1. General

* Object oriented
* Platform independent – run on JVM
* **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
* **JDK** – Java Development Kit – the libraries required to develop a Java app
* **JRE** – Java Runtime Environment – the Java environment that run the compiled program’s bytecode
* **JVM** – Java Virtual Machine – part of the JRE.

## Pointers

* The declaration "Student x;" declares a pointer "x" to a Student object, but does not allocate the object yet.
* Objects and arrays are allocated in the heap and accessed through pointers.
* The only pointers that exist in java point to objects and arrays in the heap.
* There is no "&" operator to make a pointer to something in the stack and there is no pointer arithmetic.
* Objects and arrays are allocated with the "new" operator.
* **Shallow Copying:** Using = on an object pointer just copies the pointer, so there are multiple pointers to the one object (aka "shallow" or "sharing").
* **Shallow Comparison:** Using == on object pointers just compares the pointers (for some classes, the equals() method will do a "deep" comparison of two objects)

## Data Types

Primitive types – all signed!!:

* boolean-- true of false
* byte -- 1 byte
* char -- 2 bytes (unicode)
* int -- 4 bytes
* long -- 8 bytes
* float -- 4 bytes
* double - 8 bytes
* Unlike C, the sizes of the primitives are fixed and do not vary from one platform to another, and there are no unsigned variants.

### Constants

* To declare constants in Java the final keyword is used
* Final primitive – makes it immutable.
* Final object – makes the reference to the object immutable. Note: the object itself can still change, just the reference can’t point to another object (like const pointer in C++).
* Final members can be initialized in their declaration or in the class constructor.

## Access Modifiers

* Private - only this class can access the member
* Public – all classes can access this member
* **Protected** – all classes in the **same package** can access the member (like internal in C#). This is the default in Java if no access modifier is used.

## Static

## Operators

### Printouts

System.out.println(…)

System.err.println(…) – to print on the error stream.

### equals vs ==

* **==** compare the **reference** (location in memory) of the objects
* **equals** compare the **content** of the objects

|  |
| --- |
| public class Test {      public static void main(String[] args)      {          String s1 = new String("HELLO");          String s2 = new String("HELLO");          System.out.println(s1 == s2); // false          System.out.println(s1.equals(s2)); // true      }  } |

## Enum

enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }

* Enums in Java are a full fledged class
* Can implement methods, fields and even interfaces

## Arrays

## Strings

* Immutable  
  methods that appear to modify the actual content of a string actually create a new string to return, leaving the original string unchanged.
* int length() -- number of chars
* char charAt(int index)-- char at given 0-based index
* int indexOf(char c)-- first occurrence of char in the string, or -1-
* int indexOf(String s)
* boolean equals(Object)-- test if two strings have the same characters-
* boolean equalsIgnoreCase(Object) -- as above, but ignoring case-
* String toLowerCase()-- return a new String, lowercase
* String substring(int begin, int end)-- return a new String made of the begin..end-1 substring from the original.
* Support String.format(“This is %s string”, “my”); like in C++

### StringBuilder

Similar to String, but can change the chars over time. More efficient to change one StringBuilder over time, than to create 20 slightly different String objects over time.

## Map

* ConcurrentMap – allow multiple thread to work on it.
* ImmutableMap – can’t be changed (partially?)

### Useful Functions

* Remove items depending on condition:

map.entrySet()  
 .removeIf(entry -> entry.getValue().equals("0"));

* Traverse the map and update the elements:

for (Map.Entry<String, String> entry : myMap.entrySet()) {  
 entry.setValue(calculateValue(entry));  
}



## Streams

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

Intermediate operations are further divided into *stateless* and *stateful* operations:

