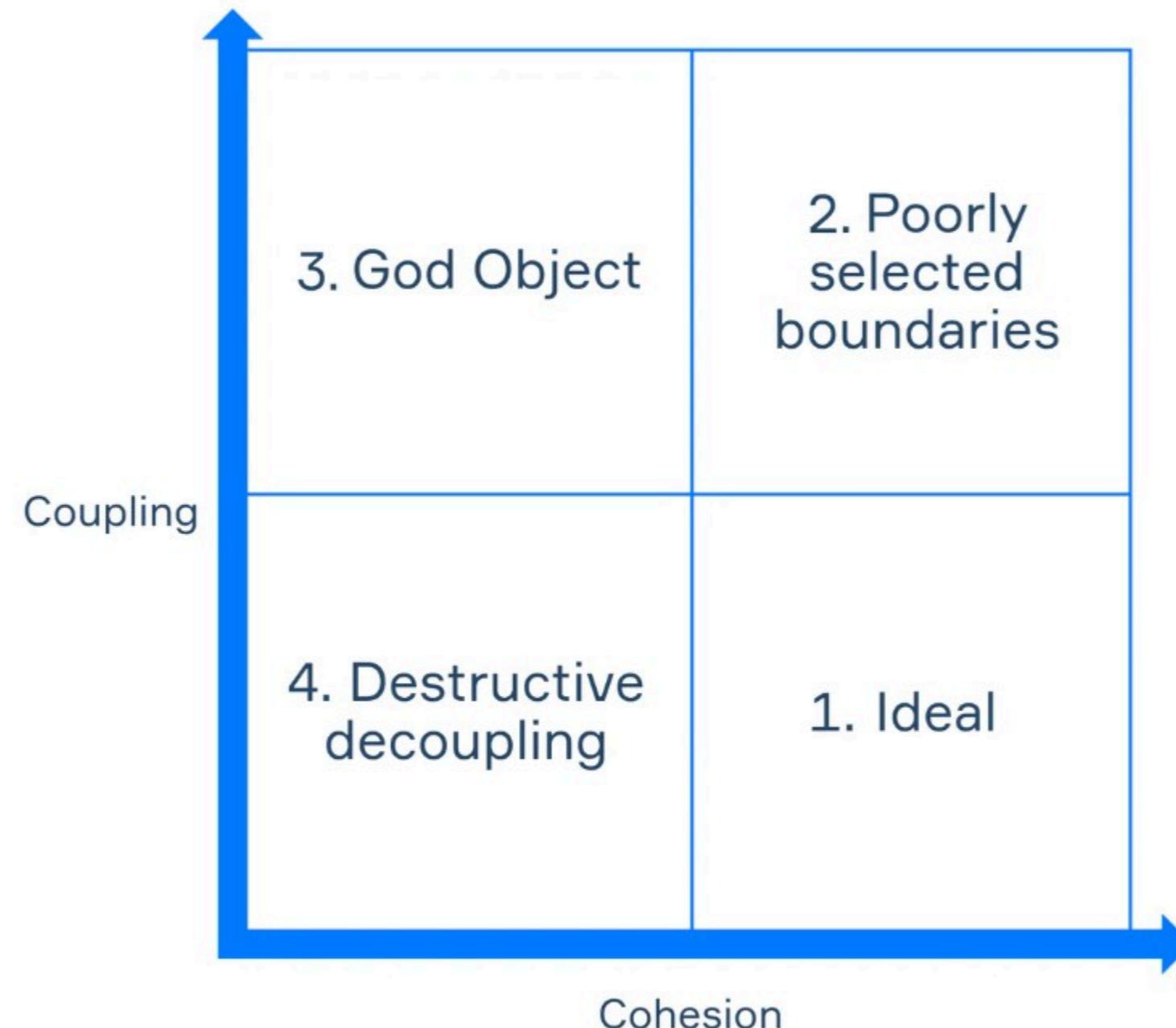
```
1 interface Paint is
2     method paint()
3
4 class Silver implements Paint is
5     method paint() is
6         print("silver")
7
8 class Gold implements Paint is
9     method paint() is
10        print("gold")
11
12 class Robot is
13     Paint col = new Gold()
14     col.paint()
```

COHESION

```
1 class RobotFactory is
2     method createRobots() is ...
3
4     method maintainLogistics() is ...
5
6     method manage() is ...
```

```
1 class BuildDepart is
2     method createRobots() is
3         ...
4
5
6 class Logistics is
7     method maintainLogistics() is
8         ...
9
10
11 class Management is
12     method manage() is
13         ...
```

WORKING WITH COHESION AND COUPLING



THE STRUCTURE OF SPRING APPLICATIONS

APPLICATION LAYERS

WHAT IS A LAYERED ARCHITECTURE?

1. The **presentation** layer handles the interface that is displayed to the user.
2. The **business** layer implements rules for handling the problem your application was designed to solve.
3. The **persistence** layer contains database storage logic and handles the translation of objects into database formats.
4. The **database** layer contains the actual database storage system and handles tables, indices, and any database-related operations.

IMPLEMENTING LAYERED ARCHITECTURES

```
1 # Setup for the H2 console for viewing data in the database
2 spring.h2.console.enabled=true
3 spring.h2.console.path=/h2
4
5 # H2 data source setup
6 spring.datasource.url=jdbc:h2:file:~/test
7 spring.datasource.username=sa
8 spring.datasource.password=
9
10 # Automatically update tables when persistence objects have changed
11 spring.jpa.hibernate.ddl-auto=update
```

IMPLEMENTING LAYERED ARCHITECTURES

```
1 package com.example.demo2.businesslayer;
2
3 import jakarta.persistence.*;
4
5 @Entity
6 @Table(name = "users")
7 public class User {
8     @Id
9     private long id;
10    @Column(name = "username")
11    private String username;
12    @Column(name = "firstName")
13    private String firstName;
14    @Column(name = "lastName")
15    private String lastName;
16
17    public User() {
18    }
19    public User(long id, String username, String firstName, String lastName) {
20        this.id = id;
21        this.username = username;
22        this.firstName = firstName;
23        this.lastName = lastName;
24    }
25    // getters and setters
26 }
```

IMPLEMENTING LAYERED ARCHITECTURES

```
1 package com.example.demo2.persistence;  
2  
3 import org.springframework.data.repository.CrudRepository;  
4 import org.springframework.stereotype.Repository;  
5 import com.example.demo2.businesslayer.User;  
6  
7 @Repository  
8 public interface UserRepository extends CrudRepository<User, Long> {  
9     User findUserById(Long id);  
10 }
```

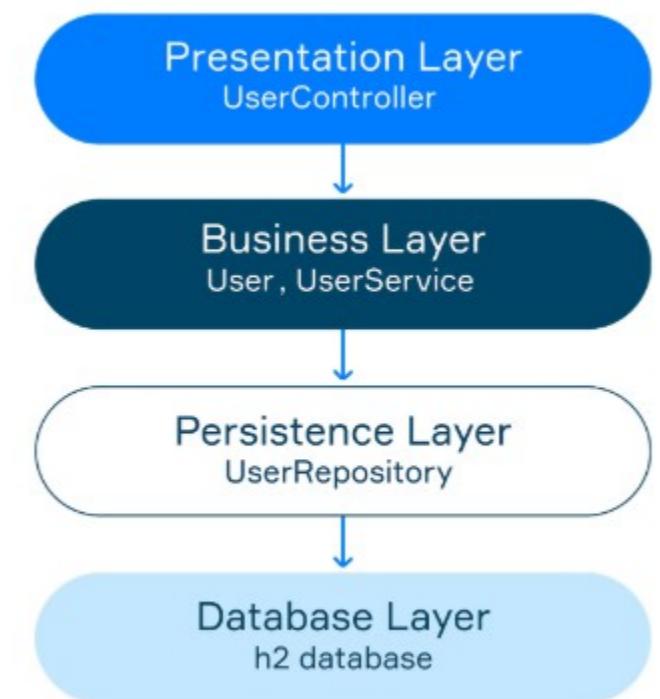
IMPLEMENTING LAYERED ARCHITECTURES

```
1 package com.example.demo2.businesslayer;  
2  
3 import org.springframework.stereotype.Service;  
4 import org.springframework.beans.factory.annotation.Autowired;  
5 import com.example.demo2.persistence.UserRepository;  
6  
7 @Service  
8 public class UserService {  
9     private final UserRepository userRepository;  
10  
11     @Autowired  
12     public UserService(UserRepository userRepository) {  
13         this.userRepository = userRepository;  
14     }  
15  
16     public User findUserById(Long id) {  
17         return userRepository.findUserById(id);  
18     }  
19  
20     public User save(User toSave) {  
21         return userRepository.save(toSave);  
22     }  
23 }
```

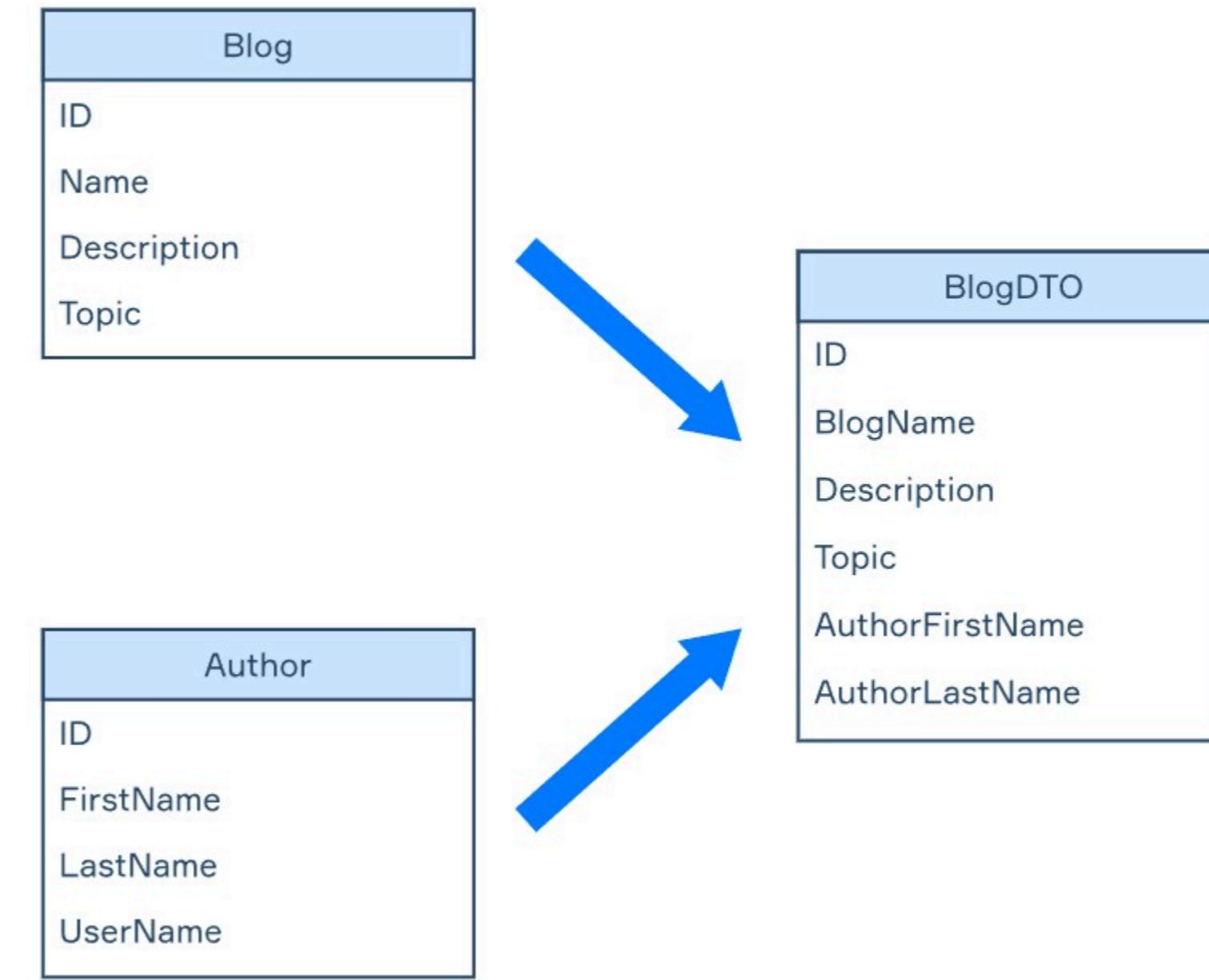
IMPLEMENTING LAYERED ARCHITECTURES

```
1 package com.example.demo2.presentation;
2
3 import org.springframework.web.bind.annotation.*;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import com.example.demo2.businesslayer.UserService;
6 import com.example.demo2.businesslayer.User;
7
8 @RestController
9 public class UserController {
10     @Autowired
11     UserService userService;
12
13     @PostMapping("/user")
14     public User saveUser(@RequestBody User user) {
15         User createdUser = userService.save(new User(
16             user.getId(), user.getUsername(),
17             user.getFirstName(), user.getLastName()));
18
19         return createdUser;
20     }
21
22     @GetMapping("/user/{id}")
23     public User getUser(@PathVariable long id) {
24         return userService.findUserById(id);
25     }
26 }
```

CONCLUSION



DATA TRANSFER OBJECT IN SPRING



IMPLEMENTATION: MAPPERS

```
1 public class User {  
2     private int id;  
3     private String name;  
4     private String email;  
5     private LocalDate accountCreatedAt;  
6  
7     // constructors, getters and setters  
8 }
```

```
1 public class UserDTO {  
2     private int id;  
3     private String name;  
4     private String email;  
5  
6     // constructors, getters, and setters  
7 }
```

IMPLEMENTATION: MAPPERS

```
1 UserDTO convertUserToDTO(User user) {  
2     return new UserDTO(user.getId(), user.getName(), user.getEmail());  
3 }
```

```
1 User convertDTOToUser(UserDTO dto) {  
2     User user = new User(dto.getId(), dto.getName(), dto.getEmail(), null);  
3     user.setAccountCreatedAt(LocalDate.now());  
4     return user;  
5 }
```

IMPLEMENTATION: MAPPERS

```
1 @Component
2 public class MyMapper {
3
4     UserDTO convertUserToDTO(User user) {
5         return new UserDTO(user.getId(), user.getName(), user.getEmail());
6     }
7
8     User convertDTOToUser(UserDTO dto) {
9         User user = new User(dto.getId(), dto.getName(), dto.getEmail(), null);
10        user.setAccountCreatedAt(LocalDate.now());
11        return user;
12    }
13 }
```

IMPLEMENTATION: MAPPERS

```
1 public UserService(UserRepository userRepository, MyMapper myMapper) {  
2     this.userRepository = userRepository;  
3     this.myMapper = myMapper;  
4 }
```

```
1 public List<UserDTO> getAllUsers() {  
2     return userRepository.findAll().stream()  
3             /* here we use the mapper to transform a User into a UserDTO */  
4             .map(myMapper::convertUserToDTO)  
5             .collect(Collectors.toList());  
6 }
```

IMPLEMENTATION: MAPPERS

```
1 dependencies {  
2     implementation 'org.modelmapper:modelmapper:3.1.0'  
3 }
```

```
1 @Bean  
2 public ModelMapper modelMapper() {  
3     return new ModelMapper();  
4 }
```

IMPLEMENTATION: MAPPERS

```
1  @Component
2  public class MyMapper {
3
4      private final ModelMapper modelMapper;
5
6      public MyMapper(ModelMapper modelMapper) {
7          this.modelMapper = modelMapper;
8      }
9
10     UserDTO convertUserToDTO(User user) {
11         /* here we make use of the 3rd party library to transform a User into a UserDTO
12         return modelMapper.map(user, UserDTO.class);
13     }
14
15     /* convertDTOToUser follows a similar implementation! Give it a try! */
16 }
```

LOMBOK VS. JAVA RECORDS

```
1 @NoArgsConstructor
2 @AllArgsConstructor
3 @Getter
4 @Setter
5 public class UserDTO {
6     private int id;
7     private String username;
8     private LocalDate dateOfBirth;
9 }
```

```
1 public record UserDTO(int id, String username, LocalDate dateOfBirth) {
2 }
```

SPRING TESTING

INTRODUCTION TO SPRING TESTING

- Types of Tests:
 - *Unit Tests*: Testing one component (bean) in isolation.
 - *Integration Tests*: Testing multiple components together according to their dependencies.
- Testing with Maven/Gradle
- Libraries for Testing in Spring Boot:
 - *JUnit*: The standard library for testing Java/Kotlin applications, which can also be used for Spring applications.
 - *Spring Testing Framework*: Provides tools for working with container-managed objects.
 - *spring-boot-starter-test*: Includes JUnit, Spring Test Framework, Mockito, AssertJ, Hamcrest, and other libraries for testing.

