List<String> result = lines.stream().filter(line -> !"mkyong".equals(line)).collect(Collectors.toList());

Map<String, Long> counting = items.stream().collect(Collectors.groupingBy(Item::getName, Collectors.counting()));

Map<String, Integer> sum = items.stream().collect(Collectors.groupingBy(Item::getName, Collectors.summingInt(Item::getQty)));

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

Abstraction -190 Page

Encapsulation

Polymorphism

Inheritance

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

Overloading - Method Resolution taken care by compiler

Overriding - method Resolution taken care by jvm(Runtime)

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

Method hiding - Parent class have Static method and inherting child class also have same static method (Same method sig and same return type)

this is called method hiding.

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

There are several differences between HashMap and Hashtable in Java:

Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications,

as unsynchronized Objects typically perform better than synchronized ones.

Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.

One of HashMap's subclasses is LinkedHashMap,

so in the event that you'd want predictable iteration order (which is insertion order by default),

you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

Since synchronization is not an issue for you, I'd recommend HashMap.

If synchronization becomes an issue, you may also look at ConcurrentHashMap.

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

Arrays can hold only primitives where as collections can hold both primitive and objects.

Arrays are best according to Performance and worst according to memory where as collections are reverse.

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

concurrent modification

This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.

For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it.

In general, the results of the iteration are undefined under these circumstances.

Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE)

may choose to throw this exception if this behavior is detected.

Iterators that do this are known as fail-fast iterators, as they fail quickly and cleanly,

rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that this exception does not always indicate that an object has been concurrently modified by a different thread.

If a single thread issues a sequence of method invocations that violates the contract of an object,

the object may throw this exception. For example,if a thread modifies a collection directly while it is iterating over the

collection with a fail-fast iterator, the iterator will throw this exception.

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

In every collection class toString method is overridden to print desired format while used in sysout

ArrayList is more versatile than vector because we can get synchronized list or read-only list from arraylist easily using Collections utility class.

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

Java 1.5 Concurrent package (java.util.concurrent) contains thread-safe collection classes that allow collections

to be modified while iterating.

By design Iterator implementation in java.util packages are fail-fast and throws ConcurrentModificationException.

But Iterator implementation in java.util.concurrent packages are fail-safe and we can modify the collection while iterating.

Some of these classes are CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet.

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

JAVA 8 Features

Lambda Expression: This allows you to pass an anonymous function as object.

Stream API: Allows you to take advantage of multiple cores of modern CPU and lets you write concise code.

Date and Time API: There is a solid and easy to use date and time library in JDK.

Extension Methods: You can include static and default method into your interface.

Repeated Annotation: This lets you apply the same annotation multiple times on a type.

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

throw keyword / throws keyword

1)throw is used to explicitly throw an exception. throws is used to declare an exception.

2)checked exceptions can not be propagated with throw only. checked exception can be propagated with throws.

3)throw is followed by an instance. throws is followed by class.

4)throw is used within the method. throws is used with the method signature.

5)You cannot throw multiple exception You can declare multiple exception e.g. public void method()throws IOException,SQLException.

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

Singleton class means that any given time only one instance of the class is present, in one JVM

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

Since salary ( In Employee Object )is a transient variable, it’s value was not saved to file and hence not retrieved in the new object.

Similarly static variable values are also not serialized since they belongs to class and not object.

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

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.

Since load() throws exception when data is not found, we should use it only when we know data exists.

We should use get() when we want to make sure data exists in the database.

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

Zipkin as spring boot app will not work for spring boot(2.X versions)

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

For Spring Config server for communicating with Remote GIT need to disable HTTP SSL

Using RUN git config --global http.sslVerify false

**-------------------------------------------------------------------------**-------

Docker related Commands

Docker Network

docker network create networkName - to create docker network

docker network ls - network list

docker network connect networkname containername -> to connect container to docker

docker network inspect networkname - > to check what all containers are connect

docker container ps -a

docker images

docker network ls

docker rm spring-config-server spring-config-client spring-slueth-app

docker network rm spring-config-server\_configconnect

docker run -p 8001:8001 --name spring-config-client --link spring-config-server spring-config-client

docker pull splunk/splunk

docker run -d -e "SPLUNK\_START\_ARGS=--accept-license" -e "SPLUNK\_USER=root" -p "8000:8000" splunk/splunk

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

Stateless

Statelessness means that every HTTP request happens in complete isolation.