* + - * **Stateless** operations, such as filter and map, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements.
      * **Stateful** operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.
* **Terminal operations**, such as Stream.forEach or IntStream.sum, may traverse the stream to **produce a result or a side-effect**. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations **are *eager***, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.
* some operations are deemed ***short-circuiting* operations**. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time. Having a short-circuiting operation in the pipeline is a necessary, but not sufficient, condition for the processing of an infinite stream to terminate normally in finite time.
* Stream.filter(condition) – will return a stream containing all the elements for which the {condition} is true.  
  // will print out all the elements in list that are   
  // dividable by 5:  
  list.stream()  
   .filter(num -> num % 5 ==0)  
   .forEach(System.out::println);
* Stream.collect – collect all the elements of the stream that satisfied the conditions. Collect is a terminal operation and therefore, it is always evaluated. So for example:  
  final List<String> result = someList.stream()  
   .filter(num -> num % 5 == 0)  
   .collect(Collectors.toList());  
  will always return a list (empty of not) – if the input is an empty list or if the filtering resulted in an empty list -> result will end up being an empty list. Never a null.

# Program

## Main

public class MyFirstJavaProgram {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\*/

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

System.out.println("Hello World"); // prints Hello World

}

}

## Compiler

In command line:

> javac <MyJavaFile.java>

Will compile the MyJavaFile.java to .exe.

## Build Tools

### Maven

Maven is a build and deployment tool and dependencies manager.

## Synchronizing Methods and code Blocks

In Java it is possible to specify synchronized blocks of code that ensure that only one thread can access a particular object at a time and then create a critical section of code.

public void withdrawAmount(int num){

synchronized(this){

if(num < this.amount)

this.amount -= num;

}

}

Both C# and Java support the concept of synchronized methods. Whenever a synchronized method is called, the thread that called the method locks the object that contains the method. Thus other threads cannot call a synchronized method on the same object until the object is unlocked by the first thread when it finishes executing the synchronized method. Synchronized methods are marked in Java by using the synchronized keyword

public class BankAccount{

public synchronized void withdrawAmount(int num){

if(num < this.amount)

this.amount - num;

}

}//BankAccount

## Threads

* Java threads are created by subclassing the java.lang.Thread class and overriding its run() method or by implementing the java.lang.Runnable interface and implementing the run() method.
* Every class inherits the wait(), notify() and notifyAll() from java.lang.Object which are used for thread operations.

class WorkerThread **extends Thread**{

…

public void run(){

…

}/\* run() \*/

} // WorkerThread

## Flow Control

### For-Each

String[] greek\_alphabet = {"alpha", "beta", "gamma", "delta", "epsilon"};

for(**String** str **: greek\_alphabet**)

System.out.println(str + " is a letter of the greek alphabet");

# Functions

## Variable Length Parameter List

By using <arg-type>… <array of args> :

class Test{

public static void PrintInts(String title, **Integer... args**){

System.out.println(title + ":");

for(int num : args)

System.out.println(num);

}

public static void main(String[] args){

PrintInts("First Ten Numbers in Fibonacci Sequence", 0, 1, 1, 2, 3, 5, 8, 13, 21, 34);

}

}

# Class

* New objects are set to all 0's first, then the ctor (if any) is run to further initialize the object. (Setting to all 0's avoids security problems where a new object re-uses memory that used to contain something secret, like a password.)

class B **extends** A **implements** Comparable{

int compareTo(){}

public static void main(String[] args){

System.out.println("Hello World");

}

}

* Calling the base class: with ‘super’ keyword.

## Interfaces

* There are limitations on using static methods and fields in interfaces. The exact details depend on the Java version.

## Constructor vs Static Factory

If a class requires multiple constructors, you can use static factory methods to initialize the class instead.

For example:

public class Time {  
 public static Time fromSeconds(int seconds){…}  
 public static Time fromMinutes(int seconds){…}  
 ….  
}

A good pattern is to call the factory methods ‘fromX’

### Static Factory Benefits

Additional advantages of using static factory methods instead of const:

* Factory methods don’t have to create a new object every time they are invoked. They can control the number of instances invoked.
* Factory methods can return any sub-type of their return type. For example, can return objects of private classes that derive from the base class
* The class of the returned object can vary from call to call depending on the input parameters.

