1.What are enhancemets in java8 for memory management

Oracle has completely gotten rid of ‘PermGen’ and replaced it with Metaspace.

**What is PermGen ?**

Short form for Permanent Generation, PermGen is the memory area in Heap that is used by the JVM to **store class and method objects**. If your application loads lots of classes, PermGen utilization will be high. PermGen also **holds ‘interned’ Strings**

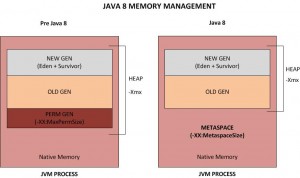
The size of the PermGen space is configured by the Java command line option**-XX:MaxPermSize**

Typically 256 MB should be more than enough of PermGen space for most of the applications

However, It is not unusal to see the error “**java.lang.OutOfMemoryError: PermGen space“** if you are loading unusual number of classes.

Gone are the days of OutOfMemory Errors due to PermGen space. **With Java 8, there is NO PermGen**. That’s right. So no more OutOfMemory Errors due to PermGen

The key difference between PermGen and Metaspace is this: while PermGen is part of Java Heap (Maximum size configured by -Xmx option), **Metaspace is NOT part of Heap.** Rather Metaspace is part of **Native Memory (process memory)** which is only limited by the Host Operating System.

[](http://karunsubramanian.com/wp-content/uploads/2014/07/Java8-heap.jpg)

**So, what is the significance of this change?**

While you will NOT run out of PermGen space anymore (since there is NO PermGen), you may consume excessive Native memory making the total process size large. The issue is, if your application loads lots of classes (and/or interned strings), **you may actually bring down the Entire Server** (not just your application). Why ? Because the native memory is only limited by the Operating System. This means you can literally take up all the memory on the Server. Not good.

It is critical that you add the new option **-XX:MaxMetaspaceSize**  which sets the Maximum Metaspace size for your application.

Note that it is no longer sufficient to just monitor the Heap Size. You must also monitor the Metaspace which you can do by just **keeping an eye on the ‘process size’** using your Operating System utilities (Example: ‘top’ in Unix/Linux, ‘Task Manager’ in Windows).

**Bonus Tip:**

You can use the **jmap**command to print out Memory statistics of your current pre Java 8 application.

**jmap -permstat <PID>**

There you have it. With Java 8, PermGen is gone and Metaspace is in. Metaspace is part of Native Memory and NOT part of Java Heap. While this change may not be significant during development stage of the application, it is critical to consider this when going to production as you might not only bring down your application but bring down the entire server if your application eats up excessive Metaspace.

## sort the employees based on names using java8

List<Employee> employees  = getEmployeesFromDB();

//Sort all employees by first name

employees.sort(Comparator.comparing(e -> e.getFirstName()));

//OR you can use below

employees.sort(Comparator.comparing(Employee::getFirstName));

//Sort all employees by first name in reverse order

Comparator<Employee> comparator = Comparator.comparing(e -> e.getFirstName());

employees.sort(comparator.reversed());

//Sorting on multiple fields; Group by.

Comparator<Employee> groupByComparator = Comparator.comparing(Employee::getFirstName).thenComparing(Employee::getLastName);

employees.sort(groupByComparator);

## Undo/redo operation is an application of which data structure

Stack data structure is most suitable to implement redo-undo feature. This is because the stack is implemented with LIFO(last in first out) order which is equivalent to redo-undo feature i.e. the last re-do is undo first.

## frequent used annotations in spring boot

<https://springframework.guru/spring-framework-annotations/>

<https://dzone.com/articles/frequently-used-annotations-in-spring-boot-applica>

@Component

public class BeanA {

@Autowired

@Qualifier("beanB2")

private BeanInterface dependency;

...

}

With the @Qualifier annotation added, Spring will now know which bean to autowire where beanB2 is the name of BeanB2.

Spring Framework 5Learn the Spring Framework with my Spring Framework 5 – Beginner to Guru Course!

@Configuration

This annotation is used on classes which define beans. @Configuration is an analog for XML configuration file – it is configuration using Java class. Java class annotated with @Configuration is a configuration by itself and will have methods to instantiate and configure the dependencies.

Here is an example:

@Configuration

public class DataConfig{

@Bean

public DataSource source(){

DataSource source = new OracleDataSource();

source.setURL();

source.setUser();

return source;

}

@Bean

public PlatformTransactionManager manager(){

PlatformTransactionManager manager = new BasicDataSourceTransactionManager();

manager.setDataSource(source());

return manager;

}

}

@ComponentScan

This annotation is used with @Configuration annotation to allow Spring to know the packages to scan for annotated components. @ComponentScan is also used to specify base packages using basePackageClasses or basePackage attributes to scan. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation.

Checkout this post for an in depth look at the Component Scan annotation.

@Bean

This annotation is used at the method level. @Bean annotation works with @Configuration to create Spring beans. As mentioned earlier, @Configuration will have methods to instantiate and configure dependencies. Such methods will be annotated with @Bean. The method annotated with this annotation works as bean ID and it creates and returns the actual bean.

Here is an example:

@Configuration

public class AppConfig{

@Bean

public Person person(){

return new Person(address());

}

@Bean

public Address address(){

return new Address();

}

}

@Lazy

This annotation is used on component classes. By default all autowired dependencies are created and configured at startup. But if you want to initialize a bean lazily, you can use @Lazy annotation over the class. This means that the bean will be created and initialized only when it is first requested for. You can also use this annotation on @Configuration classes. This indicates that all @Bean methods within that @Configuration should be lazily initialized.

@Value

This annotation is used at the field, constructor parameter, and method parameter level. The @Value annotation indicates a default value expression for the field or parameter to initialize the property with. As the @Autowired annotation tells Spring to inject object into another when it loads your application context, you can also use @Value annotation to inject values from a property file into a bean’s attribute. It supports both #{...} and ${...} placeholders.

Spring Framework Stereotype Annotations

@Component

This annotation is used on classes to indicate a Spring component. The @Component annotation marks the Java class as a bean or say component so that the component-scanning mechanism of Spring can add into the application context.

@Controller

The @Controller annotation is used to indicate the class is a Spring controller. This annotation can be used to identify controllers for Spring MVC or Spring WebFlux.

@Service

This annotation is used on a class. The @Service marks a Java class that performs some service, such as execute business logic, perform calculations and call external APIs. This annotation is a specialized form of the @Component annotation intended to be used in the service layer.

@Repository

This annotation is used on Java classes which directly access the database. The @Repository annotation works as marker for any class that fulfills the role of repository or Data Access Object.

