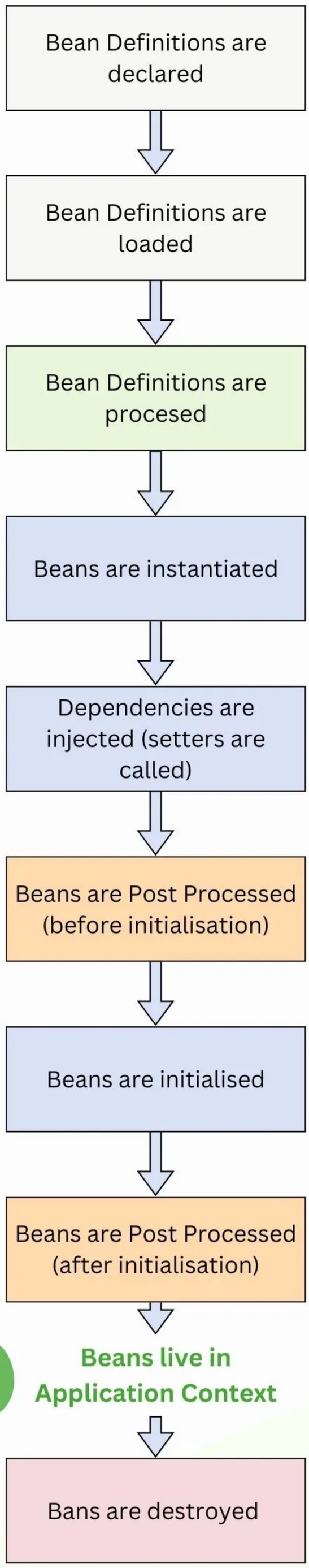
Spring Bean Lifecycle CheatSheet

Spring Bean - POJO (Plain Old Java Object) managed by the IoC container (Spring)

Bean annotations:

@Bean **@Component** **@Service**
@Controller **@RestController** **@Repository**



Bean Definitions are declared in XML | annotations with package scanning | annotations with a @Configuration-annotated class | Groovy configuration

BeanDefinitionReader

parses the configuration and **creates** BeanDefinition objects.

BeanFactoryPostProcessor

tweaks BeanDefinitions before actual bean creation. Implementing *BeanFactoryPostProcessor* interface grants access to the created BeanDefinitions and allows modifications.

BeanFactory

invokes the constructor of each bean. If needed, it delegates this to custom FactoryBean instances.

If dependencies are injected in the constructor, dependent beans are created first, followed by those that depend on them.
Avoid cyclic dependencies: use constructor Injection | @Lazy | Refactor code

BeanPostProcessor

adjusts beans in the first round (after the object is fully created but before initialisation). Called **before the init-method** (if defined) and returns the bean.

The **@PostConstruct** init method of the bean is triggered, and the bean gets initialised

Bean scopes:

Singletone (default) | Prototype | Request | Session | Application | WebSocket | Custom Scopes

BeanPostProcessor

makes their **second pass** on the bean, usually needed if you need to wrap a proxy around the bean, or in case of circular dependencies

Fully created bean is stored in the context and accessible

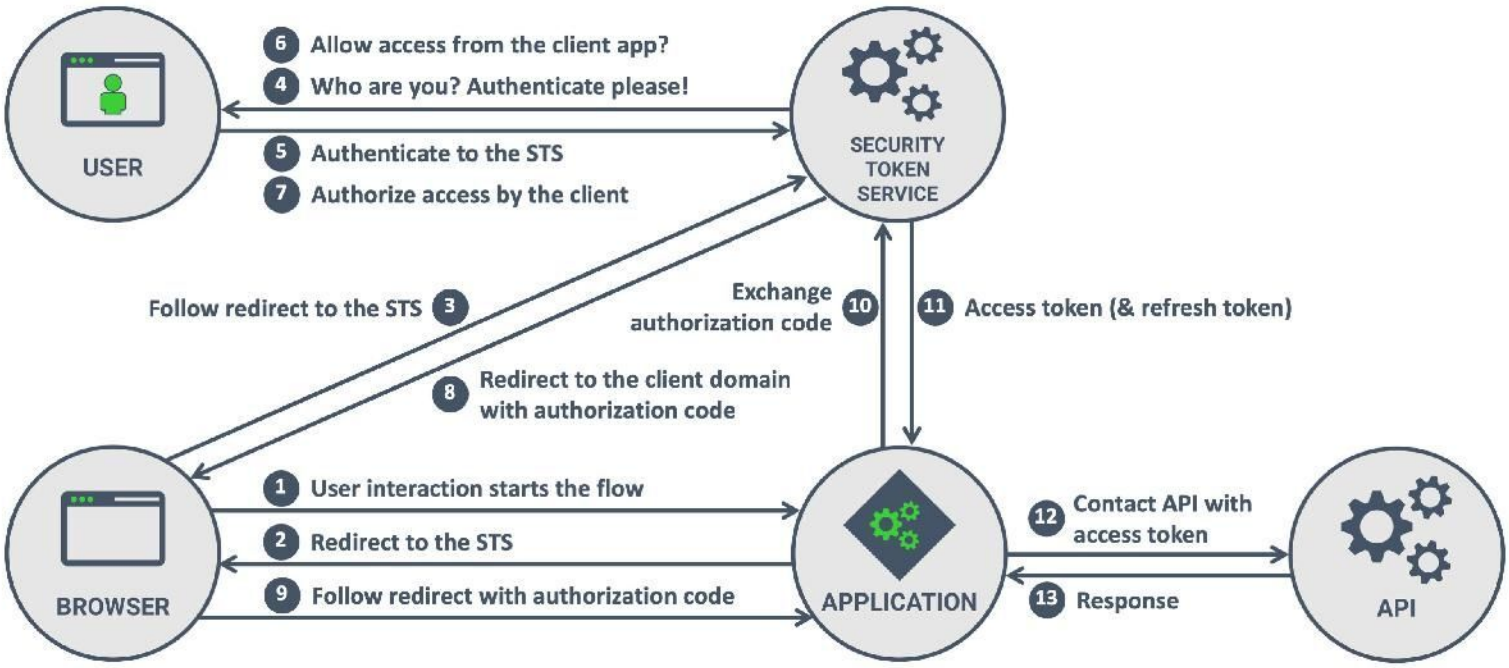
Injection annotations

@Autowired
@Setter
@Qualifier (to specify name when autowiring)
@Primary (primary bean of the same type to inject)

When the context is being destroyed or bean is disposed, the **@PreDestroy** destroy method of the bean is called.

OAUTH 2.0 BEST PRACTICES FOR DEVELOPERS

OAuth 2.0 is an elaborate framework, which continuously evolves to address current needs and security considerations. The framework is even evolving into a consolidated OAuth 2.1 specification. This cheat sheet offers an overview of current security best practices for developers building OAuth 2.x client applications.



GENERAL RECOMMENDATIONS

- ☐ Use the *Authorization Code* flow in every redirect scenario
- ☐ Always use *Proof Key for Code Exchange* (PKCE)
  - The client includes a challenge based on a secret in Step 1
  - The client includes the secret verifier in Step 10
- ⚙ When using refresh tokens, apply additional protection
  - Rotate refresh tokens and act upon double use of a token
  - Invalidate refresh tokens for web applications when ...
    - the user explicitly logs out of the security token service
    - the user's session with the security token service expires
  - Invalidate refresh tokens when the user's password changes
- ☐ Include an audience in the flow and in the access tokens
  - This restricts who accepts the access token in Step 12
- Restrict the capabilities of bearer access tokens
  - Keep the lifetime of access tokens as short as possible
  - Use scopes to restrict the permissions associated with a token

REFERENCES

- ⚙ OAuth 2.0 threat model and security considerations
- ⚙ OAuth 2.0 Security Best Current Practice
- ⚙ The OAuth 2.1 Authorization Framework (draft)

RECOMMENDATIONS FOR BACKEND CLIENTS

- ☐ Use client authentication in Step 10
  - Prefer key-based authentication over shared client secrets
- ☐ Encrypt access tokens and refresh tokens in storage
  - Store the encryption keys using a secret management service
- ☐ Use proof-of-possession access/refresh tokens
  - Using sender-constrained tokens requires possession of a secret

RECOMMENDATIONS FOR FRONTEND WEB CLIENTS

- ☐ Use the *Authorization Code* flow with PKCE for new projects
  - The *Implicit* flow is not broken, but should be phased out
- Be careful with using refresh tokens in web applications
  - Do not use long-lived refresh tokens in the browser
  - Ensure that refresh tokens are protected (see on the left)
- Focus on preventing XSS vulnerabilities in the frontend
  - XSS results in the complete compromise of the client application
  - Avoiding the use of LocalStorage is not an XSS defense

RECOMMENDATIONS FOR NATIVE CLIENTS

- ☐ Use a system browser instead of an embedded browser
  - On mobile, use *SFSafariViewController* or *Chrome Custom Tabs*
- ☐ Encrypt access tokens and refresh tokens in storage
  - Store the encryption keys in a key store provided by the OS

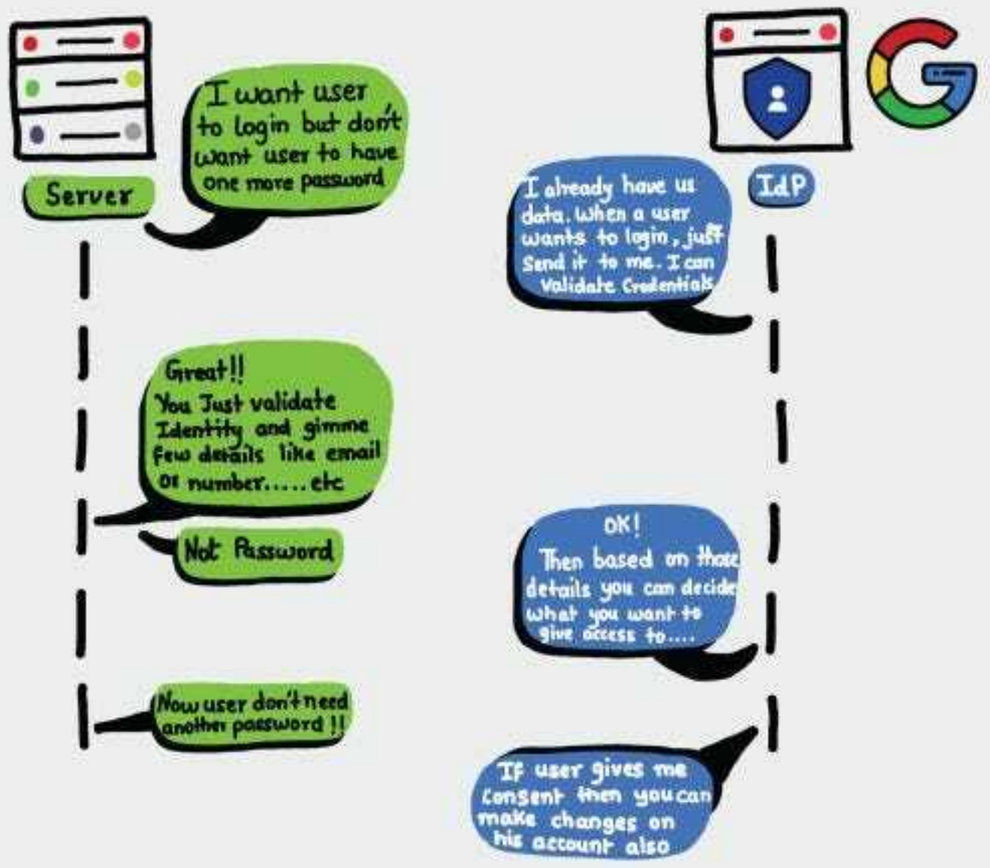
What is OAuth ?

* Protocol for sharing user Authorization across systems.

Involves 3 entities

1. User — Entity who's authorization flows across the Systems. Also known as resource Owner
2. Server — Service, User wants to access and Complete Authorization across.
3. IdP — Identity Provider which Stores user's identity and validate User's Credentials & Share user Authorization with other services.