### Static Factory Methods Drawbacks

* Classes without public or protected const cannot be subclassed.   
  Note: this can also be considered an advantage since it forces the developer to prefer composition over inheritance. In addition, this is required for immutable types.
* Naming: it’s harder for developer to know how to instantiate a class that has static methods instead of const. The way around it is to stick to common naming conventions such as:
  + from() – takes a type and return the corresponding instance of the class:

# Generics

* The Java compiler removes the generic-types in compilation and uses objects and casting under the hood instead of the actual boxes.

# Built-In Functions/Classes

## Optional<T>

like C# nullable type.

* See Brian Goetz (Java architect) response:  
  <http://stackoverflow.com/questions/26327957/should-java-8-getters-return-optional-type/26328555#26328555>

### Initialize

* With an empty object:   
  Optional<String> opt = Optional.empty();
* With a non-null object:  
  Optional<String> opt = Optional.of(someString);  
  Note: the call above will throw an exception if ‘someString’ is null!
* With an object that can be null:  
  Optional<String> opt = Optional.ofNullable(s);  
  s can be null and then opt will be empty() or initialized before the call.

### isPresent

returns true if the Optional contains a non-null value, otherwise it returns false.

* It’s not a good practice to use this function. Instead, it’s better to use .orElse() on the optional value itself.

### Get Value

* Don’t use get() – it will throw an exception if the optional is empty!!
* Instead, use one of the followings:
  + orElse() – will return the internal-value if it is not empty, otherwise, will return the orElse() value:

Optional<Person> personResult = findPerson(name);

Person person = personResult.orElse(Person.GUEST);

**Note:** the parameter of *orElse()*is **evaluated even** when having a **non-empty *Optional***.

* + orElseGet(<lambda expression returning the internal-value>):

Optional<Person> personResult = findPerson(name);  
Person person = personResult.orElseGet(()-> Person.GUEST);

With orElseGet() - the S*upplier*method passed as an argument is **only executed** when *an Optional*value **is not present**.

An Optional type can be a return type for most methods except some scenarios discussed later in the tutorial.

Most of the time, returning an Optional is just fine:

|  |
| --- |
| public static Optional<User> findUserByName(String name) {      User user = usersByName.get(name);      Optional<User> opt = Optional.ofNullable(user);      return opt;  } |

This is handy since we can use the Optional API in the calling method:

|  |  |
| --- | --- |
|  | public static void changeUserName(String oldFirstName, String newFirstName) {      findUserByFirstName(oldFirstName).ifPresent(user -> user.setFirstName(newFirstName));  } |

It's also appropriate for a static method or utility method to return an Optional value.  However, there are many situations where we should not return an Optional type.

.map – if not null – apply the mapped function to it. Otherwise, return null.

# JAR

When the code is ready, you can package your application in a Java archive (JAR) so that you can share it with other developers. A built Java archive is called an artifact.

# Eclipse

## External Libraries

To add references to external libraries:

* Right-click the project, click **Properties**.
* Choose **Java Build Path**.
* Click the **Libraries** tab
* Click **Add External JARs...**
* Browse to find the required \*.jar file and select it.
* Click **OK** to update the build path.

# Java Web Apps

* Most Java web-apps are made up of **Servlets** and **Java Server Pages**.
* Many frameworks are also built on top of this architecture. For example, Struts framework.
* Servlets includes also **Filters** and **Listeners**.
* Many frameworks uses Filters as Controllers in MVC.
* Standard Java web-application structure:
  + WEB-INF folder contains:
    - web.xml file – the web app configuration
    - classes and libs that make up the application
    - JSP – Java Server Pages folder. Jsp files can also live in the root of the application.
  + **Deployment:** Java web-apps can be deployed by copying files or by zipping the application directory structure into a .war file.

## TomCat Server

used to be the reference implementation for the Java Apps server. To start/stop the server – see startup.bat and shutdown.bat in its binary installation folder.