SETTING UP SPRING BOOT TESTS

LOADING APPLICATION CONTEXT

```
1 dependencies {  
2     testImplementation 'org.springframework.boot:spring-boot-starter-test'  
3 }
```

```
1 package eu.mithril;  
2  
3 import org.junit.jupiter.api.Test;  
4 import org.junit.jupiter.api.extension.ExtendWith;  
5 import org.springframework.beans.factory.annotation.Autowired;  
6 import org.springframework.context.ApplicationContext;  
7 import org.springframework.test.context.junit.jupiter.SpringExtension;  
8 import static org.assertj.core.api.Assertions.assertThat;  
9  
10 @ExtendWith(SpringExtension.class)  
11 class ApplicationTests {  
12  
13     private final ApplicationContext applicationContext;  
14  
15     @Autowired  
16     ApplicationTests(ApplicationContext applicationContext) {  
17         this.applicationContext = applicationContext;  
18     }  
19  
20     @Test  
21     void contextLoads() {  
22         assertThat(applicationContext).isNotNull();  
23     }  
24 }
```

LOADING APPLICATION CONTEXT

```
1 package eu.mithril.config;  
2  
3 import org.springframework.context.annotation.Bean;  
4 import org.springframework.context.annotation.Configuration;  
5  
6 @Configuration  
7 public class Config {  
8     @Bean  
9     public String customBean(){  
10         return "custom bean";  
11     }  
12 }
```

LOADING APPLICATION CONTEXT

```
1 package eu.mithril;
2
3 import eu.mithril.config.Config;
4 import org.junit.jupiter.api.Test;
5 import org.junit.jupiter.api.extension.ExtendWith;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.context.ApplicationContext;
8 import org.springframework.test.context.ContextConfiguration;
9 import org.springframework.test.context.junit.jupiter.SpringExtension;
10 import static org.assertj.core.api.Assertions.assertThat;
11
12 @ExtendWith(SpringExtension.class)
13 @ContextConfiguration(classes = Config.class)
14 class ApplicationTests {
15
16     private final ApplicationContext applicationContext;
17
18     @Autowired
19     ApplicationTests(ApplicationContext applicationContext) {
20         this.applicationContext = applicationContext;
21     }
22
23     @Test
24     void contextLoads() {
25         assertThat(applicationContext).isNotNull();
26         assertThat(applicationContext.getBeanDefinitionNames())
27             .contains("customBean", "config");
28     }
29 }
```

LOADING APPLICATION CONTEXT

```
1 package eu.mithril;
2
3 import eu.mithril.config.Config;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import static org.assertj.core.api.Assertions.assertThat;
7
8 @SpringJUnitConfig(Config.class)
9 class ApplicationTests {
10
11     private final String customBean;
12
13     @Autowired
14     ApplicationTests(String customBean) {
15         this.customBean = customBean;
16     }
17
18     @Test
19     void contextLoads() {
20         assertThat(customBean).isEqualTo("custom bean");
21     }
22 }
```

@SPRINGBOOTTEST

```
1 package eu.mithril;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(Application.class, args);
11     }
12 }
```

@SPRINGBOOTTEST

```
1 package eu.mithril;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import org.springframework.context.ApplicationContext;
7 import static org.assertj.core.api.Assertions.assertThat;
8
9 @SpringBootTest
10 class ApplicationTests {
11
12     private final ApplicationContext applicationContext;
13
14     @Autowired
15     ApplicationTests(ApplicationContext applicationContext) {
16         this.applicationContext = applicationContext;
17     }
18
19     @Test
20     void contextLoads() {
21         assertThat(applicationContext.getBeanDefinitionCount()).isGreaterThanOrEqualTo(10);
22     }
23 }
```

CONTEXT CACHING

```
1 package eu.mithril.example;
2
3 import org.springframework.stereotype.Component;
4 import java.util.List;
5
6 @Component
7 public class PizzaMenu {
8     private final List<String> pizzas = List.of("margherita", "mushrooms and vegetables");
9
10    public boolean isOnMenu(String name) {
11        return pizzas.contains(name);
12    }
13 }
```

```
1 package eu.mithril.example;
2
3 import org.springframework.stereotype.Component;
4 import java.util.List;
5
6 @Component
7 public class DessertMenu {
8
9     private final List<String> desserts = List.of("apple pie", "almond cake");
10
11    public boolean isOnMenu(String name) {
12        return desserts.contains(name);
13    }
14 }
```

CONTEXT CACHING

```
1 package eu.mithril.example;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class PizzeriaService {
8
9     private final PizzaMenu pizzaMenu;
10
11     @Autowired
12     public PizzeriaService(PizzaMenu pizzaMenu) {
13         this.pizzaMenu = pizzaMenu;
14     }
15
16     public String orderPizza(String name) {
17         if (pizzaMenu.isOnMenu(name)) {
18             System.out.println("Thanks for the order! Your pizza will be ready in 15 minutes");
19             return name;
20         } else {
21             System.out.println("We don't have such pizza on our menu");
22             return null;
23         }
24     }
25 }
```

CONTEXT CACHING

```
1 package eu.mithril.example;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class CafeService {
8
9     private final PizzaMenu pizzaMenu;
10    private final DessertMenu dessertMenu;
11
12    @Autowired
13    public CafeService(PizzaMenu pizzaMenu, DessertMenu dessertMenu) {
14        this.pizzaMenu = pizzaMenu;
15        this.dessertMenu = dessertMenu;
16    }
17
18    public String orderFood(String name) {
19        if (pizzaMenu.isOnMenu(name) || dessertMenu.isOnMenu(name)) {
20            System.out.println("Thanks for the order. Your " + name + " will be ready soon");
21            return name;
22        } else {
23            System.out.println("We don't have such food on our menu");
24            return null;
25        }
26    }
27 }
```

CONTEXT CACHING

```
1 package eu.mithril.example;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import static org.assertj.core.api.Assertions.assertThat;
7
8 @SpringBootTest
9 class PizzeriaServiceTest {
10
11     private final PizzeriaService pizzeriaService;
12     private final ApplicationContext applicationContext;
13
14     @Autowired
15     PizzeriaServiceTest(PizzeriaService pizzeriaService, ApplicationContext applicationContext) {
16         this.pizzeriaService = pizzeriaService;
17         this.applicationContext = applicationContext;
18     }
19
20     @Test
21     void orderPizza() {
22         System.out.println("PizzeriaService context hash code: " +
23             System.identityHashCode(applicationContext));
24         assertThat(pizzeriaService.orderPizza("pepperoni")).isNull();
25     }
26 }
```

CONTEXT CACHING

```
1 package eu.mithril.example;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import org.springframework.context.ApplicationContext;
7 import static org.assertj.core.api.Assertions.assertThat;
8
9 @SpringBootTest
10 class CafeServiceTest {
11
12     private final CafeService cafeService;
13     private final ApplicationContext applicationContext;
14
15     @Autowired
16     CafeServiceTest(CafeService cafeService, ApplicationContext applicationContext) {
17         this.cafeService = cafeService;
18         this.applicationContext = applicationContext;
19     }
20
21     @Test
22     void orderFood() {
23         System.out.println("CafeService context hash code: " +
24             System.identityHashCode(applicationContext));
25         assertThat(cafeService.orderFood("apple pie")).isNotNull();
26     }
27 }
```

CONTEXT CACHING

```
1 CafeService context hash code: 1243495105
2 Thanks for the order. Your apple pie will be ready soon
3
4 -----
5
6 PizzeriaService context hash code: 1243495105
7 We don't have such pizza on our menu
```

TESTING BEANS

PREPARATION

```
1 public class Book {  
2  
3     private Long id;  
4     private String name;  
5     private String author;  
6     private int pageCount;  
7  
8     // getters, setters, constructor  
9 }
```

```
1 @Component  
2 public class BookService {  
3  
4     public List<Book> getBooksByAuthor(String author) {  
5         /* should access db and return all books with a specific author */  
6         /* but in our example it just will return a hard-coded list of books */  
7         return List.of(  
8             new Book("title1", author, 100),  
9             new Book("title2", author, 200),  
10            new Book("title3", author, 300));  
11        }  
12    }
```

PREPARATION

```
1 @Component
2 public class BookStatisticsService {
3
4     private final BookService bookService;
5
6     @Autowired
7     public BookStatisticsService(BookService bookService) {
8         this.bookService = bookService;
9     }
10
11    public int getTotalPagesByAuthor(String author) {
12        return bookService.getBooksByAuthor(author)
13            .stream()
14            .mapToInt(Book::getPageCount)
15            .sum();
16    }
17 }
```

TESTING WITH DEPENDENCIES

Testing without Spring:

```
1 import static org.assertj.core.api.Assertions.assertThat;
2
3 public class BookStatisticsServiceTest {
4     private BookService bookService = new BookService();
5     private BookStatisticsService bookStatisticsService = new BookStatisticsService(bookService);
6
7     @Test
8     void shouldReturnTotalPages() {
9         int actual = bookStatisticsService.getTotalPagesByAuthor("Jane Austen");
10        int expected = 600;
11        assertThat(actual).isEqualTo(expected);
12    }
13 }
```

TESTING WITH DEPENDENCIES

```
1 import static org.assertj.core.api.Assertions.assertThat;
2
3 @SpringBootTest
4 public class BookStatisticsServiceContextTest {
5
6     @Autowired
7     private BookStatisticsService bookStatisticsService;
8
9     @Test
10    void shouldReturnTotalPages() {
11        int actual = bookStatisticsService.getTotalPagesByAuthor("Jane Austen");
12        int expected = 600;
13        assertThat(actual).isEqualTo(expected);
14    }
15 }
```

TEST DOUBLES

- The `@Mock` annotation is used to create a mock object in the test class
- `@InjectMocks` is used to automatically inject mock objects into the fields of a tested object
- The `when(...).thenReturn(...)` statement specifies behavior when a particular method is called on a mock object.

MOCKING

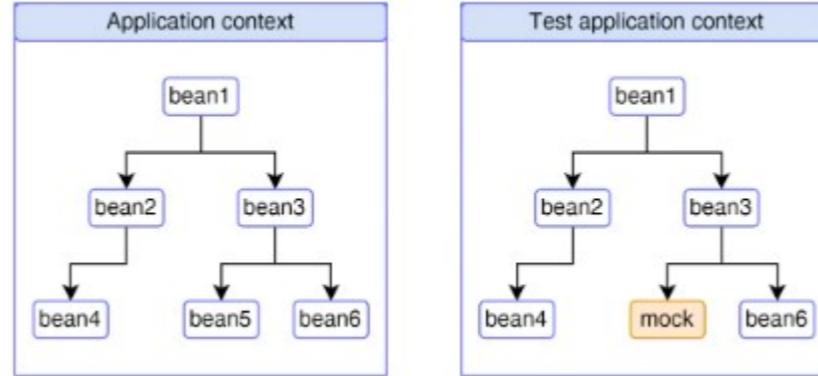
```
1 import org.junit.jupiter.api.Test;
2 import org.junit.jupiter.api.extension.ExtendWith;
3 import org.mockito.InjectMocks;
4 import org.mockito.Mock;
5 import org.mockito.Mockito;
6 import org.mockito.junit.jupiter.MockitoExtension;
7 import java.util.List;
8 import static org.assertj.core.api.Assertions.assertThat;
9
10 @ExtendWith(MockitoExtension.class)
11 public class BookStatisticsServiceMockTest {
12     @Mock
13     BookService bookService;
14     @InjectMocks
15     BookStatisticsService bookStatisticsService;
16     @Test
17     void shouldReturnTotalPages() {
18         List<Book> defaultReturnValue = bookService.getBooksByAuthor("Jane Austen");
19         /* by default the mocked method returns an empty collection*/
20         assertThat(defaultReturnValue).hasSize(0);
21         /* using when..thenReturn statement, we change the default behavior of the mock object*/
22         List<Book> janeAustenBooks = List.of(
23             new Book("Pride and Prejudice", "Jane Austen", 259),
24             new Book("Emma", "Jane Austen", 218));
25
26         Mockito.when(bookService.getBooksByAuthor("Jane Austen"))
27             .thenReturn(janeAustenBooks);
28
29         int actual = bookStatisticsService.getTotalPagesByAuthor("Jane Austen");
30         int expected = 218 + 259;
31         assertThat(actual).isEqualTo(expected);
32     }
33 }
```

MOCKING

Another way is to use Mockito from the Spring Boot package:

```
1 import org.junit.jupiter.api.Test;
2 import org.mockito.Mockito;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.test.context.SpringBootTest;
5 import org.springframework.boot.test.mock.mockito.MockBean;
6 import java.util.List;
7 import static org.assertj.core.api.Assertions.assertThat;
8
9 @SpringBootTest
10 class BookStatisticsServiceContextMockTest {
11     @MockBean
12     private BookService bookService;
13     @Autowired
14     private BookStatisticsService bookStatisticsService;
15
16     @Test
17     void shouldReturnTotalPages() {
18         List<Book> defaultReturnValue = bookService.getBooksByAuthor("Jane Austen");
19         /* by default the mocked method returns an empty collection */
20         assertThat(defaultReturnValue).hasSize(0);
21         /* using when..thenReturn statement, we change the default behavior of the mock object*/
22         List<Book> janeAustenBooks = List.of(
23             new Book("Pride and Prejudice", "Jane Austen", 259),
24             new Book("Emma", "Jane Austen", 218));
25
26         Mockito.when(bookService.getBooksByAuthor("Jane Austen"))
27             .thenReturn(janeAustenBooks);
28
29         int actual = bookStatisticsService.getTotalPagesByAuthor("Jane Austen");
30         int expected = 259 + 218;
31         assertThat(actual).isEqualTo(expected);
32     }
33 }
```

MOCKING



Test with Spring:

```
1 @SpringBootTest
2 public class ManyBeansContextTest {
3
4     @MockBean
5     private Bean5 bean5;
6
7     @Autowired
8     private Bean1 bean1;
9
10    // test method
11 }
```

MOCKING

The usual Mockito:

```
1 public class ManyBeansContextTest {  
2  
3     @Test  
4     void shouldReturnSomething() {  
5         Bean5 mock = Mockito.mock(Bean5.class);  
6         Mockito.when(mock.method()).thenReturn("MOCK");  
7         Bean6 bean6 = new Bean6();  
8         Bean3 bean3 = new Bean3(mock, bean6);  
9         Bean4 bean4 = new Bean4();  
10        Bean2 bean2 = new Bean2(bean4);  
11        Bean1 bean1 = new Bean1(bean2, bean3);  
12        ...  
13    }  
14 }
```