When the client makes an HTTP request, it includes all information necessary for the server to fulfill that request.

The server never relies on information from previous requests.

If that information was important, the client would have sent it again in this request.

Statelessness also brings new features. It’s easier to distribute a stateless application across load-balanced servers.

A stateless application is also easy to cache.

There are actually two kinds of state. Application State that lives on the client and Resource State that lives on the server.

A web service only needs to care about your application state when you’re actually making a request

The rest of the time, it doesn’t even know you exist.

This means that whenever a client makes a request, it must include all the application states the server will need to process it.

Resource state is the same for every client, and its proper place is on the server.

When you upload a picture to a server, you create a new resource: the new picture has its own URI and can be the target of future requests.

You can fetch, modify, and delete this resource through HTTP.

Hope this helps differentiate what statelessness and various states mean.

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

A Functional Interface is an interface, which contains one and only one abstract method. Functional Interface is also know as SAM Interface because it contains only one abstract method.

SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

We need to follow these rules to define a Functional Interface:

Define an interface with one and only one abstract method.

We cannot define more than one abstract method.

Use @FunctionalInterface annotation in interface definition.

We can define any number of other methods like Default methods, Static methods.

If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

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

https://stackoverflow.com/questions/23862994/what-is-the-difference-between-hibernate-and-spring-data-jpa

Hibernate is a JPA implementation, while Spring Data JPA is a JPA Data Access Abstraction. Spring Data offers a solution to GenericDao custom implementations. It can also generate JPA queries on your behalf through method name conventions.

With Spring Data, you may use Hibernate, Eclipse Link or any other JPA provider. A very interesting benefit is that you can control transaction boundaries declaratively using the @Transactional annotation.

Spring JDBC is much more lightweight, and it's intended for native querying, and if you only intend to use JDBC alone, then you are better off using Spring JDBC to deal with the JDBC verbosity.

So, Hibernate,and Spring Data are complementary rather than competitors.

There are 3 different things we are using here :

JPA : Java persistence api which provide specification for persisting, reading, managing data from your java object to relations in database.

Hibernate: There are various provider which implement jpa. Hibernate is one of them. So we have other provider as well. But if using jpa with spring it allows you to switch to different providers in future.

Spring Data JPA : This is another layer on top of jpa which spring provide to make your life easy.

So lets understand how spring data jpa and spring + hibernate works-

Spring Data JPA:

Let's say you are using spring + hibernate for your application.

Now you need to have dao interface and implementation where you will be writing crud operation using SessionFactory of hibernate. Let say you are writing dao class for Employee class, tomorrow in your application you might need to write similiar crud operation for any other entity. So there is lot of boilerplate code we can see here.

Now Spring data jpa allow us to define dao interfaces by extending its repositories(crudrepository, jparepository)

so it provide you dao implementation at runtime. You don't need to write dao implementation anymore.Thats how spring data jpa makes your life easy.

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

https://dzone.com/articles/testing-databases-junit-and

https://dzone.com/articles/integrate-h2-database-in-your-spring-boot-applicat

Joincolumn and mapped by

https://stackoverflow.com/questions/11938253/whats-the-difference-between-joincolumn-and-mappedby-when-using-a-jpa-onetoma

@jsonignoreproperties

https://fasterxml.github.io/jackson-annotations/javadoc/2.6/com/fasterxml/jackson/annotation/JsonIgnoreProperties.html

@JsonIgnoreProperties({"hibernateLazyInitializer","handler"}) -> Jackson unable to serialize some data which is created by hibernate lazy issue so for ignoring that use this

http://www.greggbolinger.com/ignoring-hibernate-garbage-via-jsonignoreproperties/

spring.jpa.properties.hibernate.enable\_lazy\_load\_no\_trans

https://stackoverflow.com/questions/25362831/solve-hibernate-lazy-init-issue-with-hibernate-enable-lazy-load-no-trans

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

Difference between MocitoJunitRunner and SpringRunner

https://stackoverflow.com/questions/49635396/runwithspringrunner-class-vs-runwithmockitojunitrunner-class

The SpringRunner provides support for loading a Spring ApplicationContext and having beans @Autowired into your test instance. It actually does a whole lot more than that (covered in the Spring Reference Manual), but that's the basic idea.