OAuth



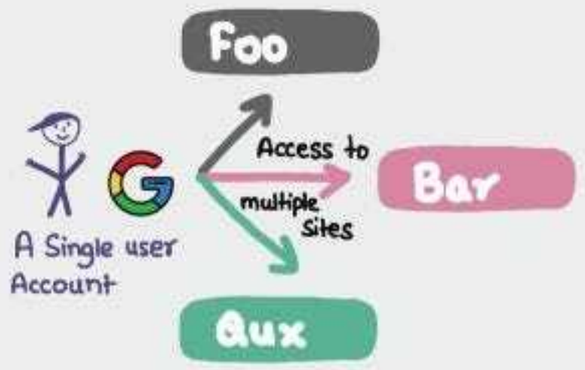
1.0 protocal designed for web browser only

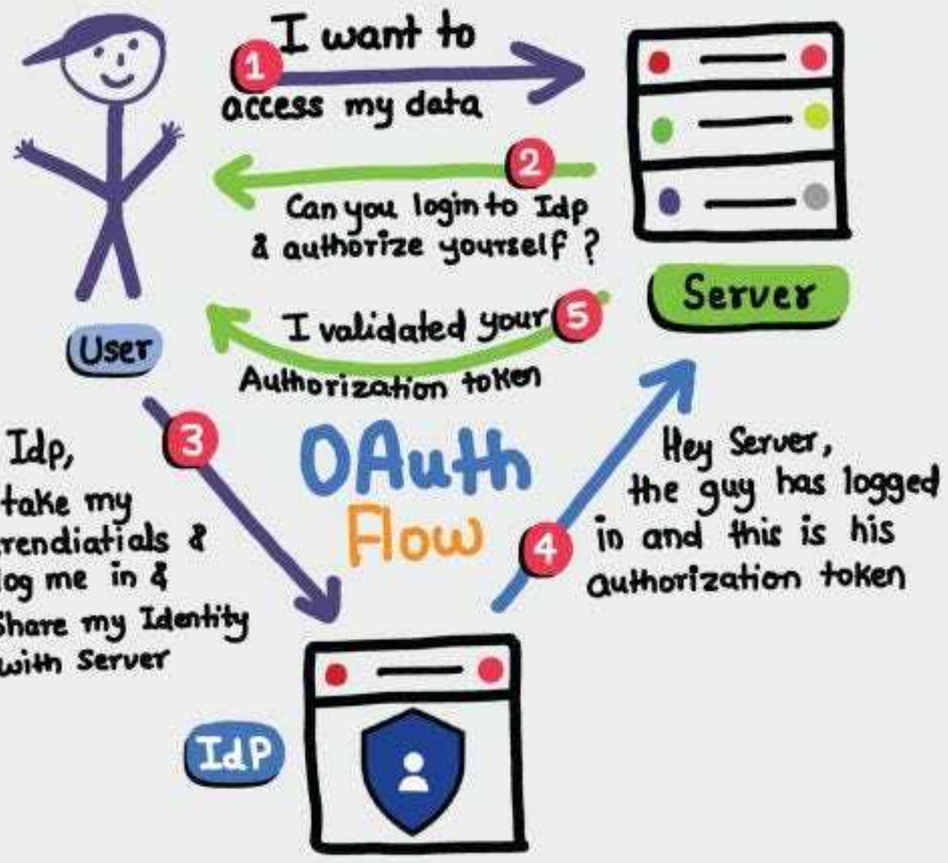
2.0 protocal upgraded for
- browser App
- non browser App
- Windows App
- mobile App
- APIs

OAUTH 2.0

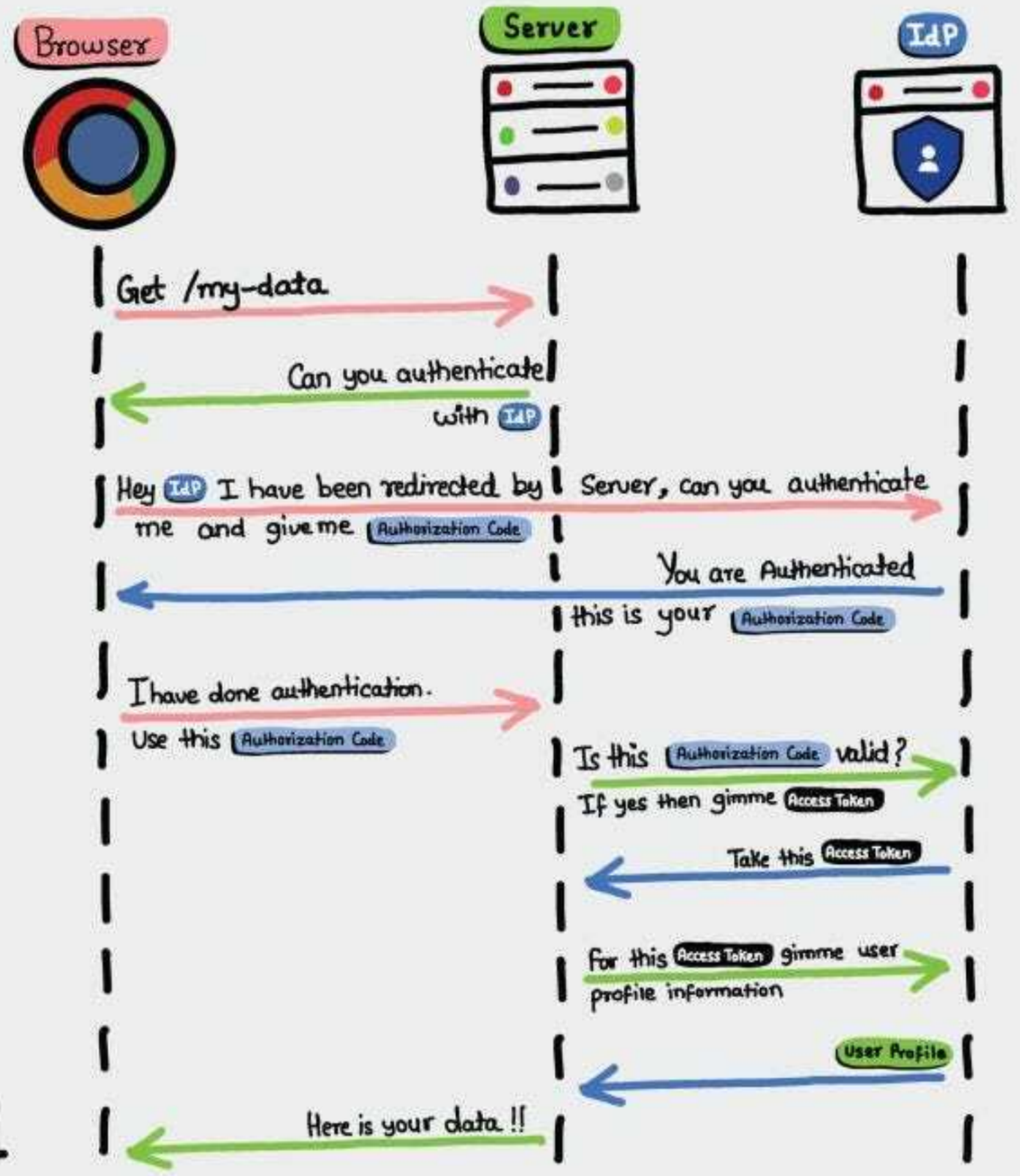
@Sec_80 ByteByteGo

Open Authorization





Authorization Code Flow



4 Types of OAuth Flows

1. Authorization code
2. Client Credentials
3. Implicit Code
4. Resourse owner Password

Cheatography

Stream API (Functional Interface) Cheat Sheet

by c0d3r via cheatography.com/192130/cs/39950/

Java Functional Interface

Consumer<T>	*void* **accept**(T t)
Function<T,R>	*R* **apply**(T t)
Supplier<T>	*T* **get()**
UnaryOperator<T>	*T* **apply**(T t)
Predicate<T>	*boolean* **test**(T t)

Boolean Supplier Functional Interface

BooleanSupplier	*boolean* **getAsBoolean**()

Primitive Functional Interface (integer)

IntConsumer	*void* **accept**(int value)
IntFunction<R>	*R* **apply**(int value)
IntSupplier	*int* **getAsInt**()
IntUnaryOperator	*int* **applyAsInt**(int operand)
IntPredicate	*boolean* **test**(int value)
IntBinaryOperator	*int* **applyAsInt**(int left, int right)

Object & Primitive Functional Interface

ObjDoubleConsumer<T>	*void* **accept**(T t, double value)
ObjIntConsumer<T>	*void* **accept**(T t, int value)
ObjLongConsumer<T>	*void* **accept**(T t, long value)

Primitive Functional Interface (double)

DoubleConsumer	*void* **accept**(double value)
DoubleFunction<R>	*R* **apply**(double value)
DoubleSupplier	*double* **getAsDouble**()
DoubleUnaryOperator	*double* **applyAsDouble**(double operand)
DoublePredicate	*boolean* **test**(double value)
DoubleBinaryOperator	*double* **applyAsDouble**(double left, double right)

To Primitive Functional Interface

ToDoubleBiFunction<T,U>	double **applyAsDouble**(T t, U u)
ToDoubleFunction<T>	double **applyAsDouble**(T value)
ToIntBiFunction<T,U>	int **applyAsInt**(T t, U u)
ToIntFunction<T>	int **applyAsInt**(T value)
ToLongBiFunction<T,U>	*long* **applyAsLong**(T t, U u)
ToLongFunction<T>	*long* **applyAsLong**(T value)

Java Functional Interface (Bi**)

BiConsumer<T,U>	*void* **accept**(T t, U u)
BinaryOperator<T>	*T* **apply**(T t, T u)
BiFunction<T,U,R>	*R* **apply**(T t, U u)
BiPredicate<T,U>	*boolean* **test**(T t, U u)

Primitive to primate Functional Interface

DoubleToIntFunction	int **applyAsInt**(double value)
DoubleToLongFunction	*long* **applyAsLong**(double value)
IntToDoubleFunction	double **applyAsDouble**(int value)
IntToLongFunction	*long* **applyAsLong**(int value)
LongToDoubleFunction	double **applyAsDouble**(long value)
LongToIntFunction	int **applyAsInt**(long value)

Primitive Functional Interface (long)

LongConsumer	*void* **accept**(long value)
LongFunction<R>	*R* **apply**(long value)
LongSupplier	*long* **getAsLong**()
LongUnaryOperator	*long* **applyAsLong**(long operand)
LongPredicate	*boolean* **test**(long value)
LongBinaryOperator	*long* **applyAsLong**(long left, long right)

Cheatography

Spring Framework 4 Cheat Sheet

by danielfc via cheatography.com/24913/cs/6341/

Modules

Core	spring-core spring-beans spring-context spring-expression
AOP and Instrumentation	spring-aop spring-aspects spring-instrument spring-instrument-tomcat
Messaging	spring-messaging
Data Access/Integration	spring-jdbc spring-tx spring-orm spring-oxm spring-jms
Web	spring-web spring-webmvc spring-webmvc-portlet spring-websocket
Test	spring-test

http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html#overview-modules

Spring MVC - Controller

@Controller	Annotation to indicate that the class is a controller class.
@RestController	A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`. Used in controllers that will behave as **RESTful resources**.
@RequestMapping	Annotation to be used on methods in `@RestController` classes. You can provide an **URI** to be served as **RESTful service**.
@ModelAttribute	Annotation used to bind values present in views.

Configuration

@Configuration	Annotation used to provide **configurations**.

Configuration (cont)

@Bean	Annotation that acts like a **provider** where you can define **how** the bean is **instantiated** when a **injection** of that type is requested. Instances of `@Bean` annotated methods will act as **singletons**.