TESTING REST CONTROLLERS

GETTING STARTED

```
1 public class Book {  
2     private int id;  
3     private String title;  
4     private String author;  
5  
6     public Book(int id, String title, String author) {  
7         this.id = id;  
8         this.title = title;  
9         this.author = author;  
10    }  
11  
12    // getters and setters  
13 }
```

GETTING STARTED

```
1 @Service
2 public class BookService {
3     private final Map<Integer, Book> books = Map.of(
4         1, new Book(1, "The Art of Programming", "John Doe"),
5         2, new Book(2, "Mastering Spring Boot", "Jane Smith"),
6         3, new Book(3, "Understanding Algorithms", "Alice Johnson")
7     );
8
9     public Book getBookById(int bookId) {
10         return books.get(bookId);
11     }
12 }
```

GETTING STARTED

```
1 @RestController
2 public class BookController {
3     private final BookService bookService;
4
5     public BookController(BookService bookService) {
6         this.bookService = bookService;
7     }
8
9     @GetMapping(path = "/books/{id}")
10    public ResponseEntity<Book> getBook(@PathVariable int id) {
11        var payload = bookService.getBookById(id);
12
13        if (payload == null) {
14            return ResponseEntity.notFound().build();
15        } else {
16            return ResponseEntity.ok(payload);
17        }
18    }
19 }
```

TESTING REST CONTROLLERS

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.boot.test.context.SpringBootTest;
3 import org.springframework.boot.test.web.client.TestRestTemplate;
4 import org.springframework.boot.test.web.server.LocalServerPort;
5
6 @SpringBootTest(webEnvironment = SpringApplication.WebEnvironment.RANDOM_PORT)
7 class BookControllerRestTemplateTests {
8
9     @LocalServerPort
10    private int port;
11
12    @Autowired
13    private TestRestTemplate restTemplate;
14
15    // tests go here...
16 }
```

TESTING REST CONTROLLERS

```
1 import org.junit.jupiter.api.DisplayName;
2 import org.junit.jupiter.api.Test;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.test.context.SpringBootTest;
5 import org.springframework.boot.test.web.client.TestRestTemplate;
6 import org.springframework.boot.test.web.server.LocalServerPort;
7 import org.springframework.http.HttpStatus;
8 import org.springframework.http.ResponseEntity;
9 import static org.assertj.core.api.Assertions.assertThat;
10
11 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
12 class BookControllerRestTemplateTests {
13
14     @LocalServerPort
15     private int port;
16
17     @Autowired
18     private TestRestTemplate restTemplate;
19
20     @Test
21     @DisplayName("GET /books/id returns 200 OK and a Book JSON")
22     void getBook_validId_returnsValidResponseEntity() {
23         String url = "http://localhost:" + port + "/books/1";
24
25         ResponseEntity<Book> response = restTemplate.getForEntity(url, Book.class);
26
27         assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
28         assertThat(response.getBody()).isNotNull();
29         assertThat(response.getBody().getAuthor()).isEqualTo("John Doe");
30     }
31 }
```

USING @WEBMVCTEST

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
3 import org.springframework.boot.test.mock.mockito.MockBean;
4 import org.springframework.test.web.servlet.MockMvc;
5
6 @WebMvcTest(BookController.class)
7 public class BookControllerMockMvcTests {
8
9     @Autowired
10    private MockMvc mockMvc;
11
12    @MockBean
13    private BookService bookService;
14
15    // tests go here...
16 }
```

USING @WEBMVCTEST

```
1 import org.junit.jupiter.api.DisplayName;
2 import org.junit.jupiter.api.Test;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
5 import org.springframework.boot.test.mock.mockito.MockBean;
6 import org.springframework.http.MediaType;
7 import org.springframework.test.web.servlet.MockMvc;
8 import static org.mockito.Mockito.when;
9 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
10 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
11 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
12
13 @WebMvcTest(BookController.class)
14 public class BookControllerMockMvcTests {
15     /* declared fields */
16
17     @Test
18     @DisplayName("GET /books/1 returns 200 OK and a valid JSON")
19     void getBook_returnsValidResponseEntity() throws Exception {
20         var mockBook = new Book(1, "Test Book", "Test Author");
21         when(bookService.getBookById(1)).thenReturn(mockBook);
22         var requestBuilder = get("/books/1");
23
24         mockMvc.perform(requestBuilder)
25             .andExpectAll(
26                 status().isOk(),
27                 content().contentType(MediaType.APPLICATION_JSON),
28                 content().json("""
29                     {
30                         "id": 1,
31                         "title": "Test Book",
32                         "author": "Test Author"
33                     }
34                     """
35             )
36     );
37 }
38 }
```

ASSERTIONS WITH MOCKMVC

- Status

```
1 mockMvc.perform(get("/books/1"))
2     .andExpect(status().isOk())
```

- Content

```
1 mockMvc.perform(get("/books/1"))
2     .andExpect(content().json("{\"id\": \"1\", \"title\": \"Test Book\", \"author\": \"Test Author\"}"))
```

- View

```
1 mockMvc.perform(get("/books/1"))
2     .andExpect(view().name("bookDetail"))
```

ASSERTIONS WITH MOCKMVC

```
1 @Test
2 @DisplayName("GET /books/1 returns 200 OK and a valid JSON")
3 void getBook_returnsValidResponseEntity() throws Exception {
4     var mockBook = new Book(1, "Test Book", "Test Author");
5
6     when(bookService.getBookById(1)).thenReturn(mockBook);
7
8     mockMvc.perform(get("/books/1"))
9         .andExpect(status().isOk())
10        .andExpect(jsonPath("$.id").value(1))
11        .andExpect(jsonPath("$.title").value("Test Book"))
12        .andExpect(jsonPath("$.author").value("Test Author"));
13 }
```

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 class DemoTests {
4
5     @Autowired
6     private MockMvc mockMvc;
7
8     // tests go here...
9 }
```

SPRING COMPONENTS

SPRING COMPONENTS

```
1 import org.springframework.stereotype.Component;
2 import java.util.Random;
3
4 @Component
5 public class PasswordGenerator {
6     private static final String CHARACTERS = "abcdefghijklmnopqrstuvwxyz";
7     private static final Random random = new Random();
8
9     public String generate(int length) {
10         StringBuilder result = new StringBuilder();
11
12         for (int i = 0; i < length; i++) {
13             int index = random.nextInt(CHARACTERS.length());
14             result.append(CHARACTERS.charAt(index));
15         }
16
17         return result.toString();
18     }
19 }
```

INTERACTING WITH THE COMMAND LINE

```
1 @Component
2 public class Runner implements CommandLineRunner {
3
4     @Override
5     public void run(String... args) {
6         System.out.println("Hello, Spring!");
7     }
8 }
```

```
1 Hello, Spring!
```

AUTOWIRING COMPONENTS

```
1 @Component
2 public class Runner implements CommandLineRunner {
3     private final PasswordGenerator generator;
4
5     @Autowired
6     public Runner(PasswordGenerator generator) {
7         this.generator = generator;
8     }
9
10    @Override
11    public void run(String... args) {
12        System.out.println("A short password: " + generator.generate(5));
13        System.out.println("A long password: " + generator.generate(10));
14    }
15 }
```

```
1 A short password: bqtik
2 A long password: tjdpswzbd
```

WHERE TO PUT THE @AUTOWIRED ANNOTATION

- before a constructor argument:

```
1  @Component
2  public class Runner implements CommandLineRunner {
3      private final PasswordGenerator generator;
4
5      @Autowired
6      public Runner(PasswordGenerator generator) {
7          this.generator = generator;
8      }
9
10     // run
11 }
```

WHERE TO PUT THE @AUTOWIRED ANNOTATION

- on top of a constructor:

```
1 @Component
2 public class Runner implements CommandLineRunner {
3     private final PasswordGenerator generator;
4
5     public Runner(@Autowired PasswordGenerator generator) {
6         this.generator = generator;
7     }
8
9     // run
10 }
```

WHERE TO PUT THE @AUTOWIRED ANNOTATION

- directly on a field to be injected:

```
1  @Component
2  public class Runner implements CommandLineRunner {
3
4      @Autowired
5      private PasswordGenerator generator;
6
7      // run
8  }
```

WHERE TO PUT THE @AUTOWIRED ANNOTATION

omit the annotation over the constructor:

```
1  @Component
2  public class Runner implements CommandLineRunner {
3      private final PasswordGenerator generator;
4
5      public Runner(PasswordGenerator generator) {
6          this.generator = generator;
7      }
8
9      // run
10 }
```

SPRING BEANS

INITIAL PREPARATIONS

```
1 @SpringBootApplication
2 public class DemoSpringApplication {
3
4     public static void main(String[] args) {
5         SpringApplication.run(DemoSpringApplication.class, args);
6     }
7
8 }
```

DECLARING BEANS

```
1 @Configuration
2 public class Addresses {
3
4     @Bean
5     public String address() {
6         return "Green Street, 102";
7     }
8
9 }
```

AUTOWIRING BEANS

```
1 class Customer {  
2     private final String name;  
3     private final String address;  
4  
5     Customer(String name, String address) {  
6         this.name = name;  
7         this.address = address;  
8     }  
9  
10    /* getters */  
11  
12    @Override  
13    public String toString() {  
14        return "Customer{" +  
15            "name='" + name + '\'' +  
16            ", address='" + address + '\'' +  
17            '}';  
18    }  
19 }
```

AUTOWIRING BEANS

```
1 @SpringBootApplication
2 public class DemoSpringApplication {
3
4     public static void main(String[] args) {
5         SpringApplication.run(DemoSpringApplication.class, args);
6     }
7
8     @Bean
9     public Customer customer(@Autowired String address) {
10         return new Customer("Clara Foster", address);
11     }
12 }
```

AUTOWIRING BEANS

```
1 @Bean
2 public Customer temporary(@Autowired Customer customer) {
3     System.out.println(customer);
4     return customer;
5 }
```

```
1 Customer{name='Clara Foster', address='Green Street, 102'}
```

DISTINGUISHING BEANS OF THE SAME TYPE

```
1 @Bean
2 public String address1() {
3     return "Green Street, 102";
4 }
5 @Bean
6 public String address2() {
7     return "Apple Street, 15";
8 }
9 @Bean
10 public Customer customer(@Qualifier("address2") String address) {
11     return new Customer("Clara Foster", address);
12 }
13 @Bean
14 public Customer temporary(@Autowired Customer customer) {
15     /* Customer{name='Clara Foster', address='Apple Street, 15'} */
16     System.out.println(customer);
17     return customer;
18 }
```

```
1 Parameter 0 of method customer in eu.mithril.beans.DemoSpringApplication
2 required a single bean, but 2 were found:
3   - address1: defined by method 'address1' in eu.mithril.beans.DemoSpringApplication
4   - address2: defined by method 'address2' in eu.mithril.beans.DemoSpringApplication
```

BEANS VS STANDARD OBJECTS

```
1 String address = "Green Street, 102";
2 Customer customer = new Customer("Clara Foster", address);
```

@BEAN VS @COMPONENT

USING @BEAN AND @COMPONENT IN CODE TOGETHER

```
1  @Configuration
2  public class PasswordConfig {
3      private static final String ALPHA = "abcdefghijklmnopqrstuvwxyz";
4      private static final String NUMERIC = "0123456789";
5      private static final String SPECIAL_CHARS = "!@#$%^&*_+=-/ ";
6
7      @Bean
8      public PasswordAlphabet allCharacters() {
9          return new PasswordAlphabet(ALPHA + NUMERIC + SPECIAL_CHARS);
10     }
11
12     static class PasswordAlphabet {
13         private final String characters;
14
15         public PasswordAlphabet(String characters) {
16             this.characters = characters;
17         }
18
19         public String getCharacters() {
20             return characters;
21         }
22     }
23 }
```

USING @BEAN AND @COMPONENT IN CODE TOGETHER

```
1  @Component
2  public class PasswordGenerator {
3      private static final Random random = new Random();
4      private final PasswordAlphabet alphabet;
5
6      public PasswordGenerator(@Autowired PasswordAlphabet alphabet) {
7          this.alphabet = alphabet;
8      }
9
10     public String generate(int length) {
11         String allCharacters = alphabet.getCharacters(); /* get the characters from the bean */
12         StringBuilder result = new StringBuilder();
13
14         for (int i = 0; i < length; i++) {
15             int index = random.nextInt(allCharacters.length());
16             result.append(allCharacters.charAt(index));
17         }
18
19         return result.toString();
20     }
21 }
```

USING @BEAN AND @COMPONENT IN CODE TOGETHER

```
1 @Component
2 public class Runner implements CommandLineRunner {
3     private final PasswordGenerator generator;
4
5     public Runner(PasswordGenerator generator) {
6         this.generator = generator;
7     }
8
9     @Override
10    public void run(String... args) {
11        System.out.println("A short password: " + generator.generate(5));
12        System.out.println("A long password: " + generator.generate(10));
13    }
14 }
```

```
1 A short password: e&7sd
2 A long password: up_&g4xtj7
```

@COMPONENT VS @BEAN

- The `@Bean` annotation is a method-level annotation, whereas `@Component` is a class-level annotation.
- The `@Component` annotation doesn't need to be used with the `@Configuration` annotation, whereas the `@Beangeneric` annotation has to be used within a class annotated with `@Configuration`.
- If you want to create a single bean for a class from an external library, you cannot just add the `@Componentannotation` because you cannot edit the class. However, you can declare a method annotated with `@Bean` and return an object of this class from this method.
- There are several specializations of the `@Component` annotation, whereas `@Bean` doesn't have any specialized stereotype annotations.

SPECIALIZATIONS OF COMPONENTS IN SPRING

- `@Component` indicates a generic Spring component.
- `@Service` indicates a business logic component but doesn't provide any additional functions.
- `@Controller` / `@RestController` indicates a component that can work in REST web services.
- `@Repository` indicates a component that interacts with an external data storage (e.g., a database).

SPRING STEREOTYPES

SPRING STEREOTYPE ANNOTATIONS

- `@Component` : Use this annotation to define a class as a Spring component.
- `@Repository` : This annotation identifies a class as a data repository.
- `@Service` : You use this annotation to mark a class as a service.
- `@Controller` : This annotation signifies a class as an HTTP controller.

@COMPONENT ANNOTATION

```
1 import org.springframework.stereotype.Component  
2  
3 @Component  
4 public class Dragon {  
5  
6     public String slayDragon() {  
7         return "Rooooaaar!";  
8     }  
9 }
```

@COMPONENT ANNOTATION

```
1 import org.springframework.beans.factory.annotation.Autowired
2 import org.springframework.stereotype.Component
3
4 @Component
5 public class Knight {
6
7     private final Dragon dragon;
8
9     @Autowired
10    public Knight(Dragon dragon) {
11        this.dragon = dragon;
12    }
13
14    public void slayDragon() {
15        System.out.println(dragon.slayDragon());
16        System.out.println("The knight slays the dragon!");
17    }
18 }
```

@COMPONENT ANNOTATION

```
1 import org.springframework.beans.factory.annotation.Autowired  
2 import org.springframework.stereotype.Controller  
3  
4 @Controller  
5 public class RoarController {  
6  
7     private final Knight knight;  
8  
9     @Autowired  
10    public RoarController(Knight knight) {  
11        this.knight = knight;  
12    }  
13  
14    // ... some methods  
15 }
```

```
1 Rooooaaar!  
2 The knight slays the dragon!
```

@CONTROLLER ANNOTATION

```
1 import org.springframework.stereotype.Controller
2 import org.springframework.web.bind.annotation.GetMapping
3 import org.springframework.web.bind.annotation.ResponseBody
4
5 @Controller
6 public class MyController {
7
8     @GetMapping("/hello")
9     @ResponseBody
10    public String helloWorld() {
11        return "Hello, World!";
12    }
13
14    // Other methods...
15 }
```

@SERVICE ANNOTATION