This annotation has a automatic translation feature. For example, when an exception occurs in the @Repository there is a handler for that exception and there is no need to add a try catch block.

Spring Boot Annotations

@EnableAutoConfiguration

This annotation is usually placed on the main application class. The @EnableAutoConfiguration annotation implicitly defines a base “search package”. This annotation tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

@SpringBootApplication

This annotation is used on the application class while setting up a Spring Boot project. The class that is annotated with the @SpringBootApplication must be kept in the base package. The one thing that the @SpringBootApplication does is a component scan. But it will scan only its sub-packages. As an example, if you put the class annotated with @SpringBootApplication in com.example then @SpringBootApplication will scan all its sub-packages, such as com.example.a, com.example.b, and com.example.a.x.

The @SpringBootApplication is a convenient annotation that adds all the following:

@Configuration

@EnableAutoConfiguration

@ComponentScan

Spring MVC and REST Annotations

@Controller

This annotation is used on Java classes that play the role of controller in your application. The @Controller annotation allows autodetection of component classes in the classpath and auto-registering bean definitions for them. To enable autodetection of such annotated controllers, you can add component scanning to your configuration. The Java class annotated with @Controller is capable of handling multiple request mappings.

This annotation can be used with Spring MVC and Spring WebFlux.

@RequestMapping

This annotation is used both at class and method level. The @RequestMapping annotation is used to map web requests onto specific handler classes and handler methods. When @RequestMapping is used on class level it creates a base URI for which the controller will be used. When this annotation is used on methods it will give you the URI on which the handler methods will be executed. From this you can infer that the class level request mapping will remain the same whereas each handler method will have their own request mapping.

Sometimes you may want to perform different operations based on the HTTP method used, even though the request URI may remain the same. In such situations, you can use the method attribute of @RequestMapping with an HTTP method value to narrow down the HTTP methods in order to invoke the methods of your class.

Here is a basic example on how a controller along with request mappings work:

@Controller

@RequestMapping("/welcome")

public class WelcomeController{

@RequestMapping(method = RequestMethod.GET)

public String welcomeAll(){

return "welcome all";

}

}

In this example only GET requests to /welcome is handled by the welcomeAll() method.

This annotation also can be used with Spring MVC and Spring WebFlux.

The @RequestMapping annotation is very versatile. Please see my in depth post on Request Mapping bere.

@CookieValue

This annotation is used at method parameter level. @CookieValue is used as argument of request mapping method. The HTTP cookie is bound to the @CookieValue parameter for a given cookie name. This annotation is used in the method annotated with @RequestMapping.

Let us consider that the following cookie value is received with a http request:

JSESSIONID=418AB76CD83EF94U85YD34W

To get the value of the cookie, use @CookieValue like this:

@RequestMapping("/cookieValue")

public void getCookieValue(@CookieValue "JSESSIONID" String cookie){

}

@CrossOrigin

This annotation is used both at class and method level to enable cross origin requests. In many cases the host that serves JavaScript will be different from the host that serves the data. In such a case Cross Origin Resource Sharing (CORS) enables cross-domain communication. To enable this communication you just need to add the @CrossOrigin annotation.

By default the @CrossOrigin annotation allows all origin, all headers, the HTTP methods specified in the @RequestMapping annotation and maxAge of 30 min. You can customize the behavior by specifying the corresponding attribute values.

An example to use @CrossOrigin at both controller and handler method levels is this.

@CrossOrigin(maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin(origins = "http://example.com")

@RequestMapping("/message")

public Message getMessage() {

// ...

}

@RequestMapping("/note")

public Note getNote() {

// ...

}

}

In this example, both getExample() and getNote() methods will have a maxAge of 3600 seconds. Also, getExample() will only allow cross-origin requests from http://example.com, while getNote() will allow cross-origin requests from all hosts.

Composed @RequestMapping Variants

Spring framework 4.3 introduced the following method-level variants of @RequestMapping annotation to better express the semantics of the annotated methods. Using these annotations have become the standard ways of defining the endpoints. They act as wrapper to @RequestMapping.

These annotations can be used with Spring MVC and Spring WebFlux.

@GetMapping

This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

@PostMapping

This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

@PutMapping

This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PUT)

@PatchMapping

This annotation is used for mapping HTTP PATCH requests onto specific handler methods. @PatchMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.PATCH)

@DeleteMapping

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

[divider style=”4″]

@ExceptionHandler

This annotation is used at method levels to handle exception at the controller level. The @ExceptionHandler annotation is used to define the class of exception it will catch. You can use this annotation on methods that should be invoked to handle an exception. The @ExceptionHandler values can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with matching @ExceptionHandler will be invoked.

@InitBinder

This annotation is a method level annotation that plays the role of identifying the methods which initialize the WebDataBinder – a DataBinder that binds the request parameter to JavaBean objects. To customise request parameter data binding , you can use @InitBinder annotated methods within our controller. The methods annotated with @InitBinder all argument types that handler methods support.

The @InitBinder annotated methods will get called for each HTTP request if you don’t specify the value element of this annotation. The value element can be a single or multiple form names or request parameters that the init binder method is applied to.

@Mappings and @Mapping

This annotation is used on fields. The @Mapping annotation is a meta annotation that indicates a web mapping annotation. When mapping different field names, you need to configure the source field to its target field and to do that you have to add the @Mappings annotation. This annotation accepts an array of @Mapping having the source and the target fields.

@MatrixVariable

This annotation is used to annotate request handler method arguments so that Spring can inject the relevant bits of matrix URI. Matrix variables can appear on any segment each separated by a semicolon. If a URL contains matrix variables, the request mapping pattern must represent them with a URI template. The @MatrixVariable annotation ensures that the request is matched with the correct matrix variables of the URI.

@PathVariable

This annotation is used to annotate request handler method arguments. The @RequestMapping annotation can be used to handle dynamic changes in the URI where certain URI value acts as a parameter. You can specify this parameter using a regular expression. The @PathVariable annotation can be used declare this parameter.

@RequestAttribute

This annotation is used to bind the request attribute to a handler method parameter. Spring retrieves the named attributes value to populate the parameter annotated with @RequestAttribute. While the @RequestParam annotation is used bind the parameter values from query string, the @RequestAttribute is used to access the objects which have been populated on the server side.

@RequestBody

This annotation is used to annotate request handler method arguments. The @RequestBody annotation indicates that a method parameter should be bound to the value of the HTTP request body. The HttpMessageConveter is responsible for converting from the HTTP request message to object.