Properties Evaluation Sequence

Command-line arguments	`java -Dproject.name=Test -jar app.jar`
System properties	`System.getProperties()`
Environment Variable	`export PROJECT_NAME=Test`
External properties/yml file	`project.name=Test`
Internal properties/yml file	`project.name=Test`

The default properties/yml files are **application.properties** and **application.yml** and they are located in `/src/resources`.

Spring Boot Initializer

http://start.spring.io	Web service that allows the user to specify the project metadata and dependencies as well as download the initial structure.
Spring CLI	A CLI tool that interacts with **http://start.spring.io** service to scaffold a new project.
Spring Tool Suit	Eclipse-based IDE that also interacts with **http://start.spring.io** to scaffold a new project.
Intellij IDEA	Intellij also provides a way of creating a new project via **http://start.spring.io**.

Spring Boot - Auto Configuration

@ConditionalOnClass	`@ConditionalOnClass (Tomcat.class)`	Only available if the Tomcat class is found in the classpath.
@ConditionalOnProperty	`@ConditionalOnProperty(name = "tomcat.version", matchIfMissing = true)`	Only available if the property `tomcat.version` is set to true.

Auto configuration is just the combination of `@Configuration` and `@Conditional*` annotations in order to correctly register beans.

By **danielfc**
cheatography.com/danielfc/

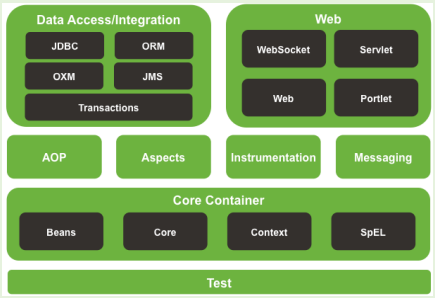
Published 9th December, 2015.
Last updated 12th December, 2016.

Cheatography

Spring Framework 4 Cheat Sheet

by danielfc via cheatography.com/24913/cs/6341/

Architecture



Dependency Injection

@Resource	Annotation used to **inject** an object that is already in the **Application Context**. It searches the instance **by name**. It also works on setter methods.
@Autowired	Annotation used to **inject** objects in many possible ways, such as: **instance variable, constructor and methods** It **does not rely on name** as `@Resource`, so, for multiple concrete implementations, the `@Qualifier` annotation must be used with it.
@Qualifier	Annotation used to **distinguish** between **multiple** concrete implementations. Used alongside with `@Autowired` annotation that does not rely on name.
@Primary	Annotation used when **no name** is provided telling *Spring* to **inject** an object of the annotated class **first**. Used along with `@Component`.
@Component	Generic stereotype annotation used to tell *Spring* to **create an instance** of the object in the **Application Context**. It's possible to define any **name** for the instance, the default is the class name as **camel case**.
@Controller	Stereotype annotation for presentation layer.
@Repository	Stereotype annotation for persistence layer.
@Service	Stereotype annotation for service layer.

Profile

spring.profiles.active	Property to be set in **application.properties** in order to tell *Spring* what profiles are active.
@Profile("!dev")	Annotation used to define which profile **can execute** the annotated method.

Spring Boot - Basics

@SpringBootApplication	Initial annotation that comprises the following annotations: `@SpringBootConfiguration`, `@EnableAutoConfiguration` and `@ComponentScan`.
@SpringBootConfiguration	Indicates that a class provides Spring Boot application `@Configuration`.
@EnableAutoConfiguration	Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need.
@ComponentScan	Configures component scanning directives for use with `@Configuration` classes.

Most of the time you will need only to declare the `@SpringBootApplication` annotation.

Spring Boot - Example

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


By **danielfc**

cheatography.com/danielfc/

Published 9th December, 2015.

Last updated 12th December, 2016.

The Latest Features in Java 22

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Unnamed Variables & Patterns – JEP 456 🌟

As of Java 22, you can use the underscore "_" to mark variables, pattern variables and entire patterns that you will not use. For example, in exceptions, lambdas, and pattern matching:

```
try {
  number = Integer.parseInt(string);
} catch (NumberFormatException _) {
  System.err.println("Not a number");
}
```

```
map
    .computeIfAbsent(key, _ -> new ArrayList<>())
    .add(value);
```

```
if (object instanceof Path(Position(int x1, int y1), _)) {
  System.out.printf("object is a path starting at x = %d, y = %d%n", x1, y1);
}
```

```
switch (object) {
  case Byte _, Short _, Integer _, Long _ -> System.out.println("Integer number");
  case Float _, Double _                 -> System.out.println("Floating point number");
  default                                -> System.out.println("Not a number");
}
```

Launch Multi-File Source-Code Programs – JEP 458 🌟

Since Java 11, we have been able to run Java programs consisting of just one file directly without having to explicitly compile them with *javac* beforehand. In Java 22, this capability has been extended so that programs consisting of several files can now also be started without prior compilation.

For example, save the following two classes in one file each, *Hello.java* and *Greetings.java*:

```
public class Hello {
 public static void main(String[] args) {
   System.out.println(Greetings.greet(args[0]));
 }
}
```

```
public class Greetings {
 public static String greet(String name) {
   return "Hello %s!%n".formatted(name);
 }
}
```

You can then start the program as follows without first compiling it with *javac*:

```
$ java Hello.java World
Hello World!
```

Foreign Function & Memory API – JEP 454 ⭐

The Foreign Function & Memory API (or FFM API for short) makes it possible to access code outside the Java Virtual Machine (e.g., functions in libraries written in other programming languages than Java) and native memory (i.e., memory that is not managed by the JVM on the heap) from Java.

In the example on the right, we call the *strlen()* function of the standard C library to determine the length of the string "Happy Coding!".

```
Linker linker = Linker.nativeLinker();
SymbolLookup stdlib = linker.defaultLookup();

MethodHandle strlen = linker.downcallHandle(
   stdlib.find("strlen").orElseThrow(),
   FunctionDescriptor.of(JAVA_LONG, ADDRESS));

try (Arena offHeap = Arena.ofConfined()) {
  MemorySegment str = offHeap.allocateFrom("Happy Coding!");
  long len = (long) strlen.invoke(str);
  System.out.println("len = " + len);
}
```

You can find a detailed description at https://www.happycoders.eu/java/foreign-function-memory-api/.

Region Pinning for G1 – JEP 423

While working with memory addresses of Java objects when working with JNI (Java Native Interface), the garbage collector must not move these objects to another position in memory. As the G1 garbage collector was previously unable to protect individual objects from being moved, garbage collection was completely paused until these memory addresses were released again. Depending on the behavior of an application, this could have drastic consequences for memory consumption and performance.

Thanks to "region pinning" added in Java 22, the G1 Garbage Collector can now "pin" a memory region containing such an object, i.e., not move the objects it contains to another memory address, so that garbage collection no longer has to be paused completely.

Bonus: Preview Features in Java 22 (Part 1)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Statements before super(...) (Preview) – JEP 447 🌟

Previously, it was not possible to execute additional code before calling the super constructor with *super(...)* or before calling an alternative constructor with *this(...)*. The validation of parameters or the calculation of arguments for the call of *super(...)* or *this(...)*, therefore, had to be extracted to separate methods, which could make the code quite unreadable.

Thanks to JDK Enhancement Proposal 447, code such as the following is now also valid (it previously led to the compiler error "call to super must be first statement in constructor"):

```
public class Rectangle {
  . . .
  public Rectangle(Color color,
                   double width, double height) {
    . . .
  }
}
```

```
public class Square extends Rectangle {
  public Square(Color color, int area) {
    if (area < 0) throw new IllegalArgumentException();
    double sideLength = Math.sqrt(area);
    super(color, sideLength, sideLength);
  }
}
```

Stream Gatherers (Preview) – JEP 461 🌟

With *filter*, *map*, *flatMap*, *mapMulti*, *distinct*, *sorted*, *peak*, *limit*, *skip*, *takeWhile*, and *dropWhile*, the Java stream API only has a limited selection of intermediate stream operations. Alternative operations, such as *window* and *fold*, are not available.

Instead of responding to the diverse wishes of the community and integrating all the desired methods into the JDK, the JDK developers decided instead to develop an API that enables both the JDK developers themselves and the Java community to write any intermediate stream operations. JEP 461 delivers the new *Stream Gatherers* API as well as a whole series of predefined gatherers, including the required *window* and *fold* operations.

Using the *fixed* and *sliding-window* operations, for example, you can group stream elements into lists of predefined sizes:

```
List<String> words = List.of("the", "be", "two", "of", "and", "a", "in", "that");
List<List<String>> fixedWindows   = words.stream().gather(Gatherers.windowFixed(3)).toList();
List<List<String>> slidingWindows = words.stream().gather(Gatherers.windowSliding(3)).toList();
System.out.println(fixedWindows);
System.out.println(slidingWindows);
```

This small example program prints the following:

```
[[the, be, two], [of, and, a], [in, that]]
[[the, be, two], [be, two, of], [two, of, and], [of, and, a], [and, a, in], [a, in, that]]
```

You can find out which other stream gatherers the JDK provides and how you can write your own gatherers for intermediate operations at https://www.happycoders.eu/java/stream-gatherers/.

Class-File API (Preview) – JEP 457

The *Class-File API* is an interface for reading and writing *.class* files, i.e., compiled Java bytecode. The new API is intended to replace the bytecode manipulation framework ASM, which is used intensively in the JDK. You can find further details in the JEP.

Implicitly Declared Classes & Instance Main Methods (Second Preview) – JEP 463 🌟

In future, a simple Java program can be written without a class declaration and with a non-static, non-public, parameterless *main* method. This makes the code on the right a valid and complete Java program.

```
void main() {
  System.out.println("Hello, world!");
}
```

Vector API (Seventh Incubator) – JEP 460

The *Vector API* will make it possible to map vector operations such as the following to the vector instructions of modern CPUs as efficiently as possible. This means that such calculations can be performed in just a few CPU cycles (or in just a single CPU cycle up to a certain vector size):

$$\begin{bmatrix} 7 \\ 13 \\ 8 \end{bmatrix} + \begin{bmatrix} 16 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 7+16 \\ 13+3 \\ 8+6 \end{bmatrix} = \begin{bmatrix} 23 \\ 16 \\ 14 \end{bmatrix}$$

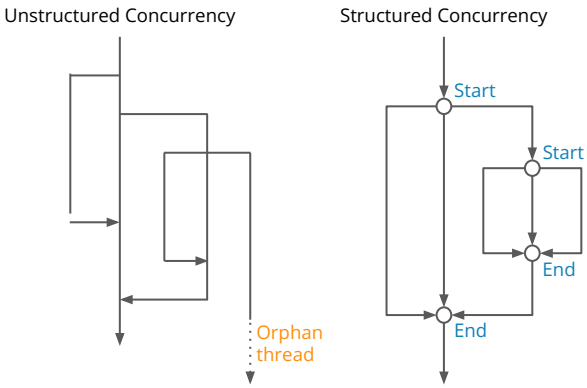
As the Vector API is still in the incubator stage and may therefore still be subject to significant changes, I am not showing any code at this point.

Bonus: Preview Features in Java 22 (Part 2)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Structured Concurrency (Second Preview) – JEP 462 🌟

Structured concurrency enables the simple coordination of concurrent tasks and is characterized by clearly visible start and end points of concurrent subtasks in the code – whereas with *unstructured concurrency*, the start and end of subtasks are much more difficult to recognize in the code (see figure on the right).



With *StructuredTaskScope*, JEP 462 introduces a control structure that clearly defines the start and end of concurrent tasks, enables clean error handling and can cancel subtasks whose results are no longer required in an orderly manner.

In the following example, we concurrently load an order, a customer and an invoice template and create an invoice from them. As soon as one of the subtasks fails, the subtasks that are still running are terminated and the error is propagated to the caller:

```
Invoice createInvoice(int orderId, int customerId, String language)
    throws InterruptedException, ExecutionException {

  try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Subtask<Order>           orderSubtask    = scope.fork(() -> orderService.getOrder(orderId));
    Subtask<Customer>        customerSubtask = scope.fork(() -> customerService.getCustomer(customerId));
    Subtask<InvoiceTemplate> templateSubtask = scope.fork(() -> templateService.getTemplate(language));

    scope.join();
    scope.throwIfFailed();

    var order    = orderSubtask.get();
    var customer = customerSubtask.get();
    var template = templateSubtask.get();

    return Invoice.from(order, customer, template);
  }
}
```

We can just as easily implement a *race()* method that starts two tasks, returns the result of the task that finishes first and cancels the second task (see example on the right).