```
1 import org.springframework.stereotype.Service  
2  
3 @Service  
4 public class MyService {  
5  
6     private String generatePasswordHash(String userPassword) {  
7         /* Implementing Password Hash Generation  
8             ... */  
9         return hashedPassword  
10    }  
11 }
```

@REPOSITORY ANNOTATION

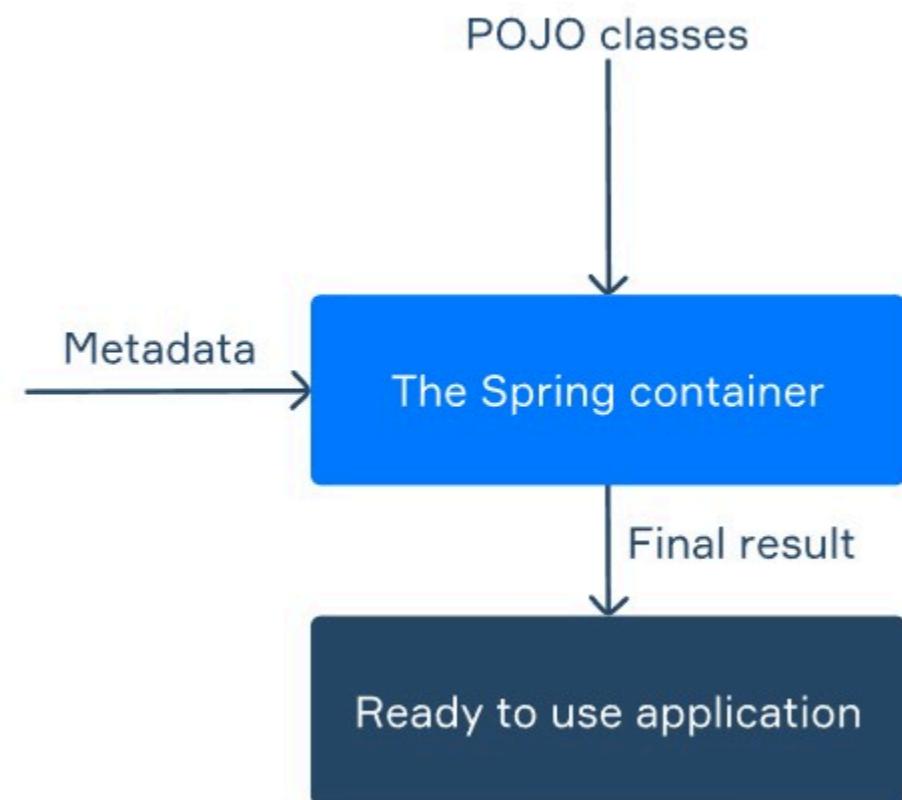
```
1 import org.springframework.stereotype.Repository  
2  
3 @Repository  
4 public class MyRepository {  
5  
6     public void save(MyEntity entity) {  
7         /* Implementing Persistence  
8             ... */  
9     }  
10  
11    public MyEntity findById(Long id) {  
12        /* Implementing ID Search  
13            ... */  
14    }  
15 }
```

BENEFITS OF USING STEREOTYPE ANNOTATIONS

- Simplified configuration
- Modularity and reusability
- Improved structure

IOC CONTAINER

SPRING CONTAINER



POJO

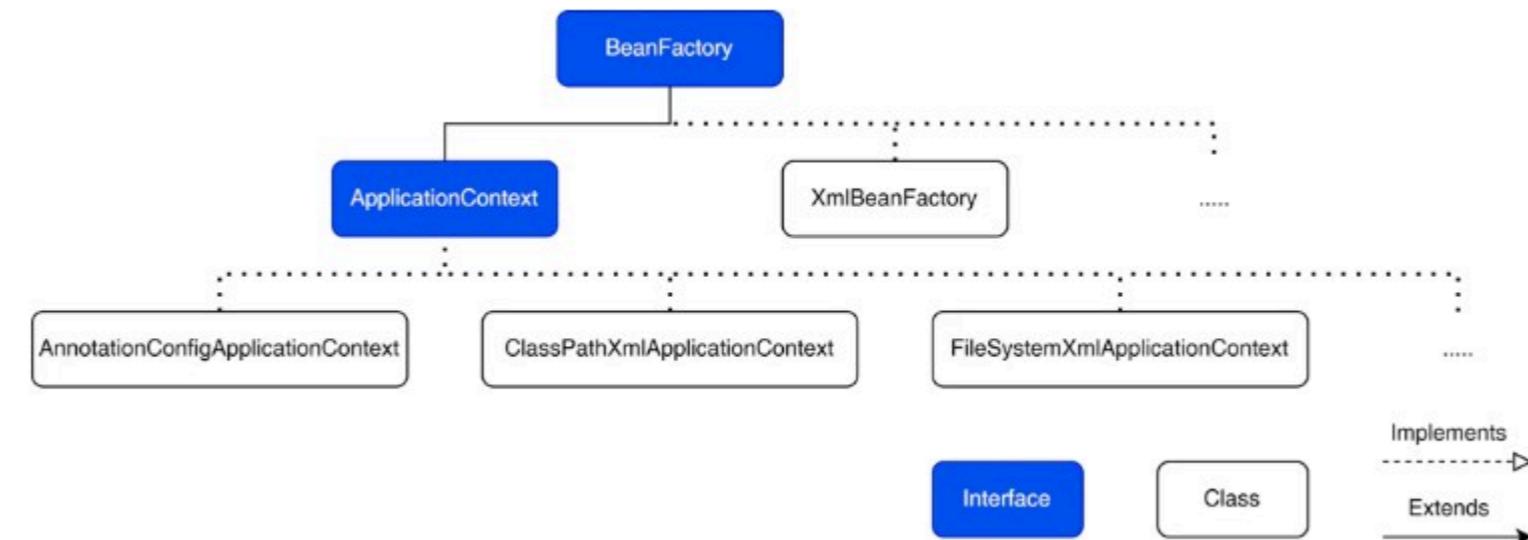
- A POJO is a simple object that doesn't depend on a framework.
- A Java Bean is a POJO with some additional requirements and restrictions.
- A Spring Bean is a POJO or JavaBean created and managed by an instance of the Spring IoC container.

CONTEXTS AND BEAN FACTORY

ApplicationContext implementations:

- FileSystemXmlApplicationContext
- ClassPathXmlApplicationContext
- WebApplicationContext == ApplicationContext

BEANFACTORY & APPLICATIONCONTEXT



CREATING AN APPLICATION CONTEXT

```
1 public class Person {  
2  
3     private String name;  
4  
5     public Person(String name) {  
6         this.name = name;  
7     }  
8 }
```

```
1 @Configuration  
2 public class Config {  
3  
4     @Bean  
5     public Person personMary() {  
6         return new Person("Mary");  
7     }  
8 }
```

CREATING AN APPLICATION CONTEXT

```
1 public class Application {  
2  
3     public static void main(String[] args) {  
4         var context = new AnnotationConfigApplicationContext(Config.class);  
5         System.out.println(Arrays.toString(context.getBeanDefinitionNames()));  
6     }  
7 }
```

```
1 [..., ..., config, personMary]
```

```
1 context.getBean(Person.class); /* returns a Person object */  
2 context.getBean("personMary"); /* returns an Object object */  
3 context.getBean("personMary", Person.class); /* returns a Person object */
```

@COMPONENTSCAN

```
1 @Component
2 public class Book {
3
4 }
5
6 @Component
7 public class Movie {
8
9 }
```

```
1 @ComponentScan
2 @Configuration
3 public class Config {
4     // bean definitions
5 }
```

@COMPONENTSCAN

```
1 public class Application {  
2  
3     public static void main(String[] args) {  
4         var context = new AnnotationConfigApplicationContext(Config.class);  
5  
6         /* [..., ..., book, config, movie, personMary, ..., ...] */  
7         System.out.println(Arrays.toString(context.getBeanDefinitionNames()));  
8     }  
9 }
```

```
1 @ComponentScan(basePackages = "packageName")  
2  
3 @ComponentScan(value = "packageName")  
4  
5 @ComponentScan("packageName")
```

APPLICATIONCONTEXT IN SPRING BOOT

```
1 @SpringBootApplication
2 public class Application {
3
4     public static void main(String[] args) {
5         SpringApplication.run(Application.class, args);
6     }
7 }
```

```
1 @SpringBootApplication
2 public class Application {
3
4     public static void main(String[] args) {
5         ConfigurableApplicationContext context = SpringApplication.run(Application.class, a
6         System.out.println(Arrays.toString(context.getBeanDefinitionNames())));
7     }
8 }
```

APPLICATIONCONTEXT IN SPRING BOOT

```
1 @Component
2 public class Runner implements CommandLineRunner {
3
4     @Autowired
5     ApplicationContext applicationContext;
6
7     @Override
8     public void run(String... args) throws Exception {
9         System.out.println(Arrays.toString(applicationContext.getBeanDefinitionNames()))
10    }
11 }
```

SCOPES OF BEANS

A TEMPLATE FOR FUTURE EXAMPLES

```
1 @Configuration
2 public class AppConfig {
3
4     @Bean
5     public AtomicInteger createCounter() {
6         return new AtomicInteger();
7     }
8 }
```

```
1 @Component
2 public class AppRunner implements CommandLineRunner {
3     private final AtomicInteger counter1;
4     private final AtomicInteger counter2;
5
6     public AppRunner(AtomicInteger counter1, AtomicInteger counter2) {
7         this.counter1 = counter1;
8         this.counter2 = counter2;
9     }
10
11    @Override
12    public void run(String... args) {
13        counter1.addAndGet(2);
14        counter2.addAndGet(3);
15        counter1.addAndGet(5);
16        System.out.println(counter1.get());
17        System.out.println(counter2.get());
18    }
19 }
```

SINGLETON SCOPE

```
1 @Bean  
2 @Scope("singleton")  
3 public AtomicInteger createCounter() { /* ... */ }
```

```
1 @Bean  
2 @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
```

```
1 10  
2 10
```

PROTOTYPE SCOPE

```
1 @Bean  
2 @Scope("prototype")  
3 public AtomicInteger createCounter() { /* ... */ }
```

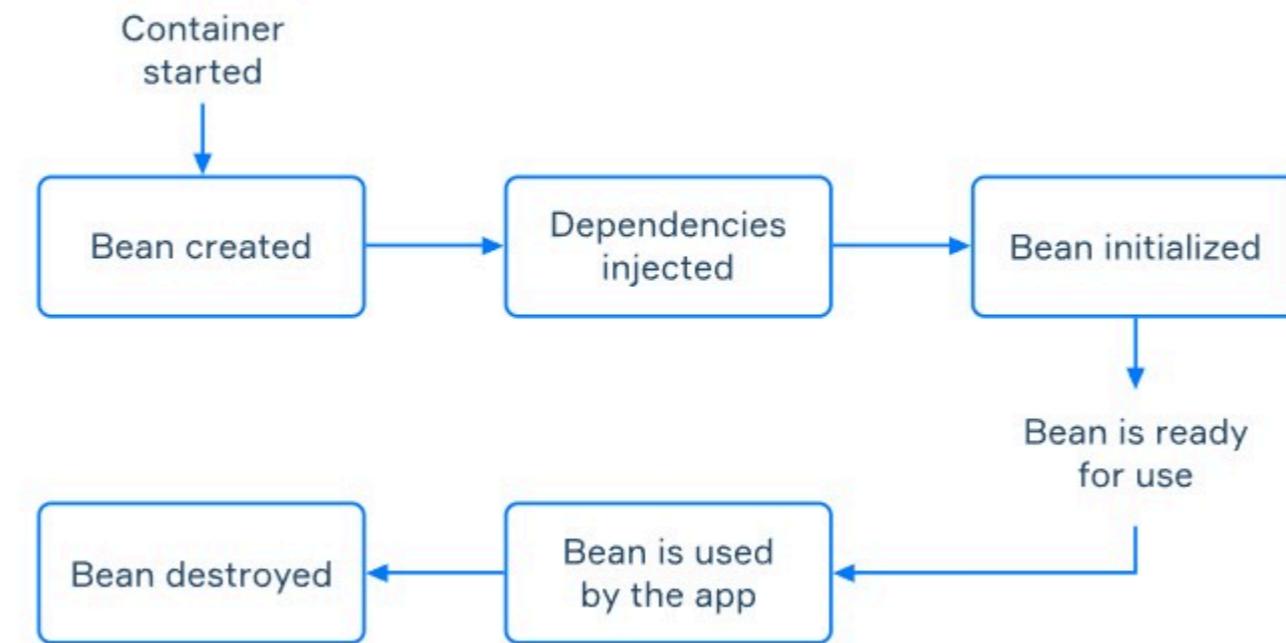
```
1 @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

```
1 7  
2 3
```

OTHER BEAN SCOPES

- Request
- Session
- Application
- WebSocket

BEAN LIFECYCLE



CUSTOMIZING BEAN INITIALIZATION AND DESTRUCTION

- Use special annotations (`@PostConstruct` , `@PreDestroy` , `@Bean`).
- Implement some interfaces (`InitializingBean` , `DisposableBean`).
- Use an XML bean definition file.

USING ANNOTATIONS FOR CUSTOMIZATION

```
1 @Component
2 class TechLibrary {
3     private final List<String> bookTitles =
4         Collections.synchronizedList(new ArrayList<>());
5
6     @PostConstruct
7     public void init() {
8         bookTitles.add("Clean Code");
9         bookTitles.add("The Art of Computer Programming");
10        bookTitles.add("Introduction to Algorithms");
11        System.out.println("The library has been initialized: " + bookTitles);
12    }
13
14    @PreDestroy
15    public void destroy() {
16        bookTitles.clear();
17        System.out.println("The library has been cleaned: " + bookTitles);
18    }
19 }
```

```
1 The library has been initialized: [Clean Code, The Art of Computer Programming, Introduction to Algorithms]
2 2022-04-22 12:08:06.515  INFO Started HsSpringApplication in 0.382 seconds (JVM running for 5.698)
3 The library has been cleaned: []
4
5 Process finished with exit code 0
```

USING ANNOTATIONS FOR CUSTOMIZATION

```
1  @Configuration
2  class Config {
3
4      @Bean(initMethod = "init", destroyMethod = "destroy")
5      public TechLibrary library() {
6          return new TechLibrary();
7      }
8  }
9
10 class TechLibrary {
11     private final List<String> bookTitles =
12         Collections.synchronizedList(new ArrayList<>());
13
14     public void init() {
15         bookTitles.add("Clean Code");
16         bookTitles.add("The Art of Computer Programming");
17         bookTitles.add("Introduction to Algorithms");
18         System.out.println("The library has been initialized: " + bookTitles);
19     }
20
21     public void destroy() {
22         bookTitles.clear();
23         System.out.println("The library has been cleaned: " + bookTitles);
24     }
25 }
```