@RequestHeader

This annotation is used to annotate request handler method arguments. The @RequestHeader annotation is used to map controller parameter to request header value. When Spring maps the request, @RequestHeader checks the header with the name specified within the annotation and binds its value to the handler method parameter. This annotation helps you to get the header details within the controller class.

@RequestParam

This annotation is used to annotate request handler method arguments. Sometimes you get the parameters in the request URL, mostly in GET requests. In that case, along with the @RequestMapping annotation you can use the @RequestParam annotation to retrieve the URL parameter and map it to the method argument. The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

@RequestPart

This annotation is used to annotate request handler method arguments. The @RequestPart annotation can be used instead of @RequestParam to get the content of a specific multipart and bind to the method argument annotated with @RequestPart. This annotation takes into consideration the “Content-Type” header in the multipart(request part).

@ResponseBody

This annotation is used to annotate request handler methods. The @ResponseBody annotation is similar to the @RequestBody annotation. The @ResponseBody annotation indicates that the result type should be written straight in the response body in whatever format you specify like JSON or XML. Spring converts the returned object into a response body by using the HttpMessageConveter.

@ResponseStatus

This annotation is used on methods and exception classes. @ResponseStatus marks a method or exception class with a status code and a reason that must be returned. When the handler method is invoked the status code is set to the HTTP response which overrides the status information provided by any other means. A controller class can also be annotated with @ResponseStatus which is then inherited by all @RequestMapping methods.

@ControllerAdvice

This annotation is applied at the class level. As explained earlier, for each controller you can use @ExceptionHandler on a method that will be called when a given exception occurs. But this handles only those exception that occur within the controller in which it is defined. To overcome this problem you can now use the @ControllerAdvice annotation. This annotation is used to define @ExceptionHandler, @InitBinder and @ModelAttribute methods that apply to all @RequestMapping methods. Thus if you define the @ExceptionHandler annotation on a method in @ControllerAdvice class, it will be applied to all the controllers.

@RestController

This annotation is used at the class level. The @RestController annotation marks the class as a controller where every method returns a domain object instead of a view. By annotating a class with this annotation you no longer need to add @ResponseBody to all the RequestMapping method. It means that you no more use view-resolvers or send html in response. You just send the domain object as HTTP response in the format that is understood by the consumers like JSON.

@RestController is a convenience annotation which combines @Controller and @ResponseBody.

@RestControllerAdvice

This annotation is applied on Java classes. @RestControllerAdvice is a convenience annotation which combines @ControllerAdvice and @ResponseBody. This annotation is used along with the @ExceptionHandler annotation to handle exceptions that occur within the controller.

@SessionAttribute

This annotation is used at method parameter level. The @SessionAttribute annotation is used to bind the method parameter to a session attribute. This annotation provides a convenient access to the existing or permanent session attributes.

@SessionAttributes

This annotation is applied at type level for a specific handler. The @SessionAtrributes annotation is used when you want to add a JavaBean object into a session. This is used when you want to keep the object in session for short lived. @SessionAttributes is used in conjunction with @ModelAttribute.

Consider this example.

@ModelAttribute("person")

public Person getPerson(){}

// within the same controller as above snippet

@Controller

@SeesionAttributes(value="person", types={Person.class})

public class PersonController{}

The @ModelAttribute name is assigned to the @SessionAttributes as value. The @SessionAttributes has two elements. The value element is the name of the session in the model and the types element is the type of session attributes in the model.

Spring Cloud Annotations

@EnableConfigServer

This annotation is used at the class level. When developing a project with a number of services, you need to have a centralized and straightforward manner to configure and retrieve the configurations about all the services that you are going to develop. One advantage of using a centralized config server is that you don’t need to carry the burden of remembering where each configuration is distributed across multiple and distributed components.

You can use Spring cloud’s @EnableConfigServer annotation to start a config server that the other applications can talk to.

@EnableEurekaServer

This annotation is applied to Java classes. One problem that you may encounter while decomposing your application into microservices is that, it becomes difficult for every service to know the address of every other service it depends on. There comes the discovery service which is responsible for tracking the locations of all other microservices.

Netflix’s Eureka is an implementation of a discovery server and integration is provided by Spring Boot. Spring Boot has made it easy to design a Eureka Server by just annotating the entry class with @EnableEurekaServer.

@EnableDiscoveryClient

This annotation is applied to Java classes. In order to tell any application to register itself with Eureka you just need to add the @EnableDiscoveryClient annotation to the application entry point. The application that’s now registered with Eureka uses the Spring Cloud Discovery Client abstraction to interrogate the registry for its own host and port.

@EnableCircuitBreaker

This annotation is applied on Java classes that can act as the circuit breaker. The circuit breaker pattern can allow a micro service continue working when a related service fails, preventing the failure from cascading. This also gives the failed service a time to recover.

The class annotated with @EnableCircuitBreaker will monitor, open, and close the circuit breaker.

@HystrixCommand

This annotation is used at the method level. Netflix’s Hystrix library provides the implementation of Circuit Breaker pattern. When you apply the circuit breaker to a method, Hystrix watches for the failures of the method. Once failures build up to a threshold, Hystrix opens the circuit so that the subsequent calls also fail. Now Hystrix redirects calls to the method and they are passed to the specified fallback methods.

Hystrix looks for any method annotated with the @HystrixCommand annotation and wraps it into a proxy connected to a circuit breaker so that Hystrix can monitor it.

Consider the following example:

@Service

public class BookService{

private final RestTemplate restTemplate;

public BookService(RestTemplate rest){

this.restTemplate = rest;

}

@HystrixCommand(fallbackMethod = "newList") public String bookList(){

URI uri = URI.create("http://localhost:8081/recommended"); return this.restTemplate.getForObject(uri, String.class);

}

public String newList(){

return "Cloud native Java";

}

}

Here @HystrixCommand is applied to the original method bookList(). The @HystrixCommand annotation has newList as the fallback method. So for some reason if Hystrix opens the circuit on bookList(), you will have a placeholder book list ready for the users.

Spring Framework 5Learn Spring Framework 5 with my Spring Framework 5: Beginner to Guru course!

Spring Framework DataAccess Annotations

@Transactional

This annotation is placed before an interface definition, a method on an interface, a class definition, or a public method on a class. The mere presence of @Transactional is not enough to activate the transactional behaviour. The @Transactional is simply a metadata that can be consumed by some runtime infrastructure. This infrastructure uses the metadata to configure the appropriate beans with transactional behaviour.

The annotation further supports configuration like:

The Propagation type of the transaction

The Isolation level of the transaction

A timeout for the operation wrapped by the transaction

A read only flag – a hint for the persistence provider that the transaction must be read only

The rollback rules for the transaction

Cache-Based Annotations

@Cacheable

This annotation is used on methods. The simplest way of enabling the cache behaviour for a method is to annotate it with @Cacheable and parameterize it with the name of the cache where the results would be stored.