* When installed: user and password on my PC: tomcat
* Its default home page after starting: <http://localhost:8080>
* Tomcat Web Application Manager:

GUI:  
<http://localhost:8080/manager/html>

Text:  
<http://localhost:8080/manager/text>

It allows managing the applications through the gui or in script. To add users for using the app-manager see Tomcat/conf/tomcat-users.xml

* To deploy the app from maven, add tomcat as a plugin to the maven’s pom.xml file.
* Set up TomCat in Eclipse:  
  Window->Show View->Servers. Add the Tomcat web server and add its root directory.
  + We can now find all the config files for TomCat under the ProjectExplorer->Servers folder.
  + We can configure the server by double-clicking it in the Servers window.  
    Make sure that ‘Server Location’ is set to ‘Use Tomcat installation’

## Hello World App

1. Project->New->Other->Web->Dynamic Web Project
2. Click on ‘Generate web.xml deployment descriptor’ and press finish.
3. To create a new Servlet:
   1. Right click on the project->New Servlet
   2. Fill in a meaningful Java Package name
   3. Program the Servlet
4. To deploy automatically:
   1. Run->Run As-> Run on Server. Choose the server to use (TomCat that you installed)
   2. This will automatically open the internal web-browser. To use the system web-browser instead: Window->Web Browser-> choose the web browser you want.

## Servlets

* Implement the Servlet interface. Main functions in this interface:
  + service – handles web requests. They are usually http requests but can also use FTP and other protocols.
* Extends one of the following generic classes:
  + GenericServlet – no protocol specific implementations in this class. Can only handle simple requests and can’t handle Session data.
  + HttpServlet – the base class we’ll usually use for creating our own servlets. It supports all the HTTP operations (get,post etc).

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebServlet;

**import** javax.servlet.http.\*;

**import** java.io.IOException;

// map the path to the servlet (localhost/myApp):

@WebServlet("/myApp")

**public** **class** MyFirstServlet **extends** HttpServlet {

**public** **void** init(){

// Perform any instance initialization

}

@Override

**protected** **void** doGet(HttpServletRequest req,

HttpServletResponse resp)

**throws** ServletException, IOException {

resp.getWriter().write("Hello World");

}

@Override

**protected** **void** doPost(HttpServletRequest req,

HttpServletResponse resp){

…

}

}

HttpServletRequest variable methods:

* getParameter – return the value of the requested parameter (as a string), or null. When we know the parameter(s) names in advance.
* getParameterValues – get the parameter name (string) and returns an array with all the parameter values or null
* getParameterNames() – return an enumeration of string objects containing the names of the parameters. When we don’t know the parameters names or not all of them are mandatory.
* getParameterMap() – returns a java.util.Map – when we don’t know the parameter names and they can hold more than one value each.

HttpServletResponse methods:

* response.setContectType(“text/html”);
* PrintWriter out = response.getWriter();

# Style

 **ClassNames** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.  
Example: *class* ***MyFirstJavaClass***

 **methodNames** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.  
Example: ***public void myMethodName()***

 **ProgramFileName** − Name of the program file should **exactly match the class name**. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).  
Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'****MyFirstJavaProgram.java****'*

 **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

## Static Analysis

* SonarQube - highly recommended for Java.
* SonarLint - plugin for intelliJ - based on SonarQube but doesn't require so much effort setting it up.
* Awesome static analysis – a list of tools for different languages

## Coding Convention

Pyramid Java Conventions

# Libraries

## Guice

<https://www.baeldung.com/guice>

<https://www.baeldung.com/guice-spring-dependency-injection>

<https://justin.abrah.ms/misc/an-overview-of-guice-java-dependency-injection.html>

A dependency injection framework (from google)

* Uses a special class called a module to manage the dependencies. This class inherits from AbstractModule().
* With @Inject:
  + All the data members to be injected are marked with @Inject:

public class GuiceUserService {

@Inject

private AccountService accountService;

}

* And included in the module’s configure() function:

public class GuiceModule extends AbstractModule {

@Override

protected void configure() {

bind(AccountService.class).to(AccountServiceImpl.class);

}

}

Normally, we expect Guice to instantiate each dependency object from their default constructors if there isn't any binding defined explicitly in the *configure()* method. **But since interfaces can't be instantiated directly, we need to define bindings** to tell Guice which interface will be paired with which implementation.