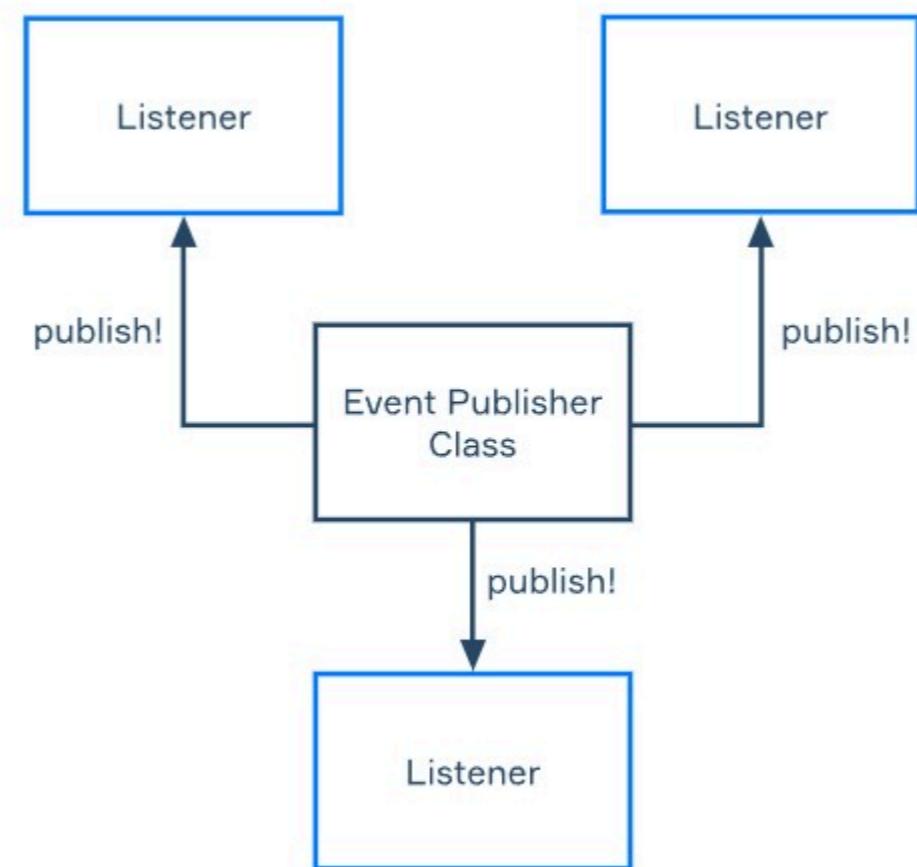
USING INTERFACES FOR CUSTOMIZATION

```
1 @Component
2 class TechLibrary implements InitializingBean, DisposableBean {
3     private final List<String> bookTitles =
4         Collections.synchronizedList(new ArrayList<>());
5
6     @Override
7     public void afterPropertiesSet() throws Exception {
8         bookTitles.add("Clean Code");
9         bookTitles.add("The Art of Computer Programming");
10        bookTitles.add("Introduction to Algorithms");
11        System.out.println("The library has been initialized: " + bookTitles);
12    }
13
14    @Override
15    public void destroy() {
16        bookTitles.clear();
17        System.out.println("The library has been cleaned: " + bookTitles);
18    }
19 }
```

POST-PROCESSORS FOR BEANS

```
1 @Component
2 class PostProcessor implements BeanPostProcessor {
3
4     @Override
5     public Object postProcessBeforeInitialization(
6         Object bean, String beanName) throws BeansException {
7
8         System.out.println("Before initialization: " + beanName);
9
10    return BeanPostProcessor.super
11        .postProcessBeforeInitialization(bean, beanName);
12 }
13
14 @Override
15 public Object postProcessAfterInitialization(
16     Object bean, String beanName) throws BeansException {
17
18     System.out.println("After initialization: " + beanName);
19
20    return BeanPostProcessor.super
21        .postProcessAfterInitialization(bean, beanName);
22 }
23 }
```

EVENTS AND LISTENERS



EVENTS AND LISTENERS IMPLEMENTATION

```
1 public class HelloEvent extends ApplicationEvent {  
2  
3     private final String message;  
4  
5     public HelloEvent(Object source, String message) {  
6         super(source);  
7         this.message = message;  
8     }  
9  
10    public String getMessage() {  
11        return message;  
12    }  
13 }
```

```
1 @Component  
2 public class HelloListenerOne implements ApplicationListener<HelloEvent> {  
3  
4     @Override  
5     public void onApplicationEvent(HelloEvent event) {  
6         System.out.println("First listener handled the event, the message is " + event.getMessage());  
7     }  
8 }  
9  
10 @Component  
11 public class HelloListenerTwo implements ApplicationListener<HelloEvent> {  
12  
13     @Override  
14     public void onApplicationEvent(HelloEvent event) {  
15         System.out.println("Second listener handled the event, the message is " + event.getMessage());  
16     }  
17 }
```

EVENTS AND LISTENERS IMPLEMENTATION

```
1 @SpringBootApplication
2 public class EventsAndListenersProjectApplication {
3
4     public static void main(String[] args) {
5         SpringApplication.run(EventsAndListenersProjectApplication.class, args);
6     }
7 }
8
9 @Component
10 class Runner implements CommandLineRunner {
11
12     private final ApplicationEventPublisher eventPublisher;
13
14     public Runner(ApplicationEventPublisher eventPublisher) {
15         this.eventPublisher = eventPublisher;
16     }
17
18     @Override
19     public void run(String... args) {
20         HelloEvent event = new HelloEvent(this, "Hello World!");
21         eventPublisher.publishEvent(event);
22     }
23 }
```

```
1 First listener handled the event, the message is Hello World!
2 Second listener handled the event, the message is Hello World!
```

SYNCHRONOUS APPROACH

```
1 @Component
2 public class HelloListenerOne implements ApplicationListener<HelloEvent> {
3
4     @Override
5     public void onApplicationEvent(HelloEvent event) {
6         System.out.println("First listener began to handle the event");
7         try {
8             Thread.sleep(500);
9         } catch (InterruptedException e) {
10             throw new RuntimeException(e);
11         }
12         System.out.println("First listener ended to handle the event");
13     }
14 }
15
16 @Component
17 public class HelloListenerTwo implements ApplicationListener<HelloEvent> {
18
19     @Override
20     public void onApplicationEvent(HelloEvent event) {
21         System.out.println("Second listener began to handle the event");
22         try {
23             Thread.sleep(500);
24         } catch (InterruptedException e) {
25             throw new RuntimeException(e);
26         }
27         System.out.println("Second listener ended to handle the event");
28     }
29 }
```

SYNCHRONOUS APPROACH

```
1 @Override
2 public void run(String... args) throws InterruptedException {
3     System.out.println("The application was started");
4     HelloEvent event = new HelloEvent(this, "Hello World!");
5     eventPublisher.publishEvent(event);
6     Thread.sleep(500);
7     eventPublisher.publishEvent(event);
8     System.out.println("The application was ended");
9 }
```

```
1 The application was started
2 First listener began to handle the event
3 First listener ended to handle the event
4 Second listener began to handle the event
5 Second listener ended to handle the event
6 First listener began to handle the event
7 First listener ended to handle the event
8 Second listener began to handle the event
9 Second listener ended to handle the event
10 The application was ended
```

ANNOTATION-DRIVEN LISTENERS

```
1 public class PojoEvent {  
2 }
```

```
1 @Component  
2 public class AnnotationListener {  
3     @EventListener  
4     public void handleEvent(PojoEvent event) {  
5         System.out.println("POJO event is being handled...");  
6     }  
7 }
```

```
1 @Override  
2 public void run(String... args) {  
3     eventPublisher.publishEvent(new PojoEvent());  
4 }
```

DEPENDENCY INJECTION IN SPRING

TYPE MATCHING

```
1 public class Car {  
2  
3     private String name;  
4     private String model;  
5  
6     public Car(String name, String model) {  
7         this.name = name;  
8         this.model = model;  
9     }  
10    //omitted getters and setters  
11 }
```

```
1 @Configuration  
2 public class Config {  
3  
4     @Bean  
5     @Primary  
6     public Car teslaCar() {  
7         return new Car("Tesla", "2023");  
8     }  
9     @Bean  
10    public Car toyotaCar() {  
11        return new Car("Toyota", "2023");  
12    }  
13 }
```

```
1 public class DiApplication {  
2     public static void main(String[] args) {  
3         /* Create the Spring application context */  
4         var context = new AnnotationConfigApplicationContext(Config.class);  
5         /* Retrieve an instance of MyBean */  
6         Car myBean = context.getBean(Car.class);  
7         System.out.println(myBean.getName());  
8     }  
9 }
```

TYPE MATCHING

Another way:

```
1 @Configuration
2 public class Config {
3
4     @Bean("tesla")
5     public Car teslaCar() {
6         return new Car("Tesla", "2023");
7     }
8
9     @Bean("toyota")
10    public Car toyotaCar() {
11        return new Car("Toyota", "2023");
12    }
13 }
```

```
1 public class DiApplication {
2     public static void main(String[] args) {
3         /* Create the Spring application context */
4         var context = new AnnotationConfigApplicationContext(Config.class);
5         /* Retrieve an instance of MyBean */
6         Car myBean = context.getBean("tesla",Car.class);
7         System.out.println(myBean.getName());
8     }
9 }
```

@QUALIFIER

```
1 public class Engine {  
2  
3     private String brand;  
4     private boolean isRunning;  
5  
6     public Engine(String brand, boolean isRunning) {  
7         this.brand = brand;  
8         this.isRunning = isRunning;  
9     }  
10  
11    //omitted getters and setters  
12  
13 }
```

```
1 @Configuration  
2 public class Config {  
3  
4     @Bean("tesla")  
5     public Car teslaCar() {  
6         return new Car("Tesla", "2023");  
7     }  
8     @Bean("toyota")  
9     public Car toyotaCar() {  
10        return new Car("Toyota", "2023");  
11    }  
12  
13     @Bean  
14     public Engine teslaEngine(){  
15         return new Engine("Tesla", true);  
16     }  
17  
18     @Bean  
19     public Engine toyotaEngine(){  
20         return new Engine("Toyota", true);  
21     }  
22  
23 }
```

@QUALIFIER

```
1 public class Car {  
2  
3     private String name;  
4     private String model;  
5  
6     @Qualifier("teslaEngine")  
7     @Autowired  
8     private Engine engine;  
9  
10    public Car(String name, String model) {  
11        this.name = name;  
12        this.model = model;  
13    }  
14  
15    //omitted getters and setters  
16 }
```

```
1 @SpringBootApplication  
2 public class DlApplication {  
3  
4     public static void main(String[] args) {  
5  
6         var context = new AnnotationConfigApplicationContext(Config.class);  
7         var bean = context.getBean("tesla", Car.class);  
8         System.out.println(bean.getEngine().getBrand());  
9     }  
10 }
```

LOOSE COUPLING

```
1 public interface Engine {  
2     void start();  
3 }
```

```
1 public class DieselEngine implements Engine {  
2     @Override  
3     public void start() {  
4         System.out.println("Goes r-r-r-r... and exhausts black smoke");  
5     }  
6 }
```

```
1 public class ElectricEngine implements Engine {  
2     @Override  
3     public void start() {  
4         System.out.println("Goes b-z-z-z-z... and produces sparks");  
5     }  
6 }
```

```
1 public class Vehicle {  
2     private Engine engine;  
3  
4     public Vehicle(Engine engine) {  
5         this.engine = engine;  
6     }  
7  
8     public void drive() {  
9         engine.start();  
10    }  
11 }
```

LOOSE COUPLING

```
1 @Configuration
2 public class Config {
3     @Bean
4     public Engine dieselEngine() {
5         return new DieselEngine();
6     }
7
8     @Bean
9     public Engine electricEngine() {
10        return new ElectricEngine();
11    }
12
13    @Bean
14    public Vehicle vehicle(@Qualifier("electricEngine") Engine engine) {
15        return new Vehicle(engine);
16    }
17 }
```

```
1 @SpringBootApplication
2 public class DiApplication {
3
4     public static void main(String[] args) {
5
6         ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
7         Vehicle vehicle = context.getBean(Vehicle.class);
8         vehicle.drive();
9     }
10 }
```

```
1 // Goes b-z-z-z-z... and produces sparks
```

EXTERNAL RESOURCES

@PROPERTYSOURCE

landmark.properties :

```
1 egypt=pyramids
2 usa=Statue of Liberty
3 france=Eiffel Tower
```

```
1 @SpringBootApplication
2 @PropertySource("classpath:landmark.properties")
3 public class DemoApplication implements CommandLineRunner {
4     @Autowired
5     private Environment environment;
6
7     public static void main(String[] args) {
8         SpringApplication.run(DemoApplication.class, args);
9     }
10
11    @Override
12    public void run(String... args) {
13        System.out.println("The most famous landmark in Egypt is " +
14            environment.getProperty("egypt"));
15    }
16 }
```

```
1 The most famous landmark in Egypt is pyramids
```

@VALUE

```
1 @SpringBootApplication
2 @PropertySource("classpath:landmark.properties")
3 public class DemoApplication implements CommandLineRunner {
4     @Autowired
5     private Environment environment;
6
7     @Value("france")
8     private String landmark;
9
10    public static void main(String[] args) {
11        SpringApplication.run(DemoApplication.class, args);
12    }
13
14    @Override
15    public void run(String... args) {
16        System.out.println(landmark);
17    }
18 }
```

```
1 france
```

To see the "Eiffel Tower" value we need to fix:

```
1     @Value("${france}")
2     private String landmark;
```

ASYNCHRONOUS PROCESSING

SCHEDULING

CREATING A BASIC SCHEDULE

```
1 import org.springframework.boot.SpringApplication;
2 import org.springframework.boot.autoconfigure.SpringBootApplication;
3 import org.springframework.scheduling.annotation.*;
4
5 @SpringBootApplication
6 @EnableScheduling
7 public class Demo2Application {
8     public static void main(String[] args) {
9         SpringApplication.run(Demo2Application.class, args);
10    }
11 }
```

```
1 import org.springframework.stereotype.Component;
2 import org.springframework.scheduling.annotation.*;
3
4 @Component
5 public class GreetScheduler{
6
7     @Scheduled(fixedRate = 2000)
8     public void greetUser() {
9         System.out.println("Hello!");
10    }
11 }
```

CREATING A BASIC SCHEDULE

```
1 @Component
2 public class ImportantTaskScheduler {
3
4     @Scheduled(fixedDelay = 2000)
5     public void runImportantTask() {
6         System.out.println("Task started!");
7     }
8 }
```

```
1 @Component
2 public class MessageScheduler {
3
4     @Scheduled(initialDelay = 2000, fixedDelay = 1000)
5     public void checkForMessage() {
6         System.out.println("New message received!");
7     }
8 }
```

ASYNCHRONOUS SCHEDULING

```
1 @Component
2 @EnableAsync /* enables async methods */
3 public class AsyncTaskScheduler {
4
5     @Async
6     @Scheduled(fixedRate = 1000)
7     public void asyncTask() throws InterruptedException {
8         System.out.println("Task started!");
9         Thread.sleep(2000);
10        System.out.println("Task completed!");
11    }
12 }
```

```
1 Task started!
2 Task started!
3 Task completed!
```

CRON JOBS AND MACROS

```
1 @Component
2 public class UpdateScheduler {
3
4     @Scheduled(cron="0 0 19 * * ?")
5     public void dailyUpdate() {
6         System.out.println("Downloading new updates!");
7     }
8 }
```

```
1 @Component
2 public class BackupScheduler {
3
4     @Scheduled(cron="@daily")
5     public void performDailyBackup() {
6         System.out.println("Backing up data!");
7     }
8 }
```

ASYNC METHODS

CREATING ASYNC METHODS

```
1 import org.springframework.context.annotation.Bean;
2 import org.springframework.context.annotation.Configuration;
3
4 @Configuration
5 public class PasswordConfig {
6     private static final String ALPHA = "abcdefghijklmnopqrstuvwxyz";
7     private static final String NUMERIC = "0123456789";
8     private static final String SPECIAL_CHARS = "!@#$%^&*_=-/ ";
9
10    @Bean
11    public PasswordAlphabet allCharacters() {
12        return new PasswordAlphabet(ALPHA + NUMERIC + SPECIAL_CHARS);
13    }
14
15    static class PasswordAlphabet {
16        private final String characters;
17
18        public PasswordAlphabet(String characters) {
19            this.characters = characters;
20        }
21
22        public String getCharacters() {
23            return characters;
24        }
25    }
26 }
```