@Cacheable("addresses")

public String getAddress(Book book){...}

In the snippet above , the method getAddress is associated with the cache named addresses. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated.

@CachePut

This annotation is used on methods. Whenever you need to update the cache without interfering the method execution, you can use the @CachePut annotation. That is, the method will always be executed and the result cached.

@CachePut("addresses")

public String getAddress(Book book){...}

Using @CachePut and @Cacheable on the same method is strongly discouraged as the former forces the execution in order to execute a cache update, the latter causes the method execution to be skipped by using the cache.

@CacheEvict

This annotation is used on methods. It is not that you always want to populate the cache with more and more data. Sometimes you may want remove some cache data so that you can populate the cache with some fresh values. In such a case use the @CacheEvict annotation.

@CacheEvict(value="addresses", allEntries="true")

public String getAddress(Book book){...}

Here an additional element allEntries is used along with the cache name to be emptied. It is set to true so that it clears all values and prepares to hold new data.

@CacheConfig

This annotation is a class level annotation. The @CacheConfig annotation helps to streamline some of the cache information at one place. Placing this annotation on a class does not turn on any caching operation. This allows you to store the cache configuration at the class level so that you don’t have declare things multiple times.

Task Execution and Scheduling Annotations

@Scheduled

This annotation is a method level annotation. The @Scheduled annotation is used on methods along with the trigger metadata. A method with @Scheduled should have void return type and should not accept any parameters.

There are different ways of using the @Scheduled annotation:

@Scheduled(fixedDelay=5000)

public void doSomething() {

// something that should execute periodically

}

In this case, the duration between the end of last execution and the start of next execution is fixed. The tasks always wait until the previous one is finished.

@Scheduled(fixedRate=5000)

public void doSomething() {

// something that should execute periodically

}

In this case, the beginning of the task execution does not wait for the completion of the previous execution.

@Scheduled(initialDelay=1000,fixedRate=5000)

public void doSomething() {

// something that should execute periodically after an initial delay

}

The task gets executed initially with a delay and then continues with the specified fixed rate.

@Async

This annotation is used on methods to execute each method in a separate thread. The @Async annotation is provided on a method so that the invocation of that method will occur asynchronously. Unlike methods annotated with @Scheduled, the methods annotated with @Async can take arguments. They will be invoked in the normal way by callers at runtime rather than by a scheduled task.

@Async can be used with both void return type methods and the methods that return a value. However methods with return value must have a Future typed return values.

Spring Framework Testing Annotations

@BootstrapWith

This annotation is a class level annotation. The @BootstrapWith annotation is used to configure how the Spring TestContext Framework is bootstrapped. This annotation is used as a metadata to create custom composed annotations and reduce the configuration duplication in a test suite.

@ContextConfiguration

This annotation is a class level annotation that defines a metadata used to determine which configuration files to use to the load the ApplicationContext for your test. More specifically @ContextConfiguration declares the annotated classes that will be used to load the context. You can also tell Spring where to locate for the file.

@ContextConfiguration(locations={"example/test-context.xml", loader = Custom ContextLoader.class})

@WebAppConfiguration

This annotation is a class level annotation. The @WebAppConfiguration is used to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext. This annotation is used to create the web version of the application context. It is important to note that this annotation must be used with the @ContextConfiguration annotation.The default path to the root of the web application is src/main/webapp. You can override it by passing a different path to the <code class="EnlighterJSRAW" data-enlighter-language="java" data-enlighter-theme="git">@WebAppConfiguration</code>.

@Timed

This annotation is used on methods. The @Timed annotation indicates that the annotated test method must finish its execution at the specified time period(in milliseconds). If the execution exceeds the specified time in the annotation, the test fails.

@Timed(millis=10000)

public void testLongRunningProcess() { ... }

In this example, the test will fail if it exceeds 10 seconds of execution.

@Repeat

This annotation is used on test methods. If you want to run a test method several times in a row automatically, you can use the @Repeat annotation. The number of times that test method is to be executed is specified in the annotation.

@Repeat(10)

@Test

public void testProcessRepeatedly() { ... }

In this example, the test will be executed 10 times.

@Commit

This annotation can be used as both class-level or method-level annotation. After execution of a test method, the transaction of the transactional test method can be committed using the @Commit annotation. This annotation explicitly conveys the intent of the code. When used at the class level, this annotation defines the commit for all test methods within the class. When declared as a method level annotation @Commit specifies the commit for specific test methods overriding the class level commit.

@RollBack

This annotation can be used as both class-level and method-level annotation. The @RollBack annotation indicates whether the transaction of a transactional test method must be rolled back after the test completes its execution. If this true @Rollback(true), the transaction is rolled back. Otherwise, the transaction is committed. @Commit is used instead of @RollBack(false).

When used at the class level, this annotation defines the rollback for all test methods within the class.

When declared as a method level annotation @RollBack specifies the rollback for specific test methods overriding the class level rollback semantics.

@DirtiesContext

This annotation is used as both class-level and method-level annotation. @DirtiesContext indicates that the Spring ApplicationContext has been modified or corrupted in some manner and it should be closed. This will trigger the context reloading before execution of next test. The ApplicationContext is marked as dirty before or after any such annotated method as well as before or after current test class.

The @DirtiesContext annotation supports BEFORE\_METHOD, BEFORE\_CLASS, and BEFORE\_EACH\_TEST\_METHOD modes for closing the ApplicationContext before a test.

NOTE: Avoid overusing this annotation. It is an expensive operation and if abused, it can really slow down your test suite.

@BeforeTransaction

This annotation is used to annotate void methods in the test class. @BeforeTransaction annotated methods indicate that they should be executed before any transaction starts executing. That means the method annotated with @BeforeTransaction must be executed before any method annotated with @Transactional.

@AfterTransaction

This annotation is used to annotate void methods in the test class. @AfterTransaction annotated methods indicate that they should be executed after a transaction ends for test methods. That means the method annotated with @AfterTransaction must be executed after the method annotated with @Transactional.

@Sql

This annotation can be declared on a test class or test method to run SQL scripts against a database. The @Sql annotation configures the resource path to SQL scripts that should be executed against a given database either before or after an integration test method. When @Sql is used at the method level it will override any @Sql defined in at class level.