Whereas, the MockitoJUnitRunner provides support for creating mocks and spies with Mockito.

However, with JUnit 4, you can only use one Runner at a time.

Thus, if you want to use support from Spring and Mockito simultaneously, you can only pick one of those runners.

But you're in luck since both Spring and Mockito provide rules in addition to runners.

For example, you can use the Spring runner with the Mockito rule as follows.

@RunWith(SpringRunner.class)

@SpringBootTest

public class MyTests {

@Rule

public MockitoRule rule = MockitoJUnit.rule();

@Mock

MyService myService;

// ...

}

Though, typically, if you're using Spring Boot and need to mock a bean from the Spring ApplicationContext

you would then use Spring Boot's @MockBean support instead of simply @Mock.

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

Difference between doReturn/when

https://stackoverflow.com/questions/20353846/mockito-difference-between-doreturn-and-when

The two syntaxes for stubbing are roughly equivalent. However, you can always use doReturn/when for stubbing; but there are cases where you can't use when/thenReturn. Stubbing void methods is one such. Others include use with Mockito spies, and stubbing the same method more than once.

One thing that when/thenReturn gives you, that doReturn/when doesn't, is type-checking of the value that you're returning, at compile time. However, I believe this is of almost no value - if you've got the type wrong, you'll find out as soon as you run your test.

I strongly recommend only using doReturn/when. There is no point in learning two syntaxes when one will do.

You may wish to refer to my answer at Forming Mockito "grammars" - a more detailed answer to a very closely related question.

--

Both approaches behave differently if you use a spied object (annotated with @Spy) instead of a mock (annotated with @Mock):

when(...) thenReturn(...) makes a real method call just before the specified value will be returned. So if the called method throws an Exception you have to deal with it / mock it etc. Of course you still get your result (what you define in thenReturn(...))

doReturn(...) when(...) does not call the method at all.

Example:

public class MyClass {

protected String methodToBeTested() {

return anotherMethodInClass();

}

protected String anotherMethodInClass() {

throw new NullPointerException();

}

}

Test:

@Spy

private MyClass myClass;

// ...

// would work fine

doReturn("test").when(myClass).anotherMethodInClass();

// would throw a NullPointerException

when(myClass.anotherMethodInClass()).thenReturn("test");

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

**Variable hiding happens when we declare a property in a local scope that has the same name as the one we already have in the outer scope.**

Before jumping to the examples, let’s briefly recap the possible variable scopes in Java. We can define them with the following categories:

* local variables – declared in a piece of code such as methods, constructors, in any block of code with curly braces
* instance variables – defined inside of a class and belong to the instance of the object
* class or *static* variables – are declared in the class with the *static* keyword. They have a class level scope.

Now, let’s describe the hiding with examples, for each individual category of variables.

**2.1. The Power of Local**

Let’s have a look at the *HideVariable* class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class HideVariable {        private String message = "this is instance variable";        HideVariable() {          String message = "constructor local variable";          System.out.println(message);      }        public void printLocalVariable() {          String message = "method local variable";          System.out.println(message);      }        public void printInstanceVariable() {          String message = "method local variable";          System.out.println(this.message);      }  } |

Here we have the *message*variable declared in 4 different places. The local variables declared inside of the constructor and the two methods are shadowing the instance variable.

Let’s test the initialization of an object and calling the methods:

|  |  |
| --- | --- |
| 1  2  3  4 | HideVariable variable = new HideVariable();  variable.printLocalVariable();    variable.printInstanceVariable(); |

The output of the code above is:

|  |  |
| --- | --- |
| 1  2  3 | constructor local variable  method local variable  this is instance variable |

Here, the first 2 calls are retrieving the local variables.

To access the instance variable from the local scope, we can use *this* keyword like it is shown in *printInstanceVariable()* method.

**2.2. Hiding and The Hierarchy**

Similarly, when both the child and the parent classes have a variable with the same name, the child’s variable hides the one from the parent.