CREATING ASYNC METHODS

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Component;
3
4 import java.util.Random;
5
6 @Component
7 public class PasswordGenerator {
8     private static final Random random = new Random();
9     private final PasswordAlphabet alphabet;
10
11    public PasswordGenerator(@Autowired PasswordAlphabet alphabet) {
12        this.alphabet = alphabet;
13    }
14
15    public String generate(int length) {
16        String allCharacters = alphabet.getCharacters(); /* take the characters from the bean */
17        StringBuilder result = new StringBuilder();
18        for (int i = 0; i < length; i++) {
19            int index = random.nextInt(allCharacters.length());
20            result.append(allCharacters.charAt(index));
21        }
22        return result.toString();
23    }
24 }
```

CREATING ASYNC METHODS

```
1 import org.springframework.boot.CommandLineRunner;
2 import org.springframework.stereotype.Component;
3
4 @Component
5 public class Runner implements CommandLineRunner {
6     private final AsyncPasswordGenerator passwordGenerator;
7
8     public Runner(AsyncPasswordGenerator passwordGenerator) {
9         this.passwordGenerator = passwordGenerator;
10    }
11
12    @Override
13    public void run(String... args) throws InterruptedException {
14        this.passwordGenerator.generateLong();
15        this.passwordGenerator.generateShort();
16    }
17 }
```

```
1 import org.springframework.stereotype.Component;
2
3 @Component
4 public class AsyncPasswordGenerator {
5     private PasswordGenerator passwordGenerator;
6
7     public AsyncPasswordGenerator(PasswordGenerator passwordGenerator) {
8         this.passwordGenerator = passwordGenerator;
9     }
10 }
```

CREATING ASYNC METHODS

```
1 import org.springframework.scheduling.annotation.EnableAsync;  
2  
3 @Component  
4 @EnableAsync  
5 public class AsyncPasswordGenerator {  
6  
7 }
```

```
1 import org.springframework.scheduling.annotation.Async;  
2  
3 @Component  
4 @EnableAsync  
5 public class AsyncPasswordGenerator {  
6     private PasswordGenerator passwordGenerator;  
7  
8     public AsyncPasswordGenerator(PasswordGenerator passwordGenerator) {  
9         this.passwordGenerator = passwordGenerator;  
10    }  
11  
12    @Async  
13    public void generateLong() throws InterruptedException {  
14        System.out.println("A long password: " + passwordGenerator.generate(10));  
15    }  
16  
17    @Async  
18    public void generateShort() throws InterruptedException {  
19        System.out.println("A short password: " + passwordGenerator.generate(5));  
20    }  
21 }
```

CREATING ASYNC METHODS

```
1 @Override  
2 public void run(String... args) throws InterruptedException {  
3     this.passwordGenerator.generateLong();  
4     this.passwordGenerator.generateShort();  
5 }
```

```
1 @Async  
2 public void generateLong() throws InterruptedException {  
3     System.out.println("A long password: " + passwordGenerator.generate(10));  
4     System.out.println(Thread.currentThread());  
5 }  
6  
7 @Async  
8 public void generateShort() throws InterruptedException {  
9     System.out.println("A short password: " + passwordGenerator.generate(5));  
10    System.out.println(Thread.currentThread());  
11 }
```

```
1 A long password: 4te_ivxdbr  
2 A short password: =ruzz  
3 Thread[task-1,5,main]  
4 Thread[task-2,5,main]
```

ASYNC RETURNS

```
1 @Async
2 public CompletableFuture<String> generateLong() throws InterruptedException {
3     return CompletableFuture.completedFuture("A long password: " + generator.generate(10));
4 }
5
6 @Async
7 public CompletableFuture<String> generateShort() throws InterruptedException {
8     return CompletableFuture.completedFuture("A short password: " + generator.generate(5));
9 }
```

```
1 @Override
2 public void run(String... args) throws InterruptedException {
3     CompletableFuture<String> longPassFuture = this.passwordGenerator.generateLong();
4     CompletableFuture<String> shortPassFuture = this.passwordGenerator.generateShort();
5
6     try {
7         String longPass = longPassFuture.get();
8         String shortPass = shortPassFuture.get();
9
10        System.out.println(longPass);
11        System.out.println(shortPass);
12
13    } catch (InterruptedException | ExecutionException ex) {
14        System.out.println(ex);
15    }
16 }
```

ASYNCHRONOUS EXECUTORS

```
1 @Bean(name = "threadPoolTaskExecutor")
2 public Executor threadPoolTaskExecutor() {
3     ThreadPoolExecutor executor = new ThreadPoolExecutor();
4
5     executor.setCorePoolSize(4);
6     executor.setMaxPoolSize(4);
7     executor.setQueueCapacity(15);
8     executor.setThreadNamePrefix("RunnerThread::");
9     executor.initialize();
10
11     return executor;
12 }
```

```
1 @Async("threadPoolTaskExecutor")
2 private CompletableFuture<String> generateLong() throws InterruptedException {
3     return CompletableFuture.completedFuture("A long password: " + generator.generate(10
4 }
```

ASYNCHRONOUS EXECUTORS

```
1 public class PasswordConfig implements AsyncConfigurer {  
2  
3     @Override  
4     public Executor getAsyncExecutor() {  
5         ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();  
6  
7         taskExecutor.setCorePoolSize(4);  
8         taskExecutor.setMaxPoolSize(4);  
9         taskExecutor.setQueueCapacity(50);  
10        taskExecutor.setThreadNamePrefix("RunnerThread::");  
11        taskExecutor.initialize();  
12  
13        return taskExecutor;  
14    }  
15 }
```

```
1 A long password: s7ts34pd8a  
2 A short password: s$@ds  
3 Thread[RunnerThread::2,5,main]  
4 Thread[RunnerThread::1,5,main]
```

SPRING WEB

INTRODUCTION TO SPRING WEB MVC

DEPENDENCY

```
1 dependencies {  
2     // ...  
3     implementation 'org.springframework.boot:spring-boot-starter-web'  
4     // ...  
5 }
```

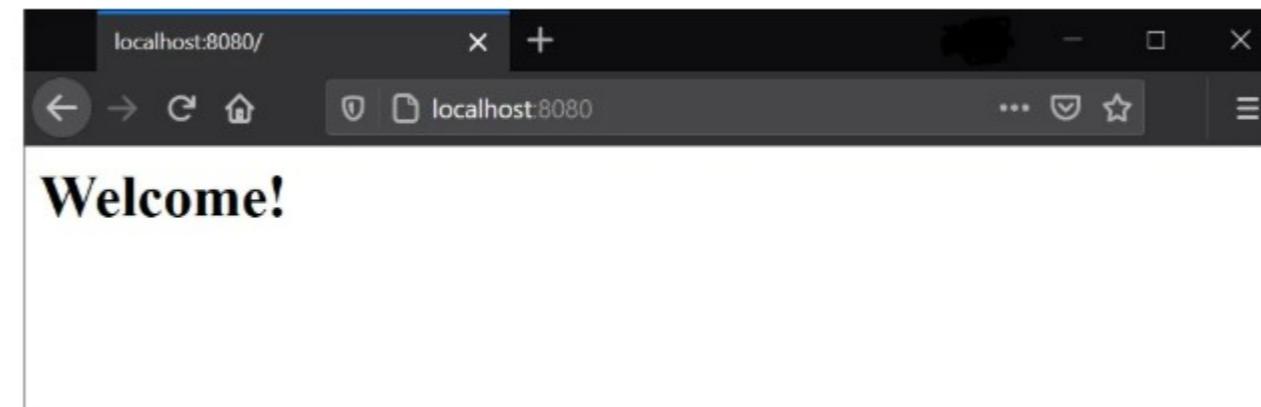
LOG

```
...
... INFO 11084 ... o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
... INFO 11084 ... o.apache.catalina.core.StandardService : Starting service [Tomcat]
... INFO 11084 ... org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.45]
... INFO 11084 ... o.a.c.c.C.[Tomcat].[localhost].[/]      : Initializing Spring embedded WebApplicationContext
... INFO 11084 ... w.s.c.ServletWebServerApplicationContext: Root WebApplicationContext:initialization completed in 1044ms
... INFO 11084 ... o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
... INFO 11084 ... o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
...
```

WEB PAGE

index.html file:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Main</title>
6 </head>
7 <body>
8     <h1>Welcome!</h1>
9 </body>
10 </html>
```



CONFIGURATION

We can change the port by including the following line in `application.properties` :

```
1 server.port=9090
```

Here is how you can change the context path in Spring Boot:

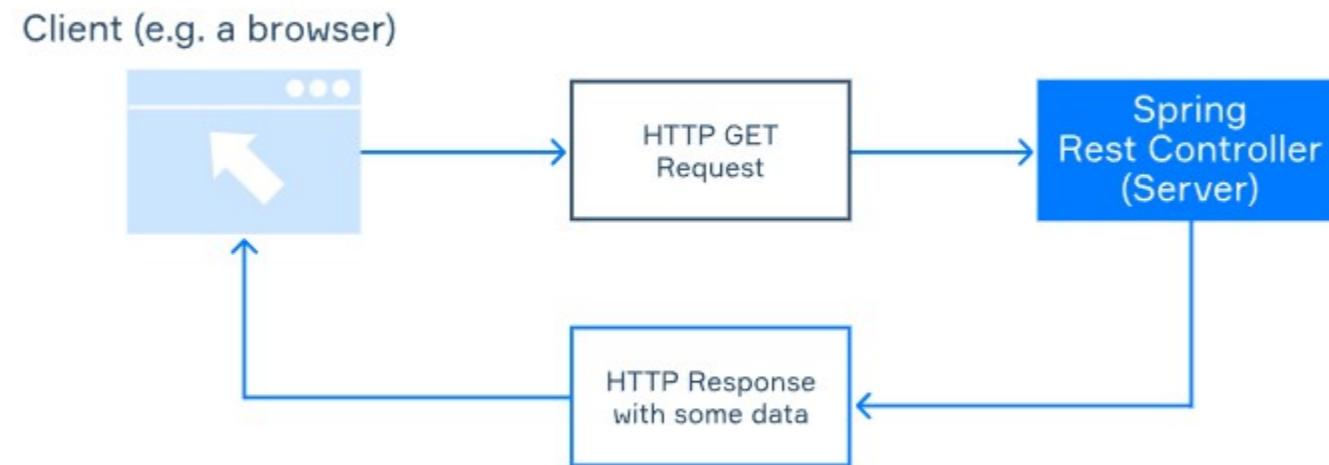
```
1 server.servlet.context-path=/myapp
```

The console log includes these changes:

```
1 ... Tomcat started on port(s): 9090 (http) with context path '/myapp'
```

Our web app can be accessed at <http://localhost:9090/myapp>

GETTING DATA FROM REST



REST CONTROLLER

The `@RestController` annotation is a wrapper of two different annotations:

- `@Controller` contains handler methods for various requests. Since we opted for `@RestController`, the methods are related to REST requests.
- `@ResponseBody` binds the return value of each handler method to a web response body. They will be represented in JSON format. When we send a request, the response we receive is in JSON format.

REST CONTROLLER

```
1 @RestController
2 public class TaskController {
3
4     @GetMapping("/test")
5     public int returnOne() {
6         return 1;
7     }
8 }
```

The screenshot shows the Postman application interface. At the top, it says "localhost:8080/test". Below that, there's a "GET" method selected and the URL "localhost:8080/test". Under the "Params" tab, there is a table with one row:

KEY	VALUE
Key	Value

At the bottom, under the "Body" tab, the response is displayed as JSON:

```
1 1
```

GET WITH COLLECTIONS

```
1 public class Task {  
2     private int id;  
3     private String name;  
4     private String description;  
5     private boolean completed;  
6  
7     public Task() {}  
8  
9     public Task(int id, String name, String description, boolean completed) {  
10         this.id = id;  
11         this.name = name;  
12         this.description = description;  
13         this.completed = completed;  
14     }  
15  
16     // getters and setters  
17 }
```

```
1 @RestController  
2 public class TaskController {  
3     private final List<Task> taskList = List.of(  
4             new Task(1, "task1", "A first test task", false),  
5             new Task(2, "task2", "A second test task", true)  
6     );  
7  
8     @GetMapping("/tasks")  
9     public List<Task> getTasks() {  
10         return taskList;  
11     }  
12 }
```

GET WITH COLLECTIONS

localhost:8080/tasks/

The screenshot shows the Postman interface with a GET request to `localhost:8080/tasks/`. The 'Params' tab is selected, showing a single query parameter named 'Key'. The 'Body' tab is selected, displaying a JSON response with two tasks:

```
1 [  
2 {  
3   "id": 1,  
4   "name": "task1",  
5   "description": "A first test task",  
6   "completed": false  
7 },  
8 {  
9   "id": 2,  
10  "name": "task2",  
11  "description": "A second test task",  
12  "completed": true  
13 }]  
14 ]
```

@PATHVARIABLE

```
1 @RestController
2 public class TaskController {
3     private final List<Task> taskList = List.of(
4         new Task(1, "task1", "A first test task", false),
5         new Task(2, "task2", "A second test task", true)
6     );
7
8     @GetMapping("/tasks/{id}")
9     public Task getTask(@PathVariable int id) {
10         return taskList.get(id - 1); // list indices start from 0
11     }
12 }
```

The screenshot shows the Postman application interface. At the top, the URL `http://localhost:8080/tasks/1` is entered. Below it, a GET request is selected with the URL `http://localhost:8080/tasks/1`. The 'Params' tab is active, showing a single entry: 'Key' under 'KEY' and 'Value' under 'VALUE'. In the 'Body' tab, the response is displayed as a JSON object:

```
1 {  
2     "id": 1,  
3     "name": "task1",  
4     "description": "A first test task",  
5     "completed": false  
6 }
```

POSTING AND DELETING DATA VIA REST

@POSTMAPPING

```
1 @RestController
2 public class AddressController {
3     private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();
4
5     @PostMapping("/addresses")
6     public void postAddress(@RequestParam String name, @RequestParam String address) {
7         addressBook.put(name, address);
8     }
9 }
```

localhost:8080/addresses?name=Bob&address=123 Younge Street

KEY	VALUE
<input checked="" type="checkbox"/> name	Bob
<input checked="" type="checkbox"/> address	123 Younge Street
Key	Value

Response

@POSTMAPPING

```
1 @RestController
2 public class AddressController {
3     private ConcurrentHashMap<String, String> addressBook = new ConcurrentHashMap<>();
4
5     @PostMapping("/addresses")
6     public void postAddress(@RequestParam String name, @RequestParam String address) {
7         addressBook.put(name, address);
8     }
9
10    @GetMapping("/addresses/{name}")
11    public String getAddress(@PathVariable String name) {
12        return addressBook.get(name);
13    }
14 }
```

localhost:8080/addresses/Bob

GET localhost:8080/addresses/Bob

Params Authorization Headers (7) Body Pre-request Script

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text

1 123 Younge Street

@POSTMAPPING

When a POST parameter is either missing or invalid:

The screenshot shows the Postman interface with the following details:

- URL:** `localhost:8080/addresses?name=Bob`
- Method:** POST
- Params Tab:** Active. Shows a table for "Query Params" with one row: `name` (checked) with value `Bob`. An empty row for "Key" and "Value" is also present.
- Headers Tab:** (8 items)
- Body Tab:** Active. Shows a JSON response with line numbers:

```
1 {  
2   "timestamp": "2021-05-02T23:10:46.541+00:00",  
3   "status": 400,  
4   "error": "Bad Request",  
5   "message": "",  
6   "path": "/addresses"  
7 }
```
- Cookies Tab:** (0 items)
- Headers Tab:** (4 items)
- Test Results Tab:** (0 items)