@SqlConfig

This annotation is used along with the @Sql annotation. The @SqlConfig annotation defines the metadata that is used to determine how to parse and execute SQL scripts configured via the @Sql annotation. When used at the class-level, this annotation serves as global configuration for all SQL scripts within the test class. But when used directly with the config attribute of @Sql, @SqlConfig serves as a local configuration for SQL scripts declared.

@SqlGroup

This annotation is used on methods. The @SqlGroup annotation is a container annotation that can hold several @Sql annotations. This annotation can declare nested @Sql annotations.

In addition, @SqlGroup is used as a meta-annotation to create custom composed annotations. This annotation can also be used along with repeatable annotations, where @Sql can be declared several times on the same method or class.

@SpringBootTest

This annotation is used to start the Spring context for integration tests. This will bring up the full autoconfigruation context.

@DataJpaTest

The @DataJpaTest annotation will only provide the autoconfiguration required to test Spring Data JPA using an in-memory database such as H2.

This annotation is used instead of @SpringBootTest

@DataMongoTest

The @DataMongoTest will provide a minimal autoconfiguration and an embedded MongoDB for running integration tests with Spring Data MongoDB.

@WebMVCTest

The @WebMVCTest will bring up a mock servlet context for testing the MVC layer. Services and components are not loaded into the context. To provide these dependencies for testing, the @MockBean annotation is typically used.

@AutoConfigureMockMVC

The @AutoConfigureMockMVC annotation works very similar to the @WebMVCTest annotation, but the full Spring Boot context is started.

@MockBean

Creates and injects a Mockito Mock for the given dependency.

@JsonTest

Will limit the auto configuration of Spring Boot to components relevant to processing JSON.

This annotation will also autoconfigure an instance of JacksonTester or GsonTester.

@TestPropertySource

Class level annotation used to specify property sources for the test class.

## PUT vs PATCH

The main difference between PUT and PATCH requests are in the way the server processes the enclosed entity to modify the resource identified by the Request-URI.

In a PUT request, the enclosed entity is considered to be a modified version of the resource stored on the origin server, and the client is requesting that the stored version be replaced.

With PATCH, however, the enclosed entity contains a set of instructions describing how a resource currently residing on the origin server should be modified to produce a new version.

Also, another difference is that when you want to update a resource with PUT request, you have to send the full payload as the request whereas with PATCH, you only send the parameters which you want to update.

Related:

Learn the Basics of HTTP

List of all HTTP status codes with explanations

Suppose we have a resource that holds the first name and last name of a person.

If we want to change the first name then we send a PUT request for Update

{ "first": "Michael", "last": "Angelo" }

Here, although we are only changing the first name, with PUT request we have to send both parameters first and last. In other words, it is mandatory to send all values again, the full payload.

When we send a PATCH request, however, we only send the data which we want to update. In other words, we only send the first name to update, no need to send the last name.

For this reason, PATCH request requires less bandwidth.

## program to convert numbers to words

**class** Main

{

**private** **static** **final** String ***EMPTY*** = "";

**private** **static** **final** String[] ***X*** =

{

***EMPTY***, "One ", "Two ", "Three ", "Four ", "Five ", "Six ",

"Seven ", "Eight ", "Nine ", "Ten ", "Eleven ","Twelve ",

"Thirteen ", "Fourteen ", "Fifteen ", "Sixteen ",

"Seventeen ", "Eighteen ", "Nineteen "

};

**private** **static** **final** String[] ***Y*** =

{

***EMPTY***, ***EMPTY***, "Twenty ", "Thirty ", "Forty ", "Fifty ",

"Sixty ", "Seventy ", "Eighty ", "Ninety "

};

// Function to convert a single-digit or two-digit number into words

**private** **static** String convertToDigit(**int** n, String suffix)

{

// if `n` is zero

**if** (n == 0) {

**return** ***EMPTY***;

}

// split `n` if it is more than 19

**if** (n > 19) {

**return** ***Y***[n / 10] + ***X***[n % 10] + suffix;

}

**else** {

**return** ***X***[n] + suffix;

}

}

// Function to convert a given number (max 9-digits) into words

**public** **static** String convert(**int** n)

{

// for storing the word representation of the given number

StringBuilder res = **new** StringBuilder();

// add digits at ten million and hundred million place

res.append(*convertToDigit*((n / 1000000000) % 100, "Billion, "));

// add digits at ten million and hundred million place

res.append(*convertToDigit*((n / 10000000) % 100, "Crore, "));

// add digits at hundred thousand and one million place

res.append(*convertToDigit*(((n / 100000) % 100), "Lakh, "));

// add digits at thousand and tens thousand place

res.append(*convertToDigit*(((n / 1000) % 100), "Thousand "));

// add digit at hundred place

res.append(*convertToDigit*(((n / 100) % 10), "Hundred "));

**if** ((n > 100) && (n % 100 != 0)) {

res.append("and ");

}

// add digits at ones and tens place

res.append(*convertToDigit*((n % 100), ""));

**return** res.toString();

}

// Java program to convert numbers to words

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

{

System.***out***.println(*convert*(99));

System.***out***.println(*convert*(1000));

System.***out***.println(*convert*(14632));

System.***out***.println(*convert*(997751076));

System.***out***.println(*convert*(Integer.***MAX\_VALUE***)); // 2147483647

}

}

## Difference between git pull and git fetch

Git Fetch

The fetch command retrieves any commits, references (like tags), branches and files from a remote repository, along with any other corresponding objects. However, not all tags are retrieved as this command only takes the ones that point to commits that you are retrieving. Basically this command fetches anything needed to reconstruct the history of the particular branch you're interested in.

The basic syntax is the following:

$ git fetch <remote-repo> <remote-branch>

Specifying <remote-branch> will only fetch changes from that branch. If this parameter is omitted then changes from all branches are retrieved.

The interesting thing about the fetch command is that it doesn't actually affect anything in your local repo. No working changes will be lost, and you'll see no direct affect on your local branches. This is because Git keeps fetched content separate from your own repo's content until it is merged in.

So, let's say you want to view changes for the "master" branch from the "origin" remote repo before deciding to merge them in to your repo. To do this you could use the following commands:

$ git fetch origin master

Now that you have the changes in your repository, you'll likely want to actually view them, which you can do by just checking out the branch:

$ git checkout origin/master

This will allow you to see the changes, and it still isn't merged in to any of your own branches.

A quicker way to view these changes would be to just look at the commit logs, which you can do with:

$ git log master..origin/master

Learn the Git Essentials

Git Essentials: Developer's Guide to Git

Stop turning to Google every time you need to commit some code - actually understand what you're doing

We cover everything from the fundamentals of Git to some more advanced operations and best practices

Includes a bonus cheat sheet for the many commands you'll still never remember 😉

Just released!