* Using @Provides:
  + Define the function that return an object from the specified type using the @Provides annotation:

@Provides

public BookService bookServiceGenerator() {

return new BookServiceImpl();

}

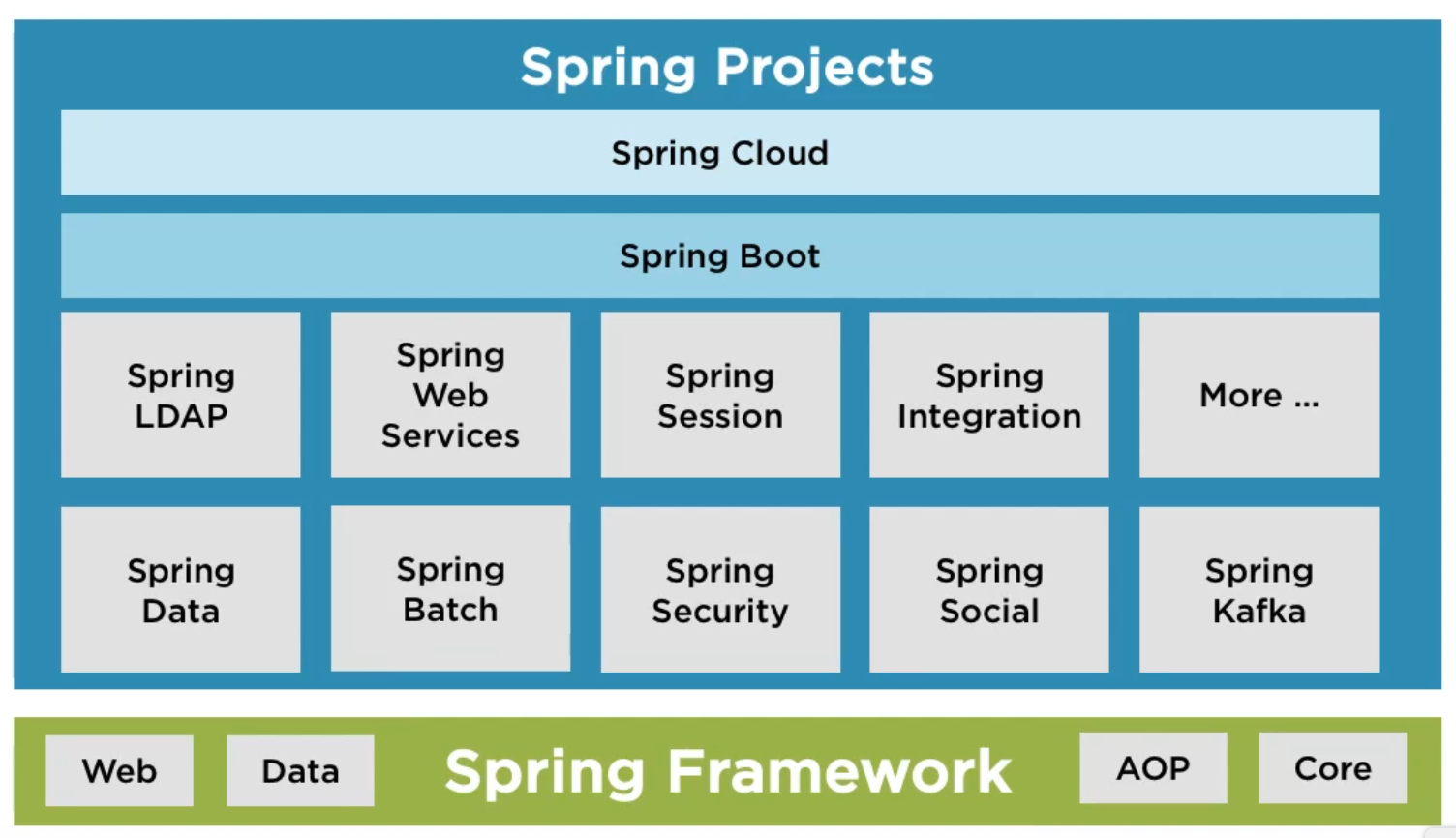
* Usually we will use @Inject for our own internal code and @Provides for external libraries.
* Then, we need to define an *Injector* using *GuiceModule* to get instances of our classes. Let's just note that all of our Guice tests will use this *Injector*:  
  Injector injector = Guice.**createInjector**(new GuiceModule());
* Finally, at runtime we retrieve a *GuiceUserService* instance with a non-null *accountService* dependency:

GuiceUserService guiceUserService = **injector.getInstance**(GuiceUserService.class);

assertNotNull(guiceUserService.getAccountService());

## Spring

Dependency injection framework



The Spring eco system includes:

* Spring Framework – reducing boiler-plate code. Used to simplify creating web and data-based applications.
* Spring Boot – a fast way to build and deploy spring-based applications
  + Auto-configuration – attempts to automatically configure your Spring application based on the … dependencies that you have added. For example, auto configure the type of database required by the application (e.g. oracle etc) based on the dependency in the code. Setting up auto-configuration is super easy:  
    @EnableAutoConfiguration  
    public class DemoApplication {…}
  + Standalone – you don’t need to configure, deploy web server etc. You can just:
    - Package the app
    - Run the app:  
      java -jar my-application.jar
  + Opinionated – it has defined defaults on how to do things.
* Spring Cloud – simplify the creation of distributed applications

### The big picture

* Beans – the different objects that are managed by Spring framework
* Autowiring – the process in which Spring recognize the dependencies and initializes and inject the correct objects to the managed componenets.
* IOC (Invertion of Control) Container
* Application Context – where all the core logic of Spring framework

@Component – tells spring that it needs to manage a class. Meaning, it needs to initialize its objects

@Autowired – tells spring that it needs to create a new object of this type (search for this type with @Componenet), initialize it and inject it into the containing class

**@Component**  
public class SpringComponentBasic {  
   
 **@Autowired**  
 SortAlgorithm sortAlgorithm;  
}  
  
  
**@Component**  
public class BubbleSortAlgorithm implements SortAlgorithm {  
 ...  
}

### Spring Project

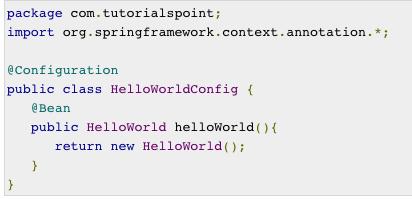
<https://start.spring.io/>

### Spring @Configuration

* Doc: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html>
* Tutorial: <https://www.tutorialspoint.com/spring/spring_java_based_configuration.htm>

Dependency injection using Spring jave-based annotations. The @Configuration is a marker annotation which indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime

* @Configuration (class annotation) – Indicates that a class declares one or more [@Bean](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Bean.html) methods and may be processed by the Spring IoC container to generate bean definitions and service requests for those beans at runtime
* @Bean (for methos in @Configuration class) – tells spring that the method will return an object that should be registered as a ben in the Spring application context
  + @Beans can call each other



The method name of the @Bean (helloWorld) is the bean ID.

* @Import (on @Configuration class) – indicate that this class depends on another @Configuration class so that its Beans can be used by the importing class.   
  Note: for every such hierarchy of @Configuration classes, only the root class needs to be passed to the Bootstrapping method (e.g. AnnotationConfigApplicationContext).
* @profile({profileName}) – the class/Bean will be processed only if the given profile(s) is active:  
  @Profile(“development”,….)  
  @Configuration  
  public class….
* @Lazy – will mark the class/Bean for lazy initialization (greedy by default).