You can find a detailed description at https://www.happycoders.eu/java/structured-concurrency/.

```
public static <R> R race(Callable<R> task1, Callable<R> task2)
    throws InterruptedException, ExecutionException {

  try (var scope = new StructuredTaskScope.ShutdownOnSuccess<R>()) {
    scope.fork(task1);
    scope.fork(task2);
    scope.join();
    return scope.result();
  }
}
```

Scoped Values (Second Preview) – JEP 464 🌟

Scoped values allow one or more variables to be passed to one or more methods without having to define them as explicit parameters and pass them from one method to the next.

The following example shows how a web server stores the logged-in user as a *scoped value* and calls the *restAdapter.process(...)* method in its scope. The call chain ultimately leads to the call of *Repository.getData(...)*, and this method can then access the logged-in user via the *scoped value*:

```
public class Server {
  public final static ScopedValue<User>
      LOGGED_IN_USER = ScopedValue.newInstance();
  . . .
  private void serve(Request request) {
    User loggedInUser = authenticateUser(request);
    ScopedValue
        .where(LOGGED_IN_USER, loggedInUser)
        .run(() -> restAdapter.process(request));
  }
  . . .
}
```

```
public class Repository {
  . . .
  public Data getData(UUID id) {
    Data data = findById(id);
    User loggedInUser = Server.LOGGED_IN_USER.get();
    if (loggedInUser.isAdmin()) {
      enrichDataWithAdminInfos(data);
    }
    return data;
  }
  . . .
}
```

You can find a detailed description at https://www.happycoders.eu/java/scoped-values/.

The Latest Features in Java 21 (Part 1)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.
You can find the non-finalized JDK Enhancement Proposals from Java 21 on pages 2 and 3.

With the finalization of *virtual threads*, *pattern matching for switch*, and *record patterns*, the Java 21 long-term support (LTS) release is a milestone in Java history. I could not summarize these and the other exciting Java 21 features on a single page. I have, therefore, divided them into two pages.

Virtual Threads – JEP 444 🌟

When scaling server applications, threads are often the bottleneck. Their number is limited, and they often have to wait for events, such as the response to a database query or a remote call, or they are blocked by locks.

Existing solutions such as *CompletableFuture* or reactive frameworks lead to code that is difficult to read and maintain, e.g:

```
@GET
@Path("/product/{productId}")
public CompletionStage<Response> getProduct(@PathParam("productId") String productId) {
  return productService.getProductAsync(productId).thenCompose(product -> {
      if (product.isEmpty()) {
        return CompletableFuture.completedFuture(Response.status(NOT_FOUND).build());
      }
      return warehouseService
          .isAvailableAsync(productId)
          .thenCompose(available -> available
              ? CompletableFuture.completedFuture(0)
              : supplierService.getDeliveryTimeAsync(product.get().supplier(), productId))
          .thenApply(daysUntilShippable ->
              Response.ok(new ProductPageResponse(product.get(), daysUntilShippable)).build());
      });
}
```

Unlike asynchronous code like the one above, virtual threads allow programming in the traditional, familiar, imperative thread-per-request style (the *@RunOnVirtualThread* annotation is defined in Jakarta EE 11):

```
@GET
@Path("/product/{productId}")
@RunOnVirtualThread
public ProductPageResponse getProduct(@PathParam("productId") String productId) {
  Product product = productService.getProduct(productId).orElseThrow(NotFoundException::new);
  boolean available = warehouseService.isAvailable(productId);
  int shipsInDays = available ? 0 : supplierService.getDeliveryTime(product.supplier(), productId);
  return new ProductPageResponse(product, shipsInDays, Thread.currentThread().toString());
}
```

Sequential code is not only much easier to write and read, but also easier to debug, as the program flow can be tracked in a debugger and stack traces reflect the expected call stack.

This is made possible by mapping virtual threads – completely transparently for the developer – to a few platform threads (as the conventional threads are now called). As soon as a virtual thread has to wait or is blocked, the platform thread executes another virtual thread. In this way, we can execute several million (!) virtual threads with just a few operating system threads.

The best part is that we don't have to change existing code. We only have to instruct our application framework not to create platform threads but virtual threads (e.g. with *@RunOnVirtualThread*).

You can find out exactly how virtual threads work, what they can and cannot do, and for which purposes they are not suitable at https://www.happycoders.eu/java/virtual-threads/.

Sequenced Collections – JEP 431 🌟

The new *SequencedCollection* interface offers uniform methods to read, remove or insert the first and last element of an ordered *Collection*, such as *List*, *Deque*, *LinkedHashSet*, or *TreeSet*. Below, you will find a few examples of how Java code can be greatly simplified using the new interface methods:

Operation	Code before Java 21	Code since Java 21
Reading the first element of a *List*	`list.get(0);`	`list.getFirst();`
Reading the last element of a *List*	`list.get(list.size() - 1);`	`list.getLast();`
Reading the first element of a *LinkedHashSet*	`linkedHashSet.iterator().next();`	`linkedHashSet.getFirst();`
Reading the last element of a *LinkedHashSet*	*only possible via iteration over the complete set*	`linkedHashSet.getLast();`

Other methods of *SequencedCollection* are *addFirst(…)*, *addLast(…)*, *removeFirst(…)*, *removeLast(…)*, and *reversed()*, which can be used to iterate over the *Collection* in reverse order.

Another interface is *SequencedMap*, which is implemented by *LinkedHashMap*, *TreeMap*, and *ConcurrentSkipListMap* and provides the methods *firstEntry()*, *lastEntry()*, *pollFirstEntry()*, *pollLastEntry()*, *putFirst(…)*, *putLast(…)*, and *reversed()*.

The Latest Features in Java 21 (Part 2)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.
You can find the non-finalized JDK Enhancement Proposals from Java 21 on pages 2 and 3.

Pattern Matching for switch – JEP 441 🌟

Pattern Matching for switch makes it possible to formulate switch statements and expressions over any object. You can now also use patterns, qualified enum constants (preceded by a type name) and *null* as case labels. An example:

```
Object o = . . .
switch (o) {
  case String s when s.length() > 5 -> System.out.println("o is a long String, its length is " + s.length());
  case String s                     -> System.out.println("o is a short String: " + s);
  case Month.JANUARY                -> System.out.println("o is the month of January");
  case null                         -> System.out.println("o is null");
  default                           -> System.out.println("o is something else");
}
```

When using one of the new switch features (pattern, qualified enum constant or *case null* label), the switch must be exhaustive, i.e., it must cover all conceivable cases or contain a *default* label.

This enables highly expressive code, especially in combination with sealed classes. Let *Shape* be a sealed class that only allows the child classes *Rectangle* and *Circle*. Then the following switch statement is considered exhaustive:

```
Shape shape = . . .
switch (shape) {
  case Rectangle r -> System.out.printf("Rectangle: top left = %s; bottom right = %s%n", r.tl(), r.br());
  case Circle c    -> System.out.printf("Circle: center = %s; radius = %f%n", c.center(), c.radius());
}
```

If you now extend *Shape* with another child class, the compiler will no longer recognize this switch as complete, and you will be forced to extend it with the new class. This eliminates many potential sources of error.

Record Patterns – JEP 440 🌟

A *record pattern* can be used to deconstruct a Java record into its components during pattern matching. Let *Position* be a record with x and y coordinates and *Path* a record with a start and end position. Then, for example, the following is possible:

```
switch (o) {
  case Position(int x, int y)         -> System.out.printf("o is a position: %d/%d", x, y);
  case Path(Position(int x1, int y1),
            Position(int x2, int y2)) -> System.out.printf("o is a path: %d/%d -> %d/%d", x1, y1, x2, y2);
  case String s                       -> System.out.printf("o is a string: %s%n", s);
  default                             -> System.out.printf("o is something else: %s", o);
}
```

Generational ZGC – JEP 439 ⭐

The "Weak Generational Hypothesis" states that most objects die shortly after they have been created, and that objects that have survived several GC cycles usually stay alive even longer.

A so-called "generational garbage collector" makes use of this hypothesis by dividing the heap into two logical areas: a "young generation", in which new objects are created, and an "old generation", into which objects that have reached a certain age are moved. As objects in the old generation are very likely to become even older, the performance of an application can be increased by the garbage collector scanning the old generation less frequently.

Previously, the Z Gargabe Collector, or ZGC for short, made no distinction between "old" and "new" objects. This changes in Java 21 if you activate the generational ZGC with the following VM options: *-XX:+UseZGC -XX:+ZGenerational*

Other Changes in Java 21

Generational Shenandoah (Experimental) (JEP 404) – In addition to the Z Garbage Collector, the "Shenandoah Garbage Collector" introduced in Java 15 will also support generations in the future.

Deprecate the Windows 32-bit x86 Port for Removal (JEP 449) – The 32-bit version of Windows 10 is hardly used anymore, support ends in October 2025, and Windows 11 – on the market since October 2021 – has never been published in a 32-bit version. Accordingly, the 32-bit Windows port has been marked as "deprecated for removal".

Prepare to Disallow the Dynamic Loading of Agents (JEP 451) – Loading agents at runtime, e.g., by a profiler, represents a significant security risk and must be explicitly permitted in future.

Key Encapsulation Mechanism API (JEP 452) – Key Encapsulation Mechanism (KEM) is a modern encryption technology that enables the exchange of symmetric keys via an asymmetric encryption method.

The Latest Features in Java 20

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 20 – Scoped Values (JEP 429), Record Patterns (JEP 432), Pattern Matching for switch (JEP 433), Foreign Function & Memory API (JEP 434), Virtual Threads (JEP 436), Structured Concurrency (JEP 437), and Vector API (JEP 438) – have been finalized in Java 21 and Java 22 or are still in the preview stage as of Java 22.

No final JDK Enhancement Proposals (JEPs) were implemented in Java 20.
In addition to the preview JEPs mentioned above, Java 20 contains the following notable changes:

Javac Warns about Type Casts in Compound Assignments with Possible Lossy Conversions ⭐

Let's start with a little quiz. What is the difference between the following two statements?

```
a += b;
a = a + b;
```

Most Java developers will say: There is none. But that is wrong. If, for example, *a* is a *short*, and *b* is an *int*, then the second line will result in a compiler error:

```
java: incompatible types: possible lossy conversion from int to short
```

That is because *a* + *b* results in an *int*, which may not be assigned to the *short* variable *a* without an explicit cast.

The first line, on the other hand, is permitted, as the compiler inserts an *implicit* cast in a so-called "compound assignment". If *a* is a *short*, then *a* += *b* is equivalent to:

```
a = (short) (a + b);
```

However, when casting from *int* to *short*, the upper 16 bits are truncated. Information is therefore lost, as the following example shows:

```
short a = 30_000;
int b = 50_000;
a += b;
System.out.println("a = " + a);
```

The program does not print 80000 (hexadecimal 0x13880), but 14464 (hexadecimal 0x3880).

To warn developers of this potentially undesirable behavior, a corresponding compiler warning was (finally!) added in Java 20.

Idle Connection Timeouts for HTTP/2

The system property *jdk.httpclient.keepalive.timeout* can be used to set how long inactive HTTP/1.1 connections are kept open. As of Java 20, this property also applies to HTTP/2 connections.

Furthermore, the system property *jdk.httpclient.keepalive.timeout.h2* has been added, with which you can overwrite this value specifically for HTTP/2 connections.

HttpClient Default Keep Alive Time is 30 Seconds

If the aforementioned system property *jdk.httpclient.keepalive.timeout* is not defined, a default value of 1,200 seconds applied until Java 19. In Java 20, the default value was reduced to 30 seconds.