Note that this is considered a "safer" method than pull since it does not actually make any changes to your local branches.

Now that we've seen what fetch does and a bit about how it works, let's take a look at pull.

Git Pull

The git pull command is what I'd call a "high level" command. By that I mean that it performs the actions of a few other Git commands in sequence, which I'll explain more about below. In this section, after I describe the difference between fetch and pull, I'll also briefly talk about the numerous different ways the command can be used.

The general syntax is the following:

$ git pull <remote-repo> <remote-branch>

Both the <remote-repo> and <remote-branch> parameters are optional, as long as your current branch is tracking a remote one.

Probably the simplest way to explain this command, and how it's different from fetch, is that it's an alias for two other Git commands, when used in its default mode: fetch and merge. So by running git pull you are essentially running these two commands in sequence:

$ git fetch <remote-repo>

$ git merge FETCH\_HEAD

Here FETCH\_HEAD is a reference to tip of the last fetch, which is being merged in to your current branch.

So obviously the big difference between fetch and pull is that pull actually performs a fetch in addition to a merge.

Although depending on the option you give to git pull, it may work differently than this. For example, if you add the --rebase option then it will instead use git rebase instead of git merge.

There is also the --no-commit option, which will perform the merge command, but will (as stated in the official documentation) "pretend the merge failed" and not autocommit it. This allows you to take a look at the changes you just fetched before actually committing it to your code.

## What is jit compiler in java

The Just-In-Time (**JIT**) **compiler** is a component of the runtime environment that improves the performance of **Java**™ applications by **compiling** bytecodes to native machine code at run time. ... The **JIT compiler** helps improve the performance of **Java** programs by **compiling** bytecodes into native machine code at run time.

## difference between REST and SOAP

|  |  |  |
| --- | --- | --- |
| **No.** | **SOAP** | **REST** |
| 1) | SOAP is a **protocol**. | REST is an **architectural style**. |
| 2) | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| 3) | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| 4) | SOAP **uses services interfaces to expose the business logic**. | REST **uses URI to expose business logic**. |
| 5) | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| 6) | SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |
| 7) | SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| 8) | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| 9) | SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| 10) | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |

## What is static and Heap memory

Java Heap Space vs. Stack Memory: How Java Applications Allocate Memory

ANGELA STRINGFELLOWSEPTEMBER 5, 2017DEVELOPER TIPS, TRICKS & RESOURCES

Java applications need a certain amount of RAM on a computer to run. Each time an object or variable is declared, it needs more RAM. Simply designating enough memory to hold every value declared and run each method would lead to a bloated application.

To keep application memory requirements lean, it is partitioned in ways that require less memory and allows the application to run more quickly.

The Java Virtual Machine (JVM) divides memory between Java Heap Space and Java Stack Memory in a way that only uses memory that’s needed.

Tip: Find application errors and performance problems instantly with Stackify Retrace

Troubleshooting and optimizing your code is easy with integrated errors, logs and code level performance insights.

Try today for free

What is Java Heap Space

It is created by the Java Virtual Machine when it starts. The memory is used as long as the application is running. Java runtime uses it to allocate memory to objects and Java Runtime Environment (JRE) classes.

When an object is created, it is always created in Heap and has global access. That means all objects can be referenced from anywhere in the application.

It is managed by two concepts: Garbage collection and young-generation, old-generation.

Garbage collection works to free memory by clearing any by objects without any references in the methods. These are objects that are no longer being used. Clearing them ensures they don’t take up space in the Heap.

Young-generation, old-generation helps prioritize objects for garbage collection by dividing Java Heap Space into two generations.

The nursery is the younger generation where the new objects are stored. When the nursery is full, garbage collection cleans it out. Note only the memory space for the nursery is full. There is still memory in the old generation.

The old generation is home to objects have been around long enough. When the old generation runs out of room, garbage collection removes the objects not being used in the old space. Again, only part of the Heap is full when old garbage collection happens. There is still room in the nursery.

What is Java Stack Memory?

This is the temporary memory where variable values are stored when their methods are invoked. After the method is finished, the memory containing those values is cleared to make room for new methods.

When a new method is invoked, a new block of memory will be created in the Stack. This new block will store the temporary values invoked by the method and references to objects stored in the Heap that are being used by the method.

Any values in this block are only accessible by the current method and will not exist once it ends.

When the method ends, that block will be erased. The next method invoked will use that empty block.

This “last in, first out” method makes it easy to find the values needed and allows fast access to those values.

How They’re Used in a Java Application

Let’s look at a very simple example of a Java application to see how memory is allocated.

package com.journaldev.test;package com.journaldev.test;

public class Memory {

public static void main(String[] args) { // Line 1 int i=1; // Line 2 Object obj = new Object(); // Line 3 Memory mem = new Memory(); // Line 4 mem.foo(obj); // Line 5 } // Line 9

private void foo(Object param) { // Line 6 String str = param.toString(); //// Line 7 System.out.println(str); } // Line 8

}

In the above example from JournalDev.com, the use of Stack and Heap is explained as follows:

All Runtime classes are loaded into the Heap Space when the program is run.

Java Runtime creates Stack memory to be used by main() method thread when it is found at line 1. At line 2, a primitive local variable is created, which is stored in the Stack memory of main() method.

Since an Object is created at line 3, it’s created in Heap memory and the reference for it is stored in Stack memory. At line 4, a similar process occurs when a Memory object is created.

When foo() method is called at line 5, a block in the top of the Stack is created for it. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.

At line 7, a string is created, which goes in the String Pool in the Heap space, while a reference for it is created in the foo() stack space. At line 8, foo() method is terminated, and the memory block allocated for it in the Stack is freed.

Finally, at line 9, main() method terminates, and the Stack memory created for it is destroyed. Because the program ends at this line, Java Runtime frees all the memory and ends the execution of the program.

Key Differences

Java Heap Space is used throughout the application, but Stack is only used for the method — or methods — currently running.

The Heap Space contains all objects are created, but Stack contains any reference to those objects.

Objects stored in the Heap can be accessed throughout the application. Primitive local variables are only accessed the Stack Memory blocks that contain their methods.

Memory allocation in the Heap Space is accessed through a complex, young-generation, old-generation system. Stack is accessed through a last-in, first-out (LIFO) memory allocation system.

Heap Space exists as long as the application runs and is larger than Stack, which is temporary, but faster.