#### Bootstrapping

* AnnotationConfigApplicationContext – used to load and provide your configuration classes to Spring container:



Example of multiple config classes:



* Spring @Configuration classes can also be bootstrapped using <beans> XML or via component scanning. Note that @Configuration is meta-annotated with @Componenet so if componenet-scan is initialized (through XML configuration or through @ComponentScan), they will be bootstrapped automatically on startup.

### Unit Test

* @Mock – create mock-object of this class
* @InjectMocks – will create a real object of the class (the class under test that will be actually called), and create mocks for all of this class’s dependencies to be able to mock them.

@Mock  
private DependencyOfMyClass dependencyOfMyClass;  
  
@InjectMocks  
MyClass myClass;

* If you need to mock dependencies of dependencies:
  + It’s a code smell – meaning that your dependencies are too hard to test in isolation.
  + If you still have to do this:
    - If UnitUnderTestClass – is the class I’m testing (should use the real class and inject all mocked dependencies) , which depends on
    - MyClass which I don’t want to mock (I want the real implementation) which depends on
    - DependencyOfMyClass which I need to mock:

@Mock  
private DependencyOfMyClass dependencyOfMyClass;  
  
@Spy  
MyClass myClass;

@InjectMocks  
UnitUnderTestClass unitUnderTestClass;

## Lombok

Auto-generate boilerplate code for you using annotations:

* Per class:
  + @Data – create a data class with setters, getters, etc.
  + @ToString – implement toString() method to stringify all non-static fields.
  + @EqualsAndHashCode – generate equals() and hashCode() for all non-static fields.
* Per field:
  + @NonNull – will check for constructor/setters if null – return NullPointerException
  + @Setter / @Getter
  + @Cleanup
  + @Builder.Default – allow setting a variable’s default so that if it’s not included in the Builder-constructor, it will default to this value instead of its type’s deafault. For Example:  
    *@Builder.Default  
    private boolean myFlag = true;*

Will default to true (instead of false) if it’s not included in the builder’s call.

* Per method:
  + @Synchronized
* More:
  + @SneakyThrows

## JSP (JavaServer Pages)

Java decorated html.

* Insert java code: <%this.isJavaCode()%>

# Patterns

* Service Provider Framework. For example: java.util.ServiceLoader
* Bridge Pattern

# Testing

## Junit

To add unit tests to your project:

* + - 1. Create a ‘tst’ directory under the main project (parallel to ‘src’)
      2. Mark it as ‘Test Sources Root’ in the IDE
      3. To add a test class to a class:  
         In the class’s declaration line, press option+Enter and choose ‘create test’

### Spy

* Mockito doesn’t support mocking static classes. For this you can use PowerMockito
* Mockito/PowerMockito doesn’t support spying on void functions.
* I had trouble using spies with Optional types. I didn’t investigate it too much but just removed the spies to fix it.

## TestNG Framework

For all types of tests: all the way from UT to E2E and integrations

|  |  |
| --- | --- |
| package example1; | |
|  |

|  |  |
| --- | --- |
| import org.testng.annotations.\*; | |
|  |

|  |  |
| --- | --- |
| public class SimpleTest { | |
|  |

|  |
| --- |
| @BeforeClass |
| public void setUp() { | |

|  |  |
| --- | --- |
| // code that will be invoked when this test is instantiated | |
| } |

|  |
| --- |
|  |
| @Test(groups = { "fast" }) | |

|  |
| --- |
| public void aFastTest() { |
| System.out.println("Fast test"); | |

|  |  |
| --- | --- |
| } | |
|  |

|  |  |
| --- | --- |
| @Test(groups = { "slow" }) | |
| public void aSlowTest() { |

|  |  |
| --- | --- |
| System.out.println("Slow test"); | |
| } |

|  |
| --- |
|  |
| } | |

# Tools

# IntelliJ

## Setting IntelliJ colours:

* instance field, Function call, Constant - 008E8E , bold
* Static field, method -
* Instance method - 007194
* parantesis, Keyword - black , bold
* Language Defaults (class, - 485B8C , bold
* Function declaration - A30100