IdentityHashMap's Remove and Replace Methods Use Object Identity

IdentityHashMap is a special *Map* implementation in which keys are not considered equal if the *equals()* method returns *true*, but if the key objects are identical, i.e., comparing them using the == operator yields *true*.

However, when the default methods *remove(Object key, Object value)* and *replace(K key, V oldValue, V newValue)* were added to the *Map* interface in Java 8, the JDK developers forgot to overwrite these methods in *IdentityHashMap* to use == instead of *equals()*.

This error was corrected in Java 20, eight and a half years later. The fact that the bug was not noticed for so long is a sign that *IdentityHashMap* is little used and may contain other bugs.

The Latest Features in Java 19

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features released in Java 19 – Record Patterns (JEP 405), Foreign Function & Memory API (JEP 424), Virtual Threads (JEP 425), Vector API (JEP 426), Pattern Matching for switch (JEP 427), and Structured Concurrency (JEP 428) – have been finalized in Java 21 and Java 22 or are still in the preview stage as of Java 22.

New Methods to Create Preallocated HashMaps ⭐

HashMap provides a constructor with which you can define the initial capacity of the map, e.g., like this:

```
Map<String, String> map = new HashMap<>(120);
```

Intuitively, one would think that this *HashMap* offers space for 120 entries. However, this is not the case!

That is because the *HashMap* is initialized with a default load factor of 0.75. This means that, as soon as the *HashMap* is 75% full, it gets rebuilt ("rehashed") with twice the size. This is to ensure that the elements are distributed as evenly as possible across the *HashMap*'s buckets and that no bucket contains more than one element.

The *HashMap* initialized with a capacity of 120 can, therefore, only hold 120 × 0.75 = 90 entries.

To create a *HashMap* for 120 entries, you previously had to calculate the capacity by dividing the number of mappings by the load factor: 120 ÷ 0.75 = 160. A HashMap for 120 entries, therefore, had to be created as follows:

```
Map<String, Integer> map = new HashMap<>(160);
```

Java 19 makes it easier for us – we can now write the following instead:

```
Map<String, Integer> map = HashMap.newHashMap(120);
```

The *newHashMap(...)* method has also been implemented in *LinkedHashMap* and *WeakHashMap*. Take a look at the source code of the new methods – you will see that they perform exactly the same calculation that we performed manually before.

Additional Date-Time Formats ⭐

Using the *DateTimeFormatter.ofLocalizedDate(...)*, *ofLocalizedTime(...)*, and *ofLocalizedDateTime(...)* methods and a subsequent call to *withLocale(...)*, we have been able to generate date and time formatters since Java 8. We can control the exact format using the *FormatStyle* enum, which can take the values *FULL*, *LONG*, *MEDIUM* and *SHORT*.

In Java 19, the *ofLocalizedPattern(String requestedTemplate)* method was added, which we can use to define more flexible formats. Here is an example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedPattern("yMMM");
System.out.println("US:      " + formatter.withLocale(Locale.US).format(LocalDate.now()));
System.out.println("Germany: " + formatter.withLocale(Locale.GERMANY).format(LocalDate.now()));
System.out.println("Japan:   " + formatter.withLocale(Locale.JAPAN).format(LocalDate.now()));
```

The code leads to the following output:

```
US:      Jan 2024
Germany: Jan. 2024
Japan:   2024年1月
```

New System Properties for System.out and System.err

If the automatic detection of the configured character set of the console/terminal does not work when printing to *System.out* or *System.err*, you can set the character set as of JDK 19 via the following VM options, for instance, to UTF-8:

```
-Dstdout.encoding=utf-8 -Dstderr.encoding=utf-8
```

If you don't want to do this every time you start the program, you can also configure these settings globally by defining the following environment variable (yes, it starts with an underscore):

```
_JAVA_OPTIONS="-Dstdout.encoding=utf-8 -Dstderr.encoding=utf-8"
```

Linux/RISC-V Port – JEP 422

Due to the increasing use of RISC-V hardware, a Linux port for this architecture will be made available for the corresponding architecture from Java 19 onwards.

The Latest Features in Java 18

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 18 – Vector API (JEP 417), Foreign Function & Memory API (JEP 419), and Pattern Matching for switch (JEP 420) – have been finalized in Java 21 and Java 22 or are still in the preview stage in Java 22.

UTF-8 by Default – JEP 400 🌟

The character encoding determines how strings are converted into bytes and vice versa in numerous JDK methods, e.g., in *FileReader*, *FileWriter*, *InputStreamReader*, *InputStreamWriter*, *Formatter*, and *Scanner*. Until Java 17, the standard character encoding depended on the operating system and language, which led to numerous problems, e.g., if a text file was written on one operating system and read on another without explicitly specifying the character encoding.

The standard character encoding could be overwritten using the system property *-Dfile.encoding*, but newer methods such as *Files.readString(...)* ignore the character encoding configured this way and always use UTF-8.

As of Java 18, the standard character encoding on all platforms and languages is UTF-8, so that the problems described above should be a thing of the past.

Code Snippets in Java API Documentation – JEP 413 🌟

As of Java 18, we can insert code examples in JavaDoc comments with the *@snippet* tag. Parts of the code can be marked with *@highight* or linked with *@link*, as in the example on the right.

You can also refer to marked code locations in other Java files. You can find more details in the Java 18 article.

```
/**
 * {@snippet :
 * // @highlight region regex="\bwrite.*?\b" type="highlighted":
 * // @link regex="\bBufferedWriter\b" target="java.io.BufferedWriter":
 * try (BufferedWriter writer = Files.newBufferedWriter(path)) {
 *  writer.write(text);
 * }
 * // @end
 * }
 */
```

Internet-Address Resolution SPI – JEP 418 ⭐

To find the IP address for a host name using *InetAddress.getByName(...)*, for example, the operating system's resolver is called, i.e., the *hosts* file is usually consulted first and then the configured DNS servers. Calling the corresponding operating system function is blocking and sometimes takes quite a long time. In addition, we cannot adjust the hostname resolution in tests, for example, in order to redirect requests to a test server.

The *Internet Address Resolution SPI* makes it very easy to replace the standard hostname resolver with your own. You can find out exactly how this works in the Java 18 article.

Simple Web Server – JEP 408

Java 18 comes with a rudimentary web server, which you can start with the command on the right.

```
$ jwebserver -b 127.0.0.100 -p 4444 -d /tmp
Serving /tmp and subdirectories on 127.0.0.100 port 4444
URL http://127.0.0.100:4444/
```

Use *-b* to specify the IP address on which the web server should accept connections, and *-p* to specify the port. If you omit this information, the web server will listen to 127.0.0.1:8000. The *-d* parameter specifies the directory that the web server should deliver. By default, this is the current directory. You can also start the web server from Java code (in fact, the *jwebserver* command does just that):

```
SimpleFileServer.createFileServer(
    new InetSocketAddress("127.0.0.10", 4444), Path.of("/tmp"), OutputLevel.INFO).start();
```

Reimplement Core Reflection with Method Handles – JEP 416

Java has two reflection mechanisms – *core reflection* and *method/variable handles*:

```
Field field = String.class.getDeclaredField("value");  // ← Core reflection
VarHandle handle = MethodHandles.privateLookupIn(String.class, MethodHandles.lookup())  // ← Method handles
                                .findVarHandle(String.class, "value", byte[].class);
```

Maintaining both mechanisms represents a considerable amount of extra work. For this reason, the code of the reflection classes *java.lang.reflect.Method*, *Field*, and *Constructor* was re-implemented in Java 18 based on method handles.

Deprecate Finalization for Removal – JEP 421

Finalization, which has existed since Java 1.0, is insecure, error-prone and can lead to performance problems. As alternatives, *try-with-resources* was introduced in Java 7, and the *Cleaner API* in Java 9. Accordingly, the *finalize()* methods in *Object* and numerous other classes of the JDK class library were marked as "deprecated" in Java 9. As of Java 18, the *finalize()* methods are marked as "deprecated for removal".

The Latest Features in Java 17 (Part 1)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Sealed Classes – JEP 409 🌟

Until now, there were only limited options for restricting the inheritability of a class – by making it *final* or marking it as non-*public*. In Java 17, we can use the keywords *sealed*, *non-sealed*, and *permits* to control inheritability in a much more granular way. In the example on the right, the *Shape* interface may only be implemented by the *Circle* and *Square* classes.

```
public sealed class Shape permits Circle, Square {}
public final class Circle extends Shape {}
public final class Square extends Shape {}
```

Sealed classes enable, for example, exhaustiveness analysis in *Pattern Matching for switch*, introduced in Java 21. You can find further use cases and characteristics at https://www.happycoders.eu/java/sealed-classes/.

Strongly Encapsulate JDK Internals – JEP 403 🌟

From Java 9 to 15, you have probably seen the warning *"WARNING: An illegal reflective access operation has occurred"*. The reason for this is that for a transitional period after the introduction of the module system in Java 9, deep reflection on code that existed before Java 9 was permitted in the so-called "relaxed strong encapsulation" mode. In Java 16, the default mode was changed to "strong encapsulation", which led to an error and program termination instead of a warning. With the VM option *--illegal-access=permit*, you could restore the previous behavior in Java 16.

In Java 17, only "strong encapsulation" mode is available; the VM option no longer has any effect and generates a warning. Reflective access must now be explicitly allowed for specific modules and packages using the following VM option:

```
--add-opens <name of the called module>/<package name>=<name of the calling module or ALL-UNNAMED>
```

Add java.time.InstantSource ⭐

The *java.time.Clock* class is extremely useful for writing tests that check time-dependent functionality. If you inject *Clock* into the application classes via dependency injection, for example, you can mock it in tests and set a specific time for the test using *Clock.fixed(...)*, for example.

However, as *Clock* provides the *getZone()* method, you always have to think about the specific time zone with which you instantiate a *Clock* object.

To enable time zone-independent time sources, the *java.time.InstantSource* interface was extracted from *Clock* in Java 17. It only provides the *instant()* and *millis()* methods for querying the time, with *millis()* being implemented as a default method.

The example at the top right shows a method that measures the execution time of a *Runnable*. In production, we can instantiate *Timer* with the system clock:

```
Timer timer = new Timer(Clock.systemDefaultZone());
```

We can test the method by mocking *InstantSource*, having the *instant()* method return two constant values and comparing the return value of *measure(...)* with the difference between these values (see test method on the right).