@DELETEMAPPING

```
1 @RestController
2 public class AddressController {
3     private ConcurrentMap<String, String> addressBook = new ConcurrentHashMap<>();
4
5     @DeleteMapping("/addresses")
6     public String removeAddress(@RequestParam String name) {
7         addressBook.remove(name);
8         return name + " removed from address book!";
9     }
10 }
```

HANDLING REQUESTS WITH BODIES

SENDING AN OBJECT TO THE SERVER

```
1 public class UserInfo {  
2  
3     private int id;  
4     private String name;  
5     private String phone;  
6     private boolean enabled;  
7  
8     /* getters and setters */  
9  
10    UserInfo() {}  
11 }
```

```
1 @RestController  
2 public class UserInfoController {  
3  
4     @PostMapping("/user")  
5     public String userStatus(@RequestBody UserInfo user) {  
6         if (user.isEnabled()) {  
7             return String.format("Hello! %s. Your account is enabled.", user.getName());  
8         } else {  
9             return String.format(  
10                 "Hello! Nice to see you, %s! Your account is disabled",  
11                 user.getName()  
12             );  
13         }  
14     }  
15 }  
16 }
```

SENDING AN OBJECT TO THE SERVER

localhost:8080/user

POST localhost:8080/user

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {  
2   "id":1,  
3   "name":"Scott",  
4   "phone":"555-555-5647",  
5   "enabled":true  
6 }
```

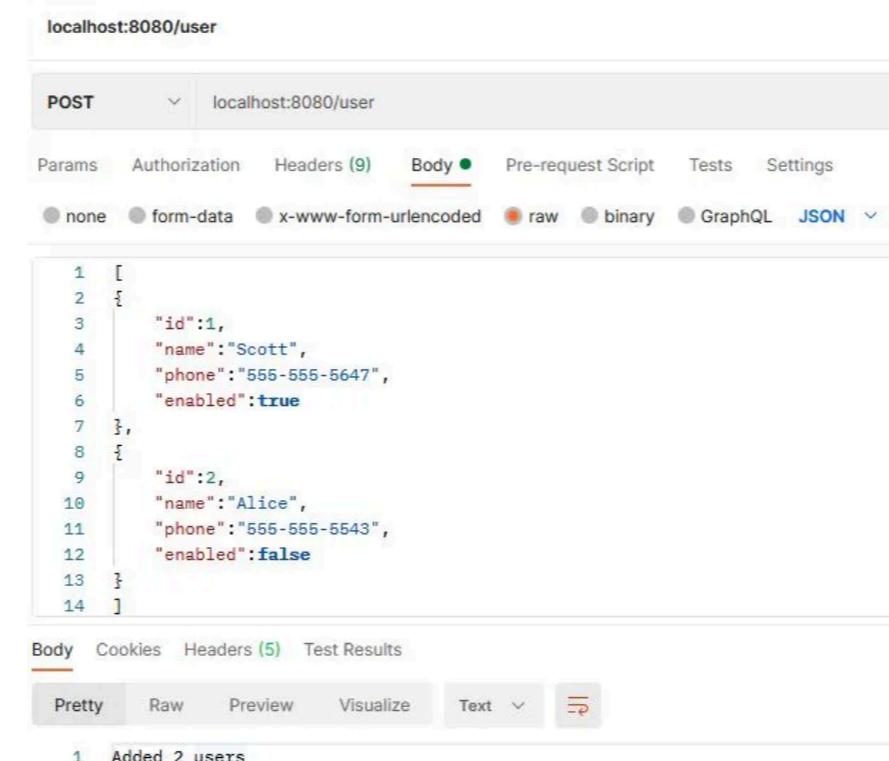
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text

```
1 Hello! Scott. Your account is enabled.
```

SENDING MULTIPLE OBJECTS

```
1 @RestController
2 public class UserInfoController {
3
4     @PostMapping("/user")
5     public String userStatus(@RequestBody List<UserInfo> userList) {
6         return String.format("Added %d users", userList.size());
7     }
8 }
```



ADDITIONAL DATA FORMATS

```
1 @RestController
2 public class UserInfoController {
3
4     @PostMapping(value = "/user", consumes = MediaType.APPLICATION_XML_VALUE)
5     public String userStatus(@RequestBody UserInfo user) {
6         return String.format("Added User %s", user);
7     }
8 }
```

RESPONSE BODIES

SERIALIZATION

```
1 import org.springframework.web.bind.annotation.GetMapping;
2 import org.springframework.web.bind.annotation.RestController;
3
4 @RestController
5 public class DemoController {
6
7     @GetMapping("/person")
8     public Person getPerson() {
9         return new Person("John", "Doe", 25);
10    }
11
12    public class Person {
13        private String firstName;
14        private String lastName;
15        private int age;
16
17        // Constructors, getters and setters are omitted for brevity
18    }
19 }
```

```
1 {
2     "firstName": "John",
3     "lastName": "Doe",
4     "age": 25
5 }
```

NESTED OBJECTS

```
1 public class Address {  
2     private String street;  
3     private String city;  
4     private String country;  
5  
6     /* Constructors, getters and setters are omitted for brevity */  
7 }  
8  
9 public class Person {  
10    private String firstName;  
11    private String lastName;  
12    private int age;  
13    private Address address;  
14  
15    /* Constructors, getters and setters are omitted for brevity */  
16 }
```

```
1 Person person = new Person("John", "Doe", 25, new Address("Olesvej", "Qaqortoq", "Greenland"))
```

The resulting JSON string:

```
1 {  
2     "firstName": "John",  
3     "lastName": "Doe",  
4     "age": 25,  
5     "address": {  
6         "street": "Olesvej",  
7         "city": "Qaqortoq",  
8         "country": "Greenland"  
9     }  
10 }
```

CIRCULAR REFERENCES

```
1 public class Parent {  
2     private String name;  
3  
4     @JsonManagedReference  
5     private List<Child> children;  
6  
7     /* Constructors, getters and setters are omitted for brevity */  
8 }  
9  
10 public class Child {  
11     private String name;  
12  
13     @JsonBackReference  
14     private Parent parent;  
15  
16     /* Constructors, getters and setters are omitted for brevity */  
17 }
```

```
1 {  
2     "name": "parent",  
3     "children": [  
4         {  
5             "name": "child 1"  
6         },  
7         {  
8             "name": "child 2"  
9         }  
10    ]  
11 }
```

JSON CUSTOMIZATION

```
1 public class Person {  
2     private String firstName;  
3     private String lastName;  
4  
5     @JsonIgnore  
6     private String password;  
7  
8     // methods omitted for brevity  
9  
10 }
```

```
1 {  
2     "firstName": "John",  
3     "lastName": "Doe"  
4 }
```

```
1 public class Person {  
2     @JsonProperty("first_name")  
3     private String firstName;  
4  
5     @JsonProperty("last_name")  
6     private String lastName;  
7  
8     private int age;  
9  
10    // Constructors, getters and setters are omitted for brevity  
11 }
```

POST VS. PUT REQUESTS

IMPLEMENTING PUT

```
1 public class Employee {  
2  
3     private long id;  
4     private String name;  
5     private int age;  
6  
7     // getters and setters  
8 }
```

```
1 @RestController  
2 public class EmployeeController {  
3     private ConcurrentMap<Long, Employee> employeeMap = new ConcurrentHashMap<>();  
4  
5     @PutMapping("/employees/{id}")  
6     public Employee updateEmployee(@PathVariable long id, @RequestBody Employee employee)  
7         employeeMap.put(id, employee);  
8         return employee;  
9     }  
10  
11    @GetMapping("/employees/{id}")  
12    public Employee getEmployee(@PathVariable long id) {  
13        return employeeMap.get(id);  
14    }  
15 }
```

IMPLEMENTING PUT

localhost:8080/employees/1

PUT localhost:8080/employees/1

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {  
2   "id":1,  
3   "name":"Scott",  
4   "age":21  
5 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "id": 1,  
3   "name": "Scott",  
4   "age": 21  
5 }
```

localhost:8080/employees/1

GET localhost:8080/employees/1

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "id": 1,  
3   "name": "Scott",  
4   "age": 21  
5 }
```

IMPLEMENTING PUT

localhost:8080/employees/1

The screenshot shows a POSTMAN interface with the following details:

- Method: PUT
- URL: localhost:8080/employees/1
- Body tab selected
- JSON selected under Body type dropdown
- Body content:

```
1 "id":1,
2   "name":"Alice",
3   "age":21
4
5
```
- Pretty tab selected under Preview dropdown
- Preview content:

```
1 {
2   "id": 1,
3   "name": "Alice",
4   "age": 21
5 }
```

localhost:8080/employees/1

The screenshot shows a POSTMAN interface with the following details:

- Method: GET
- URL: localhost:8080/employees/1
- Body tab selected
- JSON selected under Body type dropdown
- Body content:

```
1 "id":1,
2   "name":"Alice",
3   "age":21
4
5
```
- Pretty tab selected under Preview dropdown
- Preview content:

```
1 {
2   "id": 1,
3   "name": "Alice",
4   "age": 21
5 }
```

SERVING FILES

SENDING FILES

```
1 import org.springframework.core.io.PathResource;
2 import org.springframework.core.io.Resource;
3 import org.springframework.http.MediaType;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.RestController;
7 import java.nio.file.Path;
8
9 @RestController
10 class FileController {
11     @GetMapping(path = "/download")
12     public ResponseEntity<Resource> download() {
13         Path path = Path.of("files", "example.txt");
14         Resource file = new PathResource(path);
15         return ResponseEntity
16             .ok()
17             .contentType(MediaType.TEXT_PLAIN)
18             .header("Content-Disposition", "attachment; filename=" + file.getFilename())
19             .body(file);
20     }
21 }
```

SENDING FILES ASYNCHRONOUSLY

```
1 @RestController
2 class FileController {
3     @GetMapping("/stream")
4     public ResponseEntity<StreamingResponseBody> stream() {
5         Path path = Path.of("data", "bigdata.zip");
6         /* Creating a Resource from the path */
7         Resource resource = new PathResource(path);
8         /* Creating the StreamingResponseBody */
9         StreamingResponseBody responseBody = outputStream -> {
10             try (InputStream inputStream = resource.getInputStream()) {
11                 byte[] buffer = new byte[4096];
12                 int bytesRead;
13                 while ((bytesRead = inputStream.read(buffer)) != -1) {
14                     outputStream.write(buffer, 0, bytesRead);
15                 }
16             } catch (IOException e) {
17                 /* handle the exception here */
18             }
19         };
20         /* Setting the response headers */
21         HttpHeaders headers = new HttpHeaders();
22         headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
23         try {
24             headers.setContentLength(resource.contentLength());
25         } catch (IOException e) {
26             /* handle the exception */
27         }
28         headers.setContentDispositionFormData("attachment", resource.getFilename());
29         /* Assembling the response */
30         return ResponseEntity
31             .ok()
32             .headers(headers)
33             .body(responseBody);
34     }
35 }
```

RECEIVING MULTIPART FILES

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>File Uploader</title>
6 </head>
7 <body>
8 <h2>Upload File</h2>
9 <form action="/upload" method="post" enctype="multipart/form-data">
10    <label for="sender">Enter your name: </label>
11    <input type="text" id="sender" name="sender"><br>
12    <label for="file">Choose file: </label>
13    <input type="file" id="file" name="file">
14    <input type="submit">
15 </form>
16 </body>
17 </html>
```

```
1 @PostMapping("/upload")
2 public String upload(@RequestParam String sender, @RequestParam MultipartFile file) {
3     Path path = Path.of("uploads", file.getOriginalFilename());
4     try {
5         file.transferTo(path);
6     } catch (IOException e) {
7         /* handle the exception */
8         return e.getClass().getSimpleName() + ": Error uploading the file " + e.getMessage();
9     }
10    return file.getOriginalFilename() + " successfully uploaded by " + sender;
11 }
```

RECEIVING FILES AS REQUEST BODIES

```
1 @PostMapping("/upload")
2 public String upload(@RequestBody byte[] bytes) {
3     Path path = Path.of("uploads", UUID.randomUUID().toString());
4     try {
5         Files.write(path, bytes);
6     } catch (IOException e) {
7         return "Error uploading file";
8     }
9     return "File successfully uploaded";
10 }
```

CUSTOMIZING REST RESPONSES

HTTP RESPONSES

- **Status code.** This is a three-digit number sent by the server to indicate the result of the requested resource. For example, `200` means OK, `404` means Not Found, `500` means Internal Server Error, and so on.
- **Headers** provide additional information about the response or the request. One important header is `Content-Type`, which specifies the media type of the resource. For example, `Content-Type: application/json` indicates that the server is returning JSON data.
- **Body** (optional). This is the actual content of the response. It can be data in the form of text or other formats like JSON, XML, HTML, and others.

HTTP RESPONSES

```
1 import org.springframework.http.HttpStatus;
2 import org.springframework.web.bind.annotation.GetMapping;
3 import org.springframework.web.bind.annotation.ResponseStatus;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class DemoController {
8
9     @GetMapping(path = "/hello", produces = "text/plain")
10    @ResponseStatus(HttpStatus.OK)
11    public String hello() {
12        return "Hello World";
13    }
14 }
```

HTTPSERVLETRESPONSE

```
1 import jakarta.servlet.http.HttpServletResponse;
2 import org.springframework.web.bind.annotation.GetMapping;
3 import org.springframework.web.bind.annotation.RestController;
4
5 import java.io.IOException;
6
7 @RestController
8 public class DemoController {
9
10     @GetMapping(path = "/hello")
11     public void hello(HttpServletResponse response) throws IOException {
12         response.setStatus(HttpServletResponse.SC_OK);
13         response.setContentType("text/plain");
14         response.getWriter().write("Hello World");
15     }
16 }
```

RESPONSE ENTITY

- `status(HttpStatusCode)` : sets the status code of the response, such as `HttpStatus.BAD_REQUEST`.
- `header(String, String)` : adds a single header value under the given name.
- `headers(HttpHeaders)` : sets all headers with the given `HttpHeaders` instance.
- `body(T)` : sets the body of the response.
- `notFound()` : sets the 404 Not Found status code.