Let’s suppose we have the parent class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class ParentVariable {        String instanceVariable = "parent variable";        public void printInstanceVariable() {          System.out.println(instanceVariable);      }  } |

After that we define a child class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class ChildVariable extends ParentVariable {        String instanceVariable = "child variable";        public void printInstanceVariable() {          System.out.println(instanceVariable);      }  } |

To test it, let’s initialize two instances. One with parent class and another with the child, then invoke the *printInstanceVariable()* methods on each of them:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ParentVariable parentVariable = new ParentVariable();  ParentVariable childVariable = new ChildVariable();    parentVariable.printInstanceVariable();  childVariable.printInstanceVariable(); |

The output shows the property hiding:

|  |  |
| --- | --- |
| 1  2 | parent variable  child variable |

**In most cases, we should avoid creating variables with the same name both in parent and child classes**. Instead, we should use a proper access modifier like *private*and provide getter/setter methods for that purpose.

**3. Method Hiding**

Method hiding may happen in any hierarchy structure in java. When a child class defines a static method with the same signature as a static method in the parent class, then the child’s method *hides* the one in the parent class. To learn more about the *static* keyword,  [this write-up is a good place to start.](https://www.baeldung.com/spring-bean-scopes)

The same behavior involving the instance methods is called method overriding. To learn more about method overriding checkout our [guide here](https://www.baeldung.com/java-method-overload-override).

Now, let’s have a look at this practical example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class BaseMethodClass {        public static void printMessage() {          System.out.println("base static method");      }  } |

*BaseMethodClass* has a single *printMessage() static* method.

Next, let’s create a child class with the same signature as in the base class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class ChildMethodClass extends BaseMethodClass {        public static void printMessage() {          System.out.println("child static method");      }  } |

Here’s how it works:

|  |  |
| --- | --- |
| 1 | ChildMethodClass.printMessage(); |

The output after calling the *printMessage()* method:

|  |  |
| --- | --- |
| 1 | child static method |

The *ChildMethodClass.printMessage()*hides the method in *BaseMethodClass*.

**3.1. Method Hiding vs Overriding**

Hiding doesn’t work like overriding, because static methods are not polymorphic. Overriding occurs only with instance methods. It supports late binding, so which method will be called is determined at runtime.

**On the other hand, method hiding works with static ones. Therefore it’s determined at compile time**.

# [What is perm space?](https://stackoverflow.com/questions/1279449/what-is-perm-space)

Perm Gen stands for permanent generation which holds the meta-data information about the classes.

1. Suppose if you create a class name A, it's instance variable will be stored in heap memory and class A along with static classloaders will be stored in permanent generation.
2. Garbage collectors will find it difficult to clear or free the memory space stored in permanent generation memory. Hence it is always recommended to keep the permgen memory settings to the advisable limit.
3. JAVA8 has introduced the concept called meta-space generation, hence permgen is no longer needed when you use jdk 1.8 versions

The change has been done on java7 itself that the *constant pool has been moved to typical heap space from permgen space(still the pool behaviour is same)* where as permgen space completely removed in Java 8. Metaspace is nothing to do with constant pool specially, it is generic for all objects.

The *String* object is the most used class in the Java language.

In this quick article, we’ll explore the Java String Pool — **the special memory region where *Strings* are stored by the JVM**.

**2. String Interning**

Thanks to the immutability of *Strings* in Java, the JVM can optimize the amount of memory allocated for them by **storing only one copy of each literal *String* in the pool**. This process is called *interning*.

When we create a*String* variable and assign a value to it, the JVM searches the pool for a *String* of equal value.

**If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.**

If not found, it’ll be added to the pool (interned) and its reference will be returned.

Let’s write a small test to verify this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | String constantString1 = "Baeldung";  String constantString2 = "Baeldung";    assertThat(constantString1)    .isSameAs(constantString2); |

**3. *Strings* Allocated using the Constructor**

When we create a *String* via the *new* operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

Every *String* created like this will point to a different memory region with its own address.

Let’s see how this is different from the previous case:

|  |  |
| --- | --- |
| 1  2  3  4 | String constantString = "Baeldung";  String newString = new String("Baeldung");    assertThat(constantString).isNotSameAs(newString); |

**4. *String* Literal vs *String Object***

**When we create a *String* object using the *new()* operator, it always creates a new object in heap memory. On the other hand, if we create an object using *String* literal syntax e.g. “Baeldung”, it may return an existing object from the String pool, if it already exists.**Otherwise, it will create a new String object and put in the string pool for future re-use.

At a high level, both are the *String* objects, but the main difference comes from the point that *new()* operator always creates a new *String* object. Also, when we create a *String* using literal – it is interned.