```
public class Timer {
  private final InstantSource instantSource;

  public Timer(InstantSource instantSource) {
    this.instantSource = instantSource;
  }

  public Duration measure(Runnable runnable) {
    Instant start = instantSource.instant();
    runnable.run();
    Instant end = instantSource.instant();
    return Duration.between(start, end);
  }
}
```

```
@Test
void shouldReturnDurationBetweenStartAndEnd() {
  var instantSource = mock(InstantSource.class);
  when(instantSource.instant())
      .thenReturn(ofEpochMilli(1640033566000L))
      .thenReturn(ofEpochMilli(1640033566075L));

  Timer timer = new Timer(instantSource);
  Duration duration = timer.measure(() -> {});
  assertThat(duration).isEqualTo(ofMillis(75));
}
```

Hex Formatting and Parsing Utility ⭐

Java 17 provides the new class *java.util.HexFormat*, which provides a standardized API for displaying and parsing hexadecimal numbers. *HexFormat* supports all primitive integers (*int*, *byte*, *char*, *long*, *short*) as well as byte arrays – but not floating point numbers. Here is an example:

```
HexFormat hf = HexFormat.of();
System.out.println(hf.toHexDigits('A'));            // → 20ac
System.out.println(hf.toHexDigits((byte) 100));     // → 64
System.out.println(hf.toHexDigits((short) 10_000)); // → 2710
System.out.println(hf.toHexDigits(1_000_000_000));  // → 3b9aca00
System.out.println(hf.withUpperCase().formatHex(new byte[] {-54, -2, -70, -66})); // → CAFEBABE
System.out.println(hf.withDelimiter(" ").formatHex("Happy Coding!".getBytes()));  // → 48 61 70 70 79 ...
```

You can find additional methods for displaying and parsing hexadecimal strings in the *HexFormat* JavaDoc.

The Latest Features in Java 17 (Part 2)

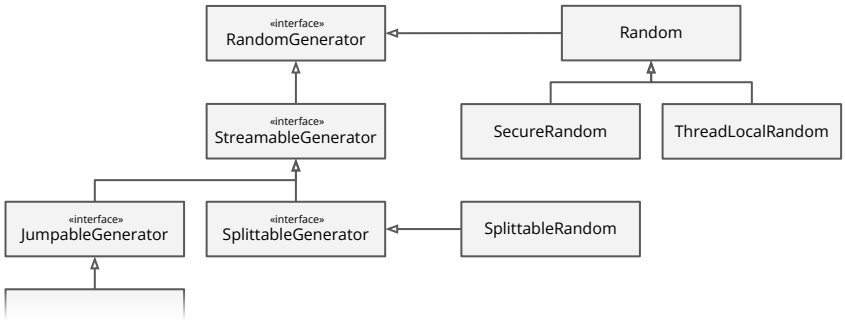
You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 17 – Pattern Matching for switch (JEP 406), Foreign Function & Memory API (JEP 412), and Vector API (JEP 414) – have been finalized in Java 21 and Java 22 or are still in the preview stage in Java 22.

Enhanced Pseudo-Random Number Generators – JEP 356 ⭐

In Java 17, a set of inheriting interfaces was introduced for existing and new random number generators, so that the use of interfaces makes it much easier than before to exchange specific algorithms.

The methods common to all random number generators, such as *nextInt()* and *nextDouble()*, are defined in *RandomGenerator*. If you only need these methods, you should always use this interface in future.



Context-Specific Deserialization Filters – JEP 415

The deserialization of objects poses a significant security risk. Attackers can construct objects via the data stream to be deserialized, which they can ultimately use to execute arbitrary code in arbitrary classes (available on the classpath). In Java 9, global and *ObjectInputStream*-specific deserialization filters were introduced, which can be used to specify which classes may be deserialized and which may not.

Java 17 extends this protection mechanism to include *context-specific* deserialization filters, which can be defined for a specific thread, class, module or third-party library, for example.

Unified Logging Supports Asynchronous Log Flushing

With asynchronous logging, log messages are first written to a buffer by the application thread. A separate I/O thread then forwards them to the configured output (console, file, or network). This means that the application thread does not have to wait for the corresponding I/O subsystem to process the message.

While most logging frameworks support asynchronous logging, log messages from the JVM itself (e.g., those from the garbage collector) could previously only be written out synchronously.

As of Java 17, asynchronous logging can be activated for the JVM itself via the VM option *-Xlog:async*. The size of the log buffer can be defined via *-XX:AsyncLogBufferSize=<size in bytes>*.

Other Changes in Java 17

Restore Always-Strict Floating-Point Semantics (JEP 306) – Before Java 1.2, floating point operations were executed "strictly" by default, i.e., they led to identical results on all architectures. Java 1.2 transitioned to the then faster "standard floating point semantics" of the processor architecture. If you wanted to continue to perform strict calculations, you had to mark a class with the *strictfp* keyword. As modern CPUs perform strict operations without loss of performance, strict calculations are used again by default from Java 17 onwards. The *strictfp* keyword is now superfluous and results in a compiler warning.

New macOS Rendering Pipeline (JEP 382) – The Swing rendering pipeline for macOS has been switched to the Metal framework, which replaced the OpenGL library previously used under macOS in 2018.

macOS/AArch64 Port (JEP 391) – Apple announced that it will be converting Macs from x64 to AArch64 CPUs in the long term. Accordingly, an AArch64 port will be provided.

Deprecate the Applet API for Removal (JEP 398) – Java applets are no longer supported by any modern web browser and were already marked as "deprecated" in Java 9. In Java 17, they are marked as "deprecated for removal".

Remove RMI Activation (JEP 407) – RMI Activation allows objects that have been destroyed on the target JVM to be instantiated again as soon as they are accessed. As this feature was time-consuming to maintain and rarely used, it was removed.

Remove the Experimental AOT and JIT Compiler (JEP 410) – The Graal compiler provided in Java 9 and 10 as an AOT or JIT compiler was only used a little and was removed from the Oracle JDK in Java 16 and also from the OpenJDK in Java 17.

Deprecate the Security Manager for Removal (JEP 411) – The Security Manager should protect the computer from downloaded Java applets. Just like Java applets, the Security Manager has now also been marked as "deprecated for removal".

The Latest Features in Java 16

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The previews published in Java 16 – Vector API (JEP 338), Foreign Linker API (JEP 389), Foreign-Memory Access API (JEP 393), and Sealed Classes (JEP 397) – will be finalized in Java 17 and Java 22 or are still in the preview stage in Java 22.

Pattern Matching for instanceof – JEP 394 🌟

As of Java 16, we no longer have to explicitly cast a variable to the target type after an *instanceof* check. Instead, we can bind the variable to a new variable of the corresponding type, e.g., like this:

```
if (o instanceof String s && !s.isEmpty()) System.out.println("o in capital letters: " + s.toUpperCase());
```

Records – JEP 395 🌟

Java records offer a compact notation method for defining classes with only final fields.

Find out what particularities you need to be aware of when working with records at https://www.happycoders.eu/java/java-records/.

```
record Position(int x, int y) {}
record Path(Point start, Point end) {}

Path p = new Path(new Pos(0,8), new Pos(1,5));
int startX = p.start().x();
```

Strongly Encapsulate JDK Internals by Default – JEP 396 🌟

You have probably encountered the warning *"WARNING: An illegal reflective access operation has occurred"*. This is because for a transitional period after the introduction of the Java module system in JDK 9, deep reflection on code that already existed before Java 9 was allowed by default in the so-called "relaxed strong encapsulation" mode.

In Java 16, the default mode is changed to "strong encapsulation", which now leads to an error and program termination instead of a warning. With the VM option *--illegal-access=permit*, you can temporarily restore the previous behavior. However, the more sustainable solution is to explicitly allow reflective access with the following VM option:

```
--add-opens <name of the called module>/<package name>=<name of the calling module or ALL-UNNAMED>
```

New Stream Methods 🌟

The new *Stream.toList()* method is a short form for the frequently used *Stream.collect(Collectors.toUnmodifiableList())*.

Stream.mapMulti(...) is an alternative to *flatMap(...)* that does not require the creation of temporary streams. You can find out exactly how *mapMulti(...)* works in the article on Java 16.

Warnings for Value-Based Classes – JEP 390 ⭐

The following classes were annotated as *@ValueBased* and their constructors marked as "deprecated for removal":

- all wrapper classes of the primitive data types (*Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean*, and *Character*),
- *Optional* and its primitive variants,
- numerous classes of the date/time API, such as *LocalDateTime*,
- the collections created by *List.of()*, *Set.of()*, and *Map.of()*.

This is a preparation for *Value Objects* developed in Project Valhalla, i.e., objects without identity and header.

Migrate from Mercurial to Git + Migrate to GitHub – JEPs 357 + 369 ⭐

With Java 16, the JDK code was migrated from the version management system Mercurial to the much more widely used Git. As hosting platform, GitHub was chosen. The JDK source code has since been available here: https://github.com/openjdk/

Other Changes in Java 16

Enable C++14 Language Features (JEP 347) – The C++ part of the JDK was previously limited to the C++98/03 language specification – a standard that is over 20 years old. In JDK 16, support is increased to C++14.

ZGC: Concurrent Thread-Stack Processing (JEP 376) – The so-called "thread stack processing", which in ZGC was previously carried out in so-called safepoints, i.e., in stop-the-world phases, runs parallel to the running application from Java 16 onwards.

Unix-Domain Socket Channels (JEP 380) – As of version 16, Java supports Unix domain sockets, which are used for inter-process communication (IPC) within a host and are significantly faster than TCP sockets.

Alpine Linux Port (JEP 386) – The JDK is extended by a port for Alpine Linux, which is highly popular in cloud environments.

Elastic Metaspace (JEP 387) – In Java 16, the memory footprint of the metaspace, i.e., the memory area in which class metadata is stored, has been reduced. Metaspace memory is also returned to the operating system more quickly.

Windows/AArch64 Port (JEP 388) – Windows on the ARM64/AArch64 processor architecture has become an important platform. Accordingly, a port is also provided for this architecture, based on the existing Linux/AArch64 port.

Packaging Tool (JEP 392) – The *jpackage* tool packages a Java application together with the Java runtime environment (i.e., the JVM and the required modules of the JDK class library) as an installation package for various operating systems.

The Latest Features in Java 15

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 15 – Sealed Classes (JEP 360), Pattern Matching for instanceof (JEP 375), Foreign-Memory Access API (JEP 383) and Records (JEP 384) – will be finalized in Java 16, 17 and 22.

Text Blocks – JEP 378 🌟

As of Java 15, you can code multi-line strings as shown on the right, delimited by three quotation marks on each side. You no longer need to escape individual quotation marks. The indentation common to all lines is removed.

You can find details at https://www.happycoders.eu/java/java-text-blocks/.

```
String sql = """
    SELECT id, title, text FROM Article
     WHERE category = "Java"
  ORDER BY title""";
```

New String and CharSequence Methods 🌟

The *String* and *CharSequence* classes have been extended in Java 15 to include the following methods:

- *String.formatted(Object... args)* – placeholders in strings can now also be replaced as follows, for example:
  `String message = "User %,d %s logged in at %s".formatted(userId, username, ZonedDateTime.now())`
- *String.stripIndent()* – removes the same number of spaces from the beginning of each line of the string, so that the line with the previously lowest number of spaces starts at the beginning of the line. Also removes all trailing spaces.
- *String.translateEscapes()* – replaces escaped escape sequences such as "\\n" with evaluated escape sequences such as "\n".
- *CharSequence.isEmpty()* – checks whether a character sequence, e.g., a *String* or *StringBuilder*, is empty.

ZGC: A Scalable Low-Latency Garbage Collector – JEP 377 ⭐

The Z Garbage Collector, ZGC for short, promises not to exceed pause times of 10 ms while reducing the overall throughput of the application by no more than 15% compared to the G1 garbage collector (the reduction in throughput is the cost of the low latency). ZGC supports heap sizes up to 16 TB. Just like G1, ZGC is region-based, NUMA-aware and returns unused memory to the operating system.

You can activate ZGC with the following VM option: *-XX:+UseZGC*

The VM option *-XX:SoftMaxHeapSize* can be used to configure a "soft" upper heap limit, which ZGC only exceeds if it is absolutely necessary to avoid an *OutOfMemoryError*.

Shenandoah: A Low-Pause-Time Garbage Collector – JEP 379

Likewise, Shenandoah promises minimal pause times regardless of the heap size. Shenandoah also returns unused memory to the operating system. However, there is currently no support for NUMA and *SoftMaxHeapSize*.

You can activate Shenandoah with the following VM option: *-XX:+UseShenandoahGC*

Other Changes in Java 15

Edwards-Curve Digital Signature Algorithm (EdDSA) (JEP 339) – EdDSA is a modern signature method that is faster than previous signature methods such as DSA and ECDSA while offering the same level of security.

Hidden Classes (JEP 371) – Frameworks such as Jakarta EE and Spring generate numerous classes dynamically at runtime. Until now, these classes could not be distinguished from regular classes and were therefore visible to all other classes. As of Java 15, these framework-specific classes can be defined as "hidden" and thus remain concealed from the rest of the application.

Remove the Nashorn JavaScript Engine (JEP 372) – The JavaScript engine "Nashorn" introduced in Java 8 was marked as "deprecated for removal" in Java 11 and is completely removed in Java 15.

Reimplement the Legacy DatagramSocket API (JEP 373) – The *java.net.DatagramSocket* and *java.net.MulticastSocket* APIs, which have existed since Java 1.0, consist of a mixture of outdated Java and C code that is difficult to maintain and extend. The code contains some bugs and cannot be easily adapted to virtual threads developed in Project Loom. Therefore, the old implementation has been replaced by a simpler, more modern, easier to maintain implementation.

Disable and Deprecate Biased Locking (JEP 374) – Biased locking is an optimization of thread synchronization that mainly benefits classes such as *Vector*, *Hashtable*, and *StringBuffer*, where every access is synchronized. As these classes are hardly used anymore, and biased locking is difficult to maintain, it is deactivated and marked as "deprecated".

Remove the Solaris and SPARC Ports (JEP 381) – The outdated Solaris and SPARC ports were marked as "deprecated" in Java 14 and have been removed from the JDK in Java 15.

Deprecate RMI Activation for Removal (JEP 385) – RMI (Remote Method Invocation) Activation allows objects that have been destroyed on the target JVM to be automatically instantiated again as soon as they are accessed. As this feature is complex to maintain and rarely used, it is marked as "deprecated".

Compressed Heap Dumps – As of Java 15, the *jcmd GC.heap_dump* command can save the heap dump in a ZIP-compressed way by using the option *-gz=<compression level>*. Level 1 is the fastest, level 9 the strongest compression.

The Latest Features in Java 14

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features released in Java 14 – Pattern Matching for instanceof (JEP 305), Packaging Tool (JEP 343), Records (JEP 359), ZGC on macOS / Windows (JEPs 364 + 365), Text Blocks (JEP 368), and Foreign-Memory Access API (JEP 370) – will be finalized in Java 15, 16 and 22.

Switch Expressions – JEP 361 🌟

As of Java 14, switch statements can be written in the so-called arrow notation. One or more constants are followed by an arrow and then a single instruction or a code block.

The error-prone fallthrough to the next label inherited from C no longer exists with this notation (see example 1).

Switch can now also be used as an expression and return a value. With a switch expression, for example, we can write a recursive Fibonacci function quite elegantly (see example 2).

Find more details about switch expressions at www.happycoders.eu/java/switch-expressions/.

Example 1:

```
switch (dayOfWeek) {
  case MONDAY, TUESDAY     -> System.out.print("Beginning");
  case WEDNESDAY, THURSDAY -> System.out.print("Middle");
  case FRIDAY              -> System.out.print("Almost!!!");
  case SATURDAY, SUNDAY    -> System.out.print("Weekend");
}
```

Example 2:

```
public static int fibonacci(int i) {
  return switch (i) {
    case 0, 1 -> i;
    default   -> fibonacci(i - 1) + fibonacci(i - 2);
  };
}
```

Helpful NullPointerExceptions – JEP 358 🌟

As of Java 14, a *NullPointerException* no longer only shows you in which line it occurred, but also which object in this line is *null*, for instance, like this:

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "Map.getValue()" because the return value of "Container.getMap()" is null
    at eu.happycoders.BusinessLogic.calculate(BusinessLogic.java:80)
```

Accounting Currency Format Support

In some countries, e.g., in the USA, negative numbers in accounting are not indicated by a minus sign, but by round brackets. Java 14 adds the so-called language tag extension "u-cf-account", which makes it possible to specify additional information in a *Locale* object as to whether it is used in the context of accounting:

```
NumberFormat nf           = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat nfAccounting = NumberFormat.getCurrencyInstance(Locale.forLanguageTag("en-US-u-cf-account"));
System.out.println("Normal:     " + nf          .format(-14.95));  // → -$14.95
System.out.println("Accounting: " + nfAccounting.format(-14.95));  // → ($14.95)
```

Other Changes in Java 14

Non-Volatile Mapped Byte Buffers (JEP 352) – You can use a *MappedByteBuffer* to map a file into memory so that it can then be read and modified using regular memory access operations. Modified data must be regularly transferred to the storage medium. For NVMs, more efficient methods are available than for conventional storage media. As of Java 14, you can specify the *ExtendedMapMode.READ_ONLY_SYNC* or *READ_WRITE_SYNC* modes when calling *FileChannel.map(...)*.

NUMA-Aware Memory Allocation for G1 (JEP 345) – On modern servers, the physical distance between the CPU core and the memory module plays an increasingly important role. The G1 Garbage Collector can use NUMA (Non-Uniform Memory Access) data to allocate objects on the memory node that is closest to the CPU core on which the object-generating thread is running. NUMA-aware memory allocation is only available for Linux and must be activated via the VM option *+XX:+UseNUMA*.

JFR Event Streaming (JEP 349) – The Java Flight Recorder (JFR) collects data about the JVM during the execution of an application and saves it in a file. The data can then be visualized with JDK Mission Control. As of Java 14, continuous monitoring is also possible. You can find out how this works in the Java 14 article.

Deprecate the Solaris and SPARC Ports (JEP 362) – Solaris and the SPARC processor architecture are no longer up to date. In order to be able to use development resources elsewhere, the corresponding ports are marked as "deprecated".

Remove the Concurrent Mark Sweep (CMS) Garbage Collector (JEP 363) – The Concurrent Mark Sweep (CMS) garbage collector was marked as "deprecated" in Java 9. Development resources are to be redistributed in favor of more modern garbage collectors such as G1GC (standard since Java 9) and ZGC. CMS is completely removed from the JDK in Java 14.

Deprecate the ParallelScavenge + SerialOld GC Combination (JEP 366) – The rarely used combination of GC algorithms, parallel for the young generation and serial for the old generation, activated by *-XX:+UseParallelGC -XX:-UseParallelOldGC*, is rarely used and requires a high maintenance effort. It is therefore marked as "deprecated".

Remove the Pack200 Tools and API (JEP 367) – The "Pack200" compression method was introduced in Java 5 in order to save as much bandwidth as possible in the early 2000s through more effective compression. In times of 100 MBit lines, the development costs are no longer in proportion to the benefits. Pack200 is therefore being removed in Java 14.

The Latest Features in Java 13

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 13 – Switch Expressions (JEP 354), Text Blocks (JEP 355), and ZGC: Uncommit Unused Memory (351) – will all be finalized in Java 15.

Dynamic CDS Archives – JEP 350

Application Class-Data Sharing (see Java 10) was previously quite complicated to set up. In addition, not only the classes of the application itself, but also all JDK classes in use were included in the shared archive file.

In Java 13, only the classes of the application are stored in the shared archive; the JDK classes are loaded from the basic archive *classes.jsa*, which has been supplied with the JDK since Java 12. The shared archive of a simple "Hello World" program can thus be reduced from 9 MB to 256 KB.

A shared archive for a "Hello World" application compiled to *target/helloworld.jar* with a main method in the class *eu.happycoders.helloworld.Main* can be created as follows, for example:

```
java -XX:ArchiveClassesAtExit=helloworld.jsa -classpath target/helloworld.jar eu.happycoders.helloworld.Main
```

To start the program using the shared archive, start it as follows:

```
java -XX:SharedArchiveFile=helloworld.jsa -classpath target/helloworld.jar eu.happycoders.helloworld.Main
```

Reimplement the Legacy Socket API – JEP 353

The *java.net.Socket* and *java.net.ServerSocket* APIs have existed since Java 1.0. The underlying code, a mixture of Java and C code, is difficult to maintain and extend – particularly with regard to Project Loom, which will introduce virtual threads in Java 21.

In Java 13, the old implementation is replaced by a more modern, more maintainable, and extendable implementation, which should be able to be adapted to virtual threads without further refactoring.

Buffer.slice()

You can use the new *Buffer.slice(int index, int length)* method to create a view of a subsection of a buffer (e.g. *ByteBuffer*, *IntBuffer*, *DoubleBuffer*) that starts at position *index* and contains *length* bytes.

New XxxBuffer.get() and put() Methods

Similarly, there are two new *get()* and *put()* methods in each of the concrete buffer classes *ByteBuffer*, *IntBuffer*, *DoubleBuffer*, etc., with which the data is not read/written at the current position of the buffer as before, but at the explicitly specified position. The following is an example of the new methods in the concrete class *ByteBuffer*:

- *get(int index, byte[] dst, int offset, int length)*
  transfers *length* bytes from this buffer at the position specified by *index* to the byte array *dst* at position *offset*.
- *get(int index, byte[] dst)*
  transfers data from this buffer at the position specified by *index* into the byte array *dst*. The number of bytes transferred corresponds to the length of the target array.
- *put(int index, byte[] src, int offset, int length)*
  transfers *length* bytes from the byte array *src* at position *offset* into this buffer at position *index*.
- *put(int index, byte[] src)*
  transfers all bytes from the byte array *src* into this buffer at position *index*.

The position of the buffer remains unchanged with all four methods.

FileSystems.newFileSystem()

You can use the *FileSystems.newFileSystem(Path path, ClassLoader loader)* method to create a pseudo file system whose content is mapped to a file (such as a ZIP or JAR file).

The method was overloaded in Java 13 with a variant that makes it possible to pass a provider-specific configuration of the file system via the *env* parameter: *FileSystems.newFileSystem(Path path, Map env, ClassLoader loader)*.

Furthermore, two variants have been added, each without the *loader* parameter. A class loader is only required if the so-called *FileSystemProvider* for the file type to be mapped is not registered in the JDK, but is to be loaded via the specified class loader. This is not necessary for common file types such as ZIP or JAR.

The Latest Features in Java 12

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

The preview features published in Java 12 – Switch Expressions (JEP 325) and Shenandoah: A Low-Pause-Time Garbage Collector (JEP 189) – will be finalized in Java 14 and Java 15.

New String and Files Methods 🌟

In Java 12, the following methods have been added to the *String* and *Files* classes:

- *String.indent(int n)* – indents the string on which *indent(...)* is called line by line by *n* spaces.
- *String.transform(Function<...> f)* – calls the function *f* with this string as a parameter and returns the result.
- *Files.mismatch()* – indicates the position of the first byte in which the files differ, or -1 for identical files.

Teeing Collector ⭐

The new "Teeing Collector" allows you to specify two stream collectors and a merge function that combines the results of the collectors. The following example determines the difference between the maximum and minimum from the stream elements:

```
long width = numbersStream.collect(
        Collectors.teeing(Collectors.minBy(Integer::compareTo), Collectors.maxBy(Integer::compareTo),
                          (min, max) -> (long) max.orElseThrow() - min.orElseThrow()));
```

Support for Compact Number Formatting ⭐

Using *NumberFormat.getCompactNumberInstance(...)*, we can create a formatter for so-called "compact number formatting". This is a human-readable form, such as "1 thousand" or "3 billion". Here is an example:

```
NumberFormat nfShort = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
NumberFormat nfLong  = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);
System.out.println(nfShort.format(1_000));           // ⟶ 1K
System.out.println(nfShort.format(3_456_789_000L));  // ⟶ 3B
System.out.println(nfLong.format(1_000));            // ⟶ 1 thousand
System.out.println(nfLong.format(3_456_789_000L));   // ⟶ 3 billion
```

Default CDS Archives – JEP 341

To activate *class-data sharing* (see Java 11), you previously had to execute the *java -Xshare:dump* command for each Java installation to generate the shared archive file *classes.jsa*.

As of Java 12, the JDK is delivered with this file on all 64-bit architectures, so that the execution of the above command is no longer necessary, and Java applications use the default CDS archive by default.

Abortable Mixed Collections for G1 – JEP 344

One of the goals of the G1 Gargabe Collector is to adhere to specified maximum pause times for "stop-the-world" phases – i.e., not to stop the application for longer than the specified time. G1 uses a heuristic for this, which can lead to the application being paused for longer than intended, especially if the application's behavior changes.

In Java 12, "mixed collections" (i.e., cleanup work that affects both the young and the old generation) were divided into a mandatory and an optional part. The mandatory part is executed uninterruptibly – and the optional part in incremental steps until the maximum pause time is reached. Meanwhile, G1 tries to adapt the heuristic so that it again determines collection sets that can be fully processed in the maximum pause time.