RESPONSE ENTITY

```
1 import org.springframework.http.MediaType;
2 import org.springframework.http.ResponseEntity;
3 import org.springframework.web.bind.annotation.*;
4 import java.awt.*;
5 import java.awt.image.BufferedImage;
6 import java.util.Map;
7
8 @RestController
9 public class DemoController {
10     private static final Map<String, BufferedImage> images = Map.of(
11         "green", createImage(Color.GREEN),
12         "magenta", createImage(Color.MAGENTA)
13     );
14     @GetMapping(path = "/image")
15     public ResponseEntity<BufferedImage> getImage(@RequestParam String name,
16                                                     @RequestParam String mediaType) {
17         BufferedImage image = images.get(name);
18         if (image == null) {
19             return ResponseEntity.notFound().build();
20         }
21         return ResponseEntity.ok()
22             .contentType(MediaType.parseMediaType(mediaType))
23             .body(image);
24     }
25
26     private static BufferedImage createImage(Color color) {
27         BufferedImage image = new BufferedImage(20, 20, BufferedImage.TYPE_INT_RGB);
28         Graphics2D g = image.createGraphics();
29         g.setColor(color);
30         g.fillRect(0, 0, 20, 20);
31         g.dispose();
32         return image;
33     }
34 }
```

RESTTEMPLATE

RESPONSE HANDLING

```
1 RestTemplate restTemplate = new RestTemplate();
2
3 String url = "https://jsonplaceholder.typicode.com/posts";
4
5 ResponseEntity<List<Post>> response = restTemplate.exchange(url, HttpMethod.GET, null, n
6
7 List<Post> posts = response.getBody();
```

```
1 [
2   {
3     "userId": 1,
4     "id": 1,
5     "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
6     "body": "quia...eveniet architecto"
7   },
8   {
9     "userId": 1,
10    "id": 2,
11    "title": "qui est esse",
12    "body": "est...aperiam non debitis possimus qui neque nisi nulla"
13  }
14 ]
```

RESPONSE HANDLING

```
1 public class Post {  
2  
3     private long userId;  
4     private long id;  
5     private String title;  
6     private String body;  
7  
8     public Post(long userId, String title, String body) {  
9         this.userId = userId;  
10        this.title = title;  
11        this.body = body;  
12    }  
13  
14    public Post(long userId, long id, String title, String body) {  
15        this.userId = userId;  
16        this.id = id;  
17        this.title = title;  
18        this.body = body;  
19    }  
20}
```

GETTING STATUS CODE

```
1 RestTemplate restTemplate = new RestTemplate();
2
3 String apiUrl = "https://jsonplaceholder.typicode.com/posts/{id}";
4
5 Map<String, String> params = new HashMap<>();
6
7 params.put("id", "1");
8
9 ResponseEntity<Post> response = restTemplate.getForEntity(apiUrl, Post.class, params);
10
11 HttpStatus statusCode = response.getStatusCode();
```

HEADERS

```
1 RestTemplate restTemplate = new RestTemplate();
2
3 String apiUrl = "https://jsonplaceholder.typicode.com/posts/{id}";
4
5 Map<String, String> params = new HashMap<>();
6
7 params.put("id", "1");
8
9 ResponseEntity<Post> response = restTemplate.getForEntity(apiUrl, Post.class, params);
10
11 HttpHeaders headers = response.getHeaders();
12
13 String headerValue = headers.getFirst("header-name");
```

REQUEST BODIES

```
1 RestTemplate restTemplate = new RestTemplate();
2
3 String url = "https://jsonplaceholder.typicode.com/posts";
4
5 Post post = new Post(0, "Title", "Description");
6
7 HttpHeaders headers = new HttpHeaders();
8
9 headers.setContentType(MediaType.APPLICATION_JSON);
10
11 HttpEntity<Post> request = new HttpEntity<>(post, headers);
12
13 ResponseEntity<Post> response = restTemplate.postForEntity(url, request, Post.class);
14
15 Post createdPost = response.getBody();
```

GET REQUESTS

```
1 RestTemplate restTemplate = new RestTemplate();
2
3 String apiUrl = "https://jsonplaceholder.typicode.com/posts/{id}";
4
5 Map<String, String> params = new HashMap<>();
6
7 params.put("id", "1");
8
9 Post post = restTemplate.getForObject(apiUrl, Post.class, params);
```

POST REQUESTS

```
1 String url = "https://jsonplaceholder.typicode.com/posts";
2
3 Post post = new Post(1, "Title", "Description");
4
5 ResponseEntity<Post> response = restTemplate.postForEntity(url, post, Post.class);
6
7 Post createdPost = response.getBody();
```

PUT REQUESTS

```
1 String url = "https://jsonplaceholder.typicode.com/posts/{id}";  
2  
3 Map<String, String> params = new HashMap<>();  
4  
5 params.put("id", "1");  
6  
7 Post updatedPost = new Post(2, "Food", "I like beef");  
8  
9 restTemplate.put(url, updatedPost, params);
```

DELETE REQUESTS

```
1 String url = "https://jsonplaceholder.typicode.com/posts/{id}";  
2  
3 Map<String, String> params = new HashMap<>();  
4  
5 params.put("id", "2");  
6  
7 restTemplate.delete(url, params);
```

EXCEPTION HANDLING IN SPRING BOOT

CONTROLLER PREPARATION

```
1 {
2   "id" : 3,
3   "from": "Berlin Tegel",
4   "to": "Stuttgart",
5   "gate": "D80"
6 }
```

```
1 public class FlightInfo {
2
3     private int id;
4     private String from;
5     private String to;
6     private String gate;
7
8     /* constructor
9
10    getters and setters */
11
12 }
```

CONTROLLER PREPARATION

```
1 @RestController
2 public class FlightController {
3
4     private final List<FlightInfo> flightInfoList = new ArrayList<>();
5
6     public FlightController() {
7         flightInfoList.add(
8             new FlightInfo(1, "Delhi Indira Gandhi", "Stuttgart", "D80"));
9         flightInfoList.add(
10            new FlightInfo(2, "Tokyo Haneda", "Frankfurt", "110"));
11        flightInfoList.add(
12            new FlightInfo(3, "Berlin Schönefeld", "Tenerife", "15"));
13        flightInfoList.add(
14            new FlightInfo(4, "Kilimanjaro Arusha", "Boston", "15"));
15    }
16
17    @GetMapping("flights/{id}")
18    public FlightInfo getFlightInfo(@PathVariable int id) {
19        for (FlightInfo flightInfo : flightInfoList) {
20            if (flightInfo.getId() == id) {
21                return flightInfo;
22            }
23        }
24        throw new RuntimeException();
25    }
26
27 }
```

RESPONSESTATUSEXCEPTION

There are three constructors in Spring to generate `ResponseStatusException` :

```
1 ResponseStatusException(HttpStatus status)
2 ResponseStatusException(HttpStatus status, java.lang.String reason)
3 ResponseStatusException(
4         HttpStatus status,
5         java.lang.String reason,
6         java.lang.Throwable cause
7 )
```

`HttpStatus` types:

- `200 OK` `404 NOT_FOUND` `400 BAD_REQUEST` `403 FORBIDDEN` `500 INTERNAL_SERVER_ERROR`

RESPONSESTATUSEXCEPTION

```
1 @GetMapping("flights/{id}")
2 public FlightInfo getFlightInfo(@PathVariable int id) {
3     for (FlightInfo flightInfo : flightInfoList) {
4         if (flightInfo.getId() == id) {
5             if (Objects.equals(flightInfo.getFrom(), "Berlin Schönefeld")) {
6                 throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
7                     "Berlin Schönefeld is closed for service today");
8             } else {
9                 return flightInfo;
10            }
11        }
12    }
13    throw new RuntimeException();
14 }
```

The screenshot shows a POST request to `localhost:8080/flights/3`. The **Params** tab is selected, showing a table with one row: `Key` (Value) and `Description` (Description). The **Body** tab is selected, showing a JSON response:

```
1 [
2     "timestamp": "2021-12-21T14:10:15.462+00:00",
3     "status": 400,
4     "error": "Bad Request",
5     "message": "Berlin Schönefeld is closed for service today",
6     "path": "/flights/3"
7 ]
```

The status bar at the bottom right indicates `Status: 400 Bad Request`.

CUSTOM EXCEPTIONS

```
1 @ResponseStatus(code = HttpStatus.BAD_REQUEST)
2 class FlightNotFoundException extends RuntimeException {
3     public FlightNotFoundException(String message) {
4         super(message);
5     }
6 }
```

```
1 @GetMapping("flights/{id}")
2 public FlightInfo getFlightInfo(@PathVariable int id) {
3     for (FlightInfo flightInfo : flightInfoList) {
4         if (flightInfo.getId() == id) {
5             return flightInfo;
6         }
7     }
8     throw new FlightNotFoundException("Flight not found for id =" + id);
9 }
```

The screenshot shows a Postman request for a GET operation at `localhost:8080/flights/1111`. The 'Params' tab is selected, showing a single query parameter 'Key' with value 'Value'. In the 'Body' tab, the status is shown as '400 Bad Request'. The JSON response body is displayed in 'Pretty' format:

```
1 [
2     "timestamp": "2021-12-21T14:16:16.826+00:00",
3     "status": 400,
4     "error": "Bad Request",
5     "message": "Flight not found for id =1111",
6     "path": "/flights/1111"
7 ]
```

ADVANCED EXCEPTION HANDLING IN SPRING BOOT

CONTROLLER PREPARATION

```
1 public class FlightInfo {  
2  
3     private long id;  
4  
5     private String from;  
6  
7     private String to;  
8  
9     private String gate;  
10  
11    // constructor, getters, setters  
12 }
```

```
1 public class FlightNotFoundException extends RuntimeException {  
2  
3     public FlightNotFoundException(String message) {  
4         super(message);  
5     }  
6 }
```

CONTROLLER PREPARATION

```
1 @RestController
2 public class FlightController {
3
4     private final List<FlightInfo> flightInfoList = Collections.synchronizedList(
5         new ArrayList<>());
6
7     public FlightController() {
8         flightInfoList.add(
9             new FlightInfo(
10                1,
11                "Delhi Indira Gandhi",
12                "Stuttgart",
13                "D80"));
14        flightInfoList.add(
15            new FlightInfo(
16                2,
17                "Tokyo Haneda",
18                "Frankfurt",
19                "110"));
20    }
21
22    @GetMapping("flights/{id}")
23    public FlightInfo getFlightInfo(@PathVariable long id) {
24        for (var flightInfo : flightInfoList) {
25            if (flightInfo.getId() == id) {
26                return flightInfo;
27            }
28        }
29        throw new FlightNotFoundException("Flight info not found id=" + id);
30    }
31 }
32 }
```

@EXCEPTIONHANDLER

```
1 public class CustomErrorMessage {
2     private int statusCode;
3     private LocalDateTime timestamp;
4     private String message;
5     private String description;
6
7     public CustomErrorMessage(
8         int statusCode,
9         LocalDateTime timestamp,
10        String message,
11        String description) {
12
13         this.statusCode = statusCode;
14         this.timestamp = timestamp;
15         this.message = message;
16         this.description = description;
17     }
18
19     // getters ...
20 }
```

```
1 @ExceptionHandler(FlightNotFoundException.class)
2 public ResponseEntity<CustomErrorMessage> handleFlightNotFound(
3     FlightNotFoundException e, WebRequest request) {
4
5     CustomErrorMessage body = new CustomErrorMessage(
6         HttpStatus.NOT_FOUND.value(),
7         LocalDateTime.now(),
8         e.getMessage(),
9         request.getDescription(false));
10
11     return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
12 }
```

@CONTROLLERADVICE

```
1 @ControllerAdvice
2 public class ControllerExceptionHandler {
3
4     @ExceptionHandler(FlightNotFoundException.class)
5     public ResponseEntity<CustomErrorMessage> handleFlightNotFound(
6         FlightNotFoundException e, WebRequest request) {
7
8         CustomErrorMessage body = new CustomErrorMessage(
9             HttpStatus.NOT_FOUND.value(),
10            LocalDateTime.now(),
11            e.getMessage(),
12            request.getDescription(false));
13
14         return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
15     }
16 }
```

The screenshot shows the Postman application interface. At the top, it displays a GET request to `http://localhost:8080/flights/1000`. Below the URL, there are tabs for Params, Authorization, Headers (8), Body (green dot), Pre-request Script, Tests, and Settings. Under the Params tab, there is a section for Query Params with a table header: KEY, VALUE, DESCRIPTION. The Body tab is selected, showing the response status as `Status: 404 Not Found`. Below the status, there are tabs for Body (red dot), Cookies, Headers (5), and Test Results. The Body tab is active, displaying a JSON response with the following content:

```
1 {"statusCode": 404,
2 "timestamp": "2022-04-29T14:22:42.393+00:00",
3 "message": "Not found flight id = 1000",
4 "description": "uri=/flights/1000"
5 }
```

RESPONSE ENTITY EXCEPTION HANDLER

```
1 public class FlightInfo {  
2  
3     @Min(1)  
4     private long id;  
5  
6     private String from;  
7  
8     private String to;  
9  
10    private String gate;  
11  
12    //getters...  
13 }
```

```
1 @RestController  
2 public class FlightController {  
3  
4     private final List<FlightInfo> flightInfoList = Collections.synchronizedList(  
5             new ArrayList<>());  
6  
7     /* constructor */  
8  
9     /* getFlightInfo method*/  
10  
11    @PostMapping("/flights/new")  
12    public void addNewFlightInfo(@Valid @RequestBody FlightInfo flightInfo) {  
13        flightInfoList.add(flightInfo);  
14    }  
15 }
```

RESPONSE ENTITY EXCEPTION HANDLER

```
1 @ControllerAdvice
2 public class ControllerExceptionHandler extends ResponseEntityExceptionHandler {
3     /* handleFlightNotFound method */
4     @Override
5     protected ResponseEntity<Object> handleMethodArgumentNotValid(
6         MethodArgumentNotValidException ex,
7         HttpHeaders headers,
8         HttpStatusCode status,
9         WebRequest request) {
10        /* Just like a POJO, a Map is also converted to a JSON key-value structure */
11        Map<String, Object> body = new LinkedHashMap<>();
12        body.put("status", status.value());
13        body.put("timestamp", LocalDateTime.now());
14        body.put("exception", ex.getClass());
15        return new ResponseEntity<>(body, headers, status);
16    }
17 }
```

The screenshot shows a Postman interface with a POST request to `http://localhost:8080/flights/new`. The request body is a JSON object with an invalid 'id' value (-1). The response is a 400 Bad Request with a JSON body containing the error details.

Request Body:

```
1 {"id": -1,
2   "from": "Moscow",
3   "to": "New York",
4   "gate": "10"}
```

Response Body:

```
1 {"status": 400,
2  "timestamp": "2022-05-07T14:24:26.6085563",
3  "exception": "org.springframework.web.bind.MethodArgumentNotValidException"}
```

BEAN VALIDATION

DATA VALIDATION

```
1 {
2     "name": "James",
3     "surname": "Bond",
4     "code": "007",
5     "status": "special agent",
6     "age": 51
7 }
```

```
1 public class SpecialAgent {
2
3     private String name;
4     private String surname;
5     private String code;
6     private String status;
7     private int age;
8
9     // getters and setters
```

Gradle:

```
1 implementation 'org.springframework.boot:spring-boot-starter-validation'
```

@NOTNULL, @NOTEMPTY, @NOTBLANK CONSTRAINTS

```
1 @NotNull
2 private String name;
3
4 @NotEmpty
5 private String motto;
6
7 @NotBlank
8 private String status;
```

@SIZE CONSTRAINT

```
1 @Size(min = 1, max = 3)
2 private String code;
```

```
1 @Size(min = 0, max = 4)
2 private List<String> cars;
```

@MIN, @MAX CONSTRAINTS

```
1 @Min(value = 18)
2 private int age;
3
4 @Max(5)
5 private int numberOfCurrentMissions;
```

@PATTERN AND @EMAIL

```
1 @Pattern(regexp = "[0-9]{1,3}")  
2 private String code;
```

```
1 @NotNull  
2 @Email  
3 private String email;
```

@VALID ANNOTATION

```
1 @RestController
2 public class SpecialAgentController {
3
4     @PostMapping("/agent")
5     public ResponseEntity<String> validate(@Valid @RequestBody SpecialAgent agent) {
6         return ResponseEntity.ok("Agent info is valid.");
7     }
8 }
```

```
1 @Min(value = 18, message = "Age must be greater than or equal to 18")
2 private int age;
```

@VALIDATED ANNOTATION

```
1 @RestController
2 @Validated
3 public class SpecialAgentController {
4
5     @GetMapping("/agents/{id}")
6     ResponseEntity<String> validateAgentPathVariable(@PathVariable("id") @Min(1) int id)
7         return ResponseEntity.ok("Agent id is valid.");
8     }
9
10    @GetMapping("/agents")
11    ResponseEntity<String> validateAgentRequestParam(
12        @RequestParam("code") @Pattern(regexp = "[0-9]{1,3}") String code) {
13        return ResponseEntity.ok("Agent code is valid.");
14    }
15 }
```



RESOURCES

- Effective Java 3rd Edition
- Spring Start Here
- Spring Academy Courses