This will be much more clear when we compare two *String* objects created using *String*literal and the *new* operator:

|  |  |
| --- | --- |
| 1  2  3 | String first = "Baeldung";  String second = "Baeldung";  System.out.println(first == second); // True |

In this example, the *String* objects will have the same reference.

Next, let’s create two different objects using *new* and check that they have different references:

|  |  |
| --- | --- |
| 1  2  3 | String third = new String("Baeldung");  String fourth = new String("Baeldung");  System.out.println(third == fourth); // False |

Similarly, when we compare a *String* literal with a *String* object created using *new()* operator using the == operator, it will return *false:*

|  |  |
| --- | --- |
| 1  2  3 | String fifth = "Baeldung";  String sixth = new String("Baeldung");  System.out.println(fifth == sixth); // False |

In general, **we should use the *String* literal notation when possible**. It is easier to read and it gives the compiler a chance to optimize our code.

**5. Manual Interning**

We can manually intern a *String* in the Java String Pool by calling the *intern()* method on the object we want to intern.

Manually interning the *String* will store its reference in the pool, and the JVM will return this reference when needed.

Let’s create a test case for this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | String constantString = "interned Baeldung";  String newString = new String("interned Baeldung");    assertThat(constantString).isNotSameAs(newString);    String internedString = newString.intern();    assertThat(constantString)    .isSameAs(internedString); |
|  |  |

**6. Garbage Collection**

Before Java 7, the JVM **placed the Java String Pool in the *PermGen* space, which has a fixed size — it can’t be expanded at runtime and is not eligible for garbage collection**.

The risk of interning *Strings* in the *PermGen*(instead of the *Heap*) is that **we can get an *OutOfMemory* error** from the JVM if we intern too many *Strings*.

From Java 7 onwards, the Java String Pool is **stored in the *Heap* space, which is garbage collected** by the JVM*.*The advantage of this approach is the **reduced risk of *OutOfMemory*error** because unreferenced *Strings* will be removed from the pool, thereby releasing memory.

**7. Performance and Optimizations**

In Java 6, the only optimization we can perform is increasing the *PermGen* space during the program invocation with the *MaxPermSize* JVM option:

|  |  |
| --- | --- |
| 1 | -XX:MaxPermSize=1G |

In Java 7, we have more detailed options to examine and expand/reduce the pool size. Let’s see the two options for viewing the pool size:

|  |  |
| --- | --- |
| 1 | -XX:+PrintFlagsFinal |
| 1 | -XX:+PrintStringTableStatistics |

The default pool size is 1009. If we want to increase the pool size, we can use the *StringTableSize* JVM option:

|  |  |
| --- | --- |
| 1 | -XX:StringTableSize=4901 |

**Note that increasing the pool size will consume more memory but has the advantage of reducing the time required to insert the *Strings* into the table.**

**8. A Note About Java 9**

Until Java 8, *Strings* were internally represented as an array of characters – *char[]*, encoded in *UTF-16*, so that every character uses two bytes of memory.

With Java 9 a new representation is provided, called *Compact Strings.*This new format will choose the appropriate encoding between *char[]* and *byte[]* depending on the stored content.

Since the new *String* representation will use the *UTF-16* encoding only when necessary, the amount of *heap*memory will be significantly lower, which in turn causes less *Garbage Collector* overhead on the *JVM.*

A lambda expression consists of two parts: the parameter part and the expressions part separated by a forward arrow as below:

|  |  |
| --- | --- |
| 1 | params -> expressions |

Any lambda expression has the following characteristics:

* **Optional type declaration** – when declaring the parameters on the left-hand side of the lambda, we don’t need to declare their types as the compiler can infer them from their values. So *int param -> …* and *param ->…* are all valid
* **Optional parentheses** – when only a single parameter is declared, we don’t need to place it in parentheses. This means *param -> …* and *(param) -> …* are all valid. But when more than one parameter is declared, parentheses are required
* **Optional curly braces** – when the expressions part only has a single statement, there is no need for curly braces. This means that *param – > statement* and *param – > {statement;}* are all valid. But curly braces are required when there is more than one statement
* **Optional return statement** – when the expression returns a value and it is wrapped inside curly braces, then we don’t need a return statement. That means *(a, b) – > {return a+b;}* and *(a, b) – > {a+b;}* are both valid