Additional Resources and Tutorials

To learn more, check out the following resources:

Java Pass By Value Stack Heap Memory Explanation (YouTube)

Comprehensive Java Developer’s Guide (DZone)

Java’s Garbage-Collected Heap (artima developer)

Java Heap and Stack (Guru99)

JVM Memory Management (JavaBeat)

Stack and heap are two ways Java allocates memory. Understanding when and how they work is critical for developing better Java programs.

It’s also helpful to understand how allocation works when dealing with memory leaks. Additional tools for Java for you to explore include Prefix to write better Java code for free and the full lifecycle APM, Retrace.

## why strings are immutable in java

This is how String works:

String str = "knowledge";

This, as usual, creates a string containing "knowledge" and assigns it a reference str. Simple enough? Lets perform some more functions:

// assigns a new reference to the

// same string "knowledge"

String s = str;

Let’s see how the below statement works:

str = str.concat(" base");

This appends a string " base" to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. "knowledge" and appends " base", giving us the value "knowledge base". Now, since Strings are immutable, the VM can’t assign this value to str, so it creates a new String object, gives it a value "knowledge base", and gives it a reference str.

An important point to note here is that, while the String object is immutable, its reference variable is not. So that’s why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with value "knowledge", pointed to by s, and the second one "knowledge base", pointed to by str. But, technically, we have three String objects, the third one being the literal "base" in the concat statement.

Important Facts about String and Memory usage

What if we didn’t have another reference s to "knowledge"? We would have lost that String. However, it still would have existed, but would be considered lost due to having no references.

Look at one more example below

/\*package whatever // do not write package name here \*/

import java.io.\*;

class GFG {

public static void main(String[] args)

{

String s1 = "java";

s1.concat(" rules");

// Yes, s1 still refers to "java"

System.out.println("s1 refers to " + s1);

}

}

Output:

s1 refers to java

What’s happening:

The first line is pretty straightforward: create a new String "java" and refer s1 to it.

Next, the VM creates another new String "java rules", but nothing refers to it. So, the second String is instantly lost. We can’t reach it.

The reference variable s1 still refers to the original String "java".

Almost every method, applied to a String object in order to modify it, creates new String object. So, where do these String objects go? Well, these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.

As applications grow, it’s very common for String literals to occupy large area of memory, which can even cause redundancy. So, in order to make Java more efficient, the JVM sets aside a special area of memory called the “String constant pool“.

When the compiler sees a String literal, it looks for the String in the pool. If a match is found, the reference to the new literal is directed to the existing String and no new String object is created. The existing String simply has one more reference. Here comes the point of making String objects immutable:

In the String constant pool, a String object is likely to have one or many references. If several references point to same String without even knowing it, it would be bad if one of the references modified that String value. That’s why String objects are immutable.

Well, now you could say, what if someone overrides the functionality of String class? That’s the reason that the String class is marked final so that nobody can override the behavior of its methods.

## What is generics in java8

## Is it possible to create Thread without extending Thread class and implementing Runnable interface

Using AnonymousThread:

1. package com.instanceofjava;
3. public class AnonymousThread
4. {
6. public static void main(String[] args)
7. {
8. new Thread(){
10. public void run(){
12. for (int i = 0; i  <=10; i++) {
13. System.out.println("run"+i);
14. }
16. }
18. }.start();
20. for (int i = 0; i  <=10; i++) {
21. System.out.println("main:"+i);
22. }
23. }
24. }

## . equals & hashcode methods

According to java documentation of equals() method, any implementation should adhere to following principles.

* For any object x, x.equals(x) should return true.
* For any two object x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* For multiple objects x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
* Multiple invocations of x.equals(y) should return same result, unless any of the object properties is modified that is being used in the equals() method implementation.
* Object class equals() method implementation returns true only when both the references are pointing to same object.

Java hashCode()

Java Object hashCode() is a native method and returns the integer hash code value of the object. The general contract of hashCode() method is:

* Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
* An object hash code value can change in multiple executions of the same application.
* If two objects are equal according to equals() method, then their hash code must be same.
* If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.

Importance of equals() and hashCode() method

Java hashCode() and equals() method are used in Hash table based implementations in java for storing and retrieving data. I have explained it in detail at [How HashMap works in java?](https://www.journaldev.com/11560/java-hashmap#how-hashmap-works-in-java)

The implementation of equals() and hashCode() should follow these rules.

* If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

When to override equals() and hashCode() methods?

When we override equals() method, it’s almost necessary to override the hashCode() method too so that their contract is not violated by our implementation.

Note that your program will not throw any exceptions if the equals() and hashCode() contract is violated, if you are not planning to use the class as Hash table key, then it will not create any problem.

If you are planning to use a class as Hash table key, then it’s must to override both equals() and hashCode() methods.

Let’s see what happens when we rely on default implementation of equals() and hashCode() methods and use a custom class as HashMap key.

package com.journaldev.java;

public class DataKey {

private String name;

private int id;

// getter and setter methods

@Override

public String toString() {

return "DataKey [name=" + name + ", id=" + id + "]";

}

}

package com.journaldev.java;

import java.util.HashMap;

import java.util.Map;

public class HashingTest {

public static void main(String[] args) {

Map<DataKey, Integer> hm = getAllData();

DataKey dk = new DataKey();

dk.setId(1);

dk.setName("Pankaj");

System.out.println(dk.hashCode());

Integer value = hm.get(dk);

System.out.println(value);

}

private static Map<DataKey, Integer> getAllData() {

Map<DataKey, Integer> hm = new HashMap<>();

DataKey dk = new DataKey();

dk.setId(1);

dk.setName("Pankaj");

System.out.println(dk.hashCode());

hm.put(dk, 10);

return hm;

}

}

When we run above program, it will print null. It’s because Object hashCode() method is used to find the bucket to look for the key. Since we don’t have access to the HashMap keys and we are creating the key again to retrieve the data, you will notice that hash code values of both the objects are different and hence value is not found.

Implementing equals() and hashCode() method

We can define our own equals() and hashCode() method implementation but if we don’t implement them carefully, it can have weird issues at runtime. Luckily most of the IDE these days provide ways to implement them automatically and if needed we can change them according to our requirement.

Here is the auto generated equals() and hashCode() method implementations.

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + id;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

DataKey other = (DataKey) obj;

if (id != other.id)

return false;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name))

return false;

return true;

}

Notice that both equals() and hashCode() methods are using same fields for the calculations, so that their contract remains valid.

If you will run the test program again, we will get the object from map and program will print 10.

We can also use [Project Lombok](https://www.journaldev.com/18124/java-project-lombok) to auto generate equals and hashCode method implementations.

What is Hash Collision

In very simple terms, Java Hash table implementations uses following logic for get and put operations.

1. First identify the “Bucket” to use using the “key” hash code.
2. If there are no objects present in the bucket with same hash code, then add the object for put operation and return null for get operation.
3. If there are other objects in the bucket with same hash code, then “key” equals method comes into play.
   * If equals() return true and it’s a put operation, then object value is overridden.
   * If equals() return false and it’s a put operation, then new entry is added to the bucket.
   * If equals() return true and it’s a get operation, then object value is returned.
   * If equals() return false and it’s a get operation, then null is returned.

The phenomenon when two keys have same hash code is called hash collision. If hashCode() method is not implemented properly, there will be higher number of hash collision and map entries will not be properly distributed causing slowness in the get and put operations. This is the reason for prime number usage in generating hash code so that map entries are properly distributed across all the buckets.

What if we don’t implement both hashCode() and equals()?

We have already seen above that if hashCode() is not implemented, we won’t be able to retrieve the value because HashMap use hash code to find the bucket to look for the entry.

If we only use hashCode() and don’t implement equals() then also value will be not retrieved because equals() method will return false.

Best Practices for implementing equals() and hashCode() method

* Use same properties in both equals() and hashCode() method implementations, so that their contract doesn’t violate when any properties is updated.
* It’s better to use immutable objects as Hash table key so that we can cache the hash code rather than calculating it on every call. That’s why String is a good candidate for Hash table key because it’s immutable and cache the hash code value.
* Implement hashCode() method so that least number of hash collision occurs and entries are evenly distributed across all the buckets.

## Can we use the final keyword with the constructor

Why a Constructor can not be final, static or abstract in Java?

Difficulty Level : Easy

Last Updated : 26 Dec, 2020

Prerequisite: Inheritance in Java

Constructor in java is a special type of method which is different from normal java methods/ordinary methods. Constructors are used to initialize an object. Automatically a constructor is called when an object of a class is created. It is syntactically similar to a method but it has the same name as its class and a constructor does not have a return type.

Java constructor can not be final

One of the important property of java constructor is that it can not be final. As we know, constructors are not inherited in java. Therefore, constructors are not subject to hiding or overriding. When there is no chance of constructor overriding, there is no chance of modification also. When there is no chance of modification, then no sense of restricting modification there. We know that the final keyword restricts further modification. So a java constructor can not be final because it inherently it cannot be modified. Also, a java constructor is internally final. So again there is no need for final declaration further.

Example: Suppose we are declaring a java constructor as final, now let’s see what is happening.

Java

// Java Constructor as final

class GFG {

// GFG() constructor is declared final

final GFG()

{

// This line can not be executed as compile error

// will come

System.out.print(

"Hey you have declared constructor as final, it's error");

}

}

class Main {

public static void main(String[] args)

{

// Object of GFG class created

// Automatically GFG() constructor called

GFG obj = new GFG();

}

}

Output:

prog.java:4: error: modifier final not allowed here

final GFG( )

^

1 error

From the above example also it is clear that if we are defining constructor as final the compiler will give an error as modifier final not allowed.

Java constructor can not be static

One of the important property of java constructor is that it can not be static. We know static keyword belongs to a class rather than the object of a class. A constructor is called when an object of a class is created, so no use of the static constructor. Another thing is that if we will declare static constructor then we can not access/call the constructor from a subclass. Because we know static is allowed within a class but not by a subclass.

Example:

Java

// java class and a subclass

import java.io.\*;

class GFG {

public GFG()

{

// Constructor of GFG class

System.out.println("GFG Constructor");

}

}

class SubClass extends GFG {

SubClass()

{

// Constructor of SubClass class

// By default super() is hidden here

// So Super class i.e GFG class constructor called

System.out.println("Subclass Constructor");

}

public static void main(String args[])

{

// SubClass class object created

// Automatically SubClass() constructor called

SubClass obj = new SubClass();

}

}

Output

GFG Constructor

Subclass Constructor

Above example expresses that, when an object of subclass is created then Superclass constructor is called by Subclass constructor through constructor chaining. But if we make superclass constructor static then it can’t be called by Subclass as above said static it is accessible within the class but not by the subclass.

One more important reason for not declaring the constructor as static is that, we know a static member is executed first in a program just like the main method which is static and executed first. But constructor is called each and every time when an object is created. But if we will declare it static then the constructor will be called before object creation. So in general if we will see static and constructor are opposite to each other if we want to assign initial values for an instance variable we can use constructor and if we want to assign static variables we can use static blocks.

Example: Suppose we are declaring a java constructor as static, now let’s see what is happening.

Java

// java constructor as static

class GFG {

// GFG() constructor is declared static

static GFG()

{

// This line can not be executed as it compile error

// will come

System.out.print(

"Hey you have declared constructor as static, it's error");

}

}

class Main {

public static void main(String[] args)

{

// Object of GFG class created

// Automatically GFG() constructor called

GFG obj = new GFG();

}

}

Output

prog.java:5: error: modifier static not allowed here

static GFG( ) ^

1 error

From the above example also it is clear that if we are defining constructor as static the compiler will give an error as modifier static not allowed.

Java constructor can not be abstract

One of the important property of java constructor is that it can not be abstract. If we are declaring a constructor as abstract as we have to implement it in a child class, but we know a constructor is called implicitly when the new keyword is used so it can’t lack a body and also it can not be called as a normal method. Also, if we make a constructor abstract then we have to provide the body later. But we know constructor can not be overridden so providing body is impossible. Hence, what we will do with this abstract constructor when we can not provide implementation to it.

Example: Suppose we are declaring a java constructor as abstract, now let’s see what is happening.

Java

// java constructor as static

import java.io.\*;

abstract class GFG {

// GFG() constructor is declared abstract

abstract GFG()

{

// This line can not be executed as compile error

// will come

System.out.print(

"Hey you have declared constructor as abstract, it's error");

}

}

class Main {

public static void main(String[] args)

{

// Object of GFG class created

// Automatically GFG() constructor should be called

// But object creation in abstract class is error

GFG obj = new GFG();

}

}

Output

prog.java:5: error: modifier abstract not allowed here

abstract GFG( )

^

prog.java:17: error: GFG is abstract; cannot be instantiated

GFG obj = new GFG();

^

2 errors

From the above example also it is clear that if we are defining constructor as static the compiler will give an error as modifier abstract not allowed.

Note: Java Interface can not have constructor but Abstract classes can have a constructor.

Attention reader! Don’t stop learning now. Get hold of all the important Java Foundation and Collections concepts with the Fundamentals of Java and Java Collections Course at a student-friendly price and become industry ready. To complete your preparation from learning a language to DS Algo and many more, please refer

## Paste