Promptly Return Unused Committed Memory from G1 – JEP 346

In cloud environments where you pay for the memory you actually use, it is desirable for the garbage collector to quickly return unused memory to the operating system. The G1 Garbage Collector can return memory, but only during the garbage collection phase. However, if the heap allocation or the current rate of object allocation does not trigger a garbage collection cycle, no memory is returned.

As of Java 12, you can use the VM option *-XX:G1PeriodicGCInterval=<milliseconds>* to specify an interval at which GC cycles are to be executed, regardless of whether they are necessary or not.

Other Changes in Java 12

Microbenchmark Suite (JEP 230) – Microbenchmarks, which regularly measure the performance of the JDK class library, were previously managed as a separate project. In Java 12, the code for these benchmarks was moved to the JDK source code.

JVM Constants API (JEP 334) – The constant pool of a *.class* file stores strings and the names of referenced classes and methods, among other things. The JVM Constants API is intended to make it easier to write Java programs that manipulate JVM bytecode. To this end, the API provides new interfaces and classes with which the constant pool can be managed.

One AArch64 Port, Not Two (JEP 340) – There are two ports for 64-bit ARM CPUs in the JDK: "arm64" from Oracle and "aarch64" from Red Hat. In Java 12, Oracle's port is removed in order to consolidate development resources.

The Latest Features in Java 11 (Part 1)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Launch Single-File Source-Code Programs – JEP 330 🌟

One of the most useful enhancements in Java 11 is that Java programs consisting of only a single file can be started without explicit compilation. Save the program on the right in the file *Hello.java*. You can then run it as follows without having to compile it first with *javac*:

```
$ java Hello.java
```

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Local-Variable Syntax for Lambda Parameters – JEP 323 🌟

A popular example for the use of a lambda expression is:

```
map.computeIfAbsent(key, (String k) -> new ArrayList<>()).add(value);
```

Java can derive the type of the lambda parameter *k* from the context, so you can omit it:

```
map.computeIfAbsent(key, k -> new ArrayList<>()).add(value);
```

As of Java 11, you can also specify *var* instead of a concrete type:

```
map.computeIfAbsent(key, (var k) -> new ArrayList<>()).add(value);
```

What is the point of specifying *var* if you can do without it? The reason for this is that only concrete types and *var* may be annotated, e.g., as follows:

```
map.computeIfAbsent(key, (@NotNull String k) -> new ArrayList<>()).add(value);
map.computeIfAbsent(key, (@NotNull var k)    -> new ArrayList<>()).add(value);
```

We can replace a particularly long, annotated type name with *var* – but we must not omit it.

HTTP Client (Standard) – JEP 321 🌟

With the new *HttpClient* class, Java 11 enables more convenient access to HTTP APIs, e.g., as follows:

```
try (HttpClient client = HttpClient.newHttpClient()) {
  HttpRequest request = HttpRequest.newBuilder().uri(URI.create(url)).GET().build();
  HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
  if (response.statusCode() == 200) { String responseBody = response.body(); . . . }
  else throw new IOException("HTTP error " + response.statusCode());
}
```

New Collection.toArray() Method 🌟

To convert a collection into an array, we previously had to pass an array of the target type to the *toArray(...)* method. If this array is sufficiently large, it is filled with the values of the collection; otherwise a new array of the same type is created. Since Java 11, we can also pass a method reference to the array's constructor to the method, e.g., like this:

```
String[] stringArray = stringList.toArray(String[]::new);
```

New String Methods 🌟

The *String* class has been extended in Java 11 to include the following methods:

- *String.strip()*, *stripLeading()*, *stripTailing()* – remove all leading and trailing or only the leading or only the trailing whitespaces from a string. In contrast to *trim()*, which only removes characters whose code point is U+0020 or less, *strip()* removes all characters marked as "whitespace" in the Unicode character set.
- *String.isBlank()* – returns *true* if a string consists only of characters marked as "whitespace".
- *String.repeat(int count)* – creates a new string by repeating the string on which the method is called *count* times.
- *String.lines()* – splits a string at all line breaks and returns a stream of all lines.

Files.readString() and writeString() 🌟

Previously, several lines of Java code (or a third-party library) were required to read a file into a string or write a string into a file. As of Java 11, you can use the *Files.writeString(...)* and *Files.readString(...)* methods.

Path.of() 🌟

As of Java 11, you can create a *Path* object using the static default method *of(...)* of the *Path* interface, e.g., like this:

```
Path.of("foo/bar/baz");             // ← Relative path foo/bar/baz
Path.of("foo", "bar", "baz");       // ← Relative path foo/bar/baz
Path.of("/", "foo", "bar", "baz");  // ← Absolute path /foo/bar/baz
```


The Latest Features in Java 11 (Part 2)

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Remove the Java EE and CORBA Modules – JEP 320 ⭐

The following modules were removed from the JDK in Java 11: *java.xml.ws (JAX-WS)*, *java.xml.bind (JAXB)*, *java.activation (JAF)*, *java.xml.ws.annotation (common annotations)*, *java.corba (CORBA)*, *java.transaction (JTA)*, *java.se.ee* (aggregator module for the six modules mentioned before), *jdk.xml.ws* (tools for JAX-WS), and *jdk.xml.bind* (tools for JAXB).

The technologies mentioned were originally developed for the Java EE platform and were integrated into the Standard Edition "Java SE" in Java 6. In Java 9, they were marked as "deprecated", and in Java 11, they were removed for good.

Should you miss these libraries when upgrading to Java 11, you can pull them back into your project using Maven, for example.

Flight Recorder – JEP 328 ⭐

Java Flight Recorder (JFR for short) can record numerous JVM data during the execution of an application and make it available in a file for subsequent analysis by JDK Mission Control. JFR has long existed as a commercial feature in the Oracle JDK. From Java 11 onwards, it is part of the OpenJDK and can be freely used.

Epsilon: A No-Op Garbage Collector – JEP 318 ⭐

Like any other garbage collector, the Epsilon garbage collector manages the allocation of objects. However, unlike all other GCs, it never releases the objects again. This can be helpful for performance tests, extremely short-lived applications, and the full elimination of latencies. Epsilon GC is activated via the VM option *-XX:+UseEpsilonGC*.

Nest-Based Access Control – JEP 181

The Java Language Specification (JLS) has always allowed access to private fields and methods of inner classes. As the Java Virtual Machine (JVM) did *not* previously allow this, the Java compiler (up to Java 10) inserted so-called "synthetic access methods" with the standard visibility "package-private" when accessing these private fields and methods. As a result, these fields and methods could then be accessed from the entire package. The compiler issued a warning accordingly.

In JDK 11, access control of the JVM was extended so that access to private members of inner classes is possible without those synthetic accessors.

Low-Overhead Heap Profiling – JEP 331

Heap dumps are an important debugging tool for analyzing the objects located on the heap. Until now, however, such tools have not been able to determine where in the program code such an object was created.

In Java 11, the Java Virtual Machine Tool Interface (JVMTI) – the interface through which the tools obtain data about the running application – has been expanded to include the option of collecting stack traces of all object allocations. Heap analysis tools can display this additional information, making it much easier for us to analyze problems.

Other Changes in Java 11

Dynamic Class-File Constants (JEP 309) – The *.class* file format is extended by the *CONSTANT_Dynamic* constant, "which gives language developers and compiler implementers broader options for expressiveness and performance."

Improve Aarch64 Intrinsics (JEP 315) – Intrinsics are used to execute architecture-specific assembler code instead of Java code. Intrinsics for the AArch64 platform have been improved, and new intrinsics have been added.

Key Agreement with Curve25519 and Curve448 (JEP 324) – Key exchange protocols with elliptical curves, such as "Curve25519" and "Curve448", enable particularly fast encryption and decryption of symmetric keys.

Unicode 10 (JEP 327) – Unicode support has been upgraded to version 10.0, i.e., classes such as *String* and *Character* can handle the newly introduced code blocks and characters.

ChaCha20 and Poly1305 Cryptographic Algorithms (JEP 329) – The JDK is extended by the two cryptographic algorithms mentioned above. These are used, for example, by security protocols such as TLS.

Transport Layer Security (TLS) 1.3 (JEP 332) – In addition to Secure Socket Layer (SSL) 3.0, Transport Layer Security (TLS) 1.0 to 1.2, and Datagram Transport Layer Security (DTLS) 1.0 and 1.2, Java also supports TLS 1.3 from version 11 onwards.

ZGC: A Scalable Low-Latency Garbage Collector (Experimental) (JEP 333) – A new GC that will be finalized in Java 15.

Deprecate the Nashorn JavaScript Engine (JEP 335) – The JavaScript engine "Nashorn" introduced in Java 8 is marked as "deprecated for removal" in Java 11. The reason for this is the rapid development speed of ECMAScript (the standard behind JavaScript) and the *node.js* engine, which makes further development of Nashorn too costly.

Deprecate the Pack200 Tools and API (JEP 336) – The "Pack200" compression algorithm was introduced in Java 5 to save as much bandwidth as possible in the early 2000s through more effective compression. In times of 100 MBit lines, the development costs are no longer in proportion to the benefits. Pack200 is therefore marked as "deprecated for removal".

The Latest Features in Java 10

You can find more details and the latest version of this PDF at www.happycoders.eu/java-versions.

Local-Variable Type Inference – JEP 268 🌟

Since Java 10, we can use the *var* keyword to declare local variables ("local" means: within methods), as in the example on the right.

The extent to which *var* is used will probably lead to heated discussions. I use it when a) it is significantly shorter and b) I can clearly recognize the type in the code.

```
var i = 10;
var hello = "Hello world!";
var list = List.of(1, 2, 3, 4, 5);
var httpClient = HttpClient.newBuilder().build();
var status = getStatus();
var map = new HashMap<String, String>();
```

Immutable Collections 🌟

From Java 10 onwards, you can use *List.copyOf(...)*, *Set.copyOf(...)*, and *Map.copyOf(...)* to create immutable copies of lists, sets, and maps. Attempting to modify collections created in this way results in an *UnsupportedOperationException*.

Similarly, you can use *Collectors.toUnmodifiableList()*, *toUnmodifiableSet()*, and *toUnmodifiableMap()* to collect streams in immutable lists, sets, and maps.

Optional.orElseThrow() 🌟

The *Optional.get()* method throws a *NoSuchElementException* if an *Optional* is empty; it should therefore always be used in combination with *Optional.isPresent()*. IDEs and static code analysis tools generate a warning if you dont't do this. However, if such an exception is desired, this warning is annoying.

This is why the method *Optional.orElseThrow()* was introduced, which does exactly the same as *Optional.get()*, but is not reported by static code analysis tools, as its name clearly describes its behavior.

Parallel Full GC for G1 – JEP 307

The Garbage-First (G1) garbage collector has replaced the parallel collector as the standard garbage collector in Java 9. While the parallel GC could perform a complete garbage collection ("full GC") in parallel to the running application, this was so far not possible with G1 – it had to stop the application ("stop-the-world"), which could result in noticeable latencies.

In Java 10, full gargage collection of the G1 Garbage Collector has now also been parallelized.

Application Class-Data Sharing – JEP 310

Class-Data Sharing (CDS for short) means that once loaded, *.class* files are cached in an architecture-specific binary form so that the JVM start is accelerated, and the total memory requirement is reduced by sharing access to this cache from several JVMs running in parallel.

In Java 10, CDS was extended so that not only classes of the JDK class library can be cached, but also the classes of the application ("Application CDS", "AppCDS" for short).

Other Changes in Java 10

Consolidate the JDK Forest into a Single Repository (JEP 296) – The JDK source, which was previously stored in eight separate repositories, has been consolidated into a monorepo, which greatly simplifies work through atomic commits, branches, and PRs.

Garbage-Collector Interface (JEP 304) – A clean garbage collector interface was introduced in the JDK source code to isolate the GC algorithms from the interpreter and the compilers and to simplify the development of new GCs.

Thread-Local Handshakes (JEP 312) – Thread-local handshakes are an optimization to improve virtual machine performance on x64 and SPARC-based architectures.

Remove the Native-Header Generation Tool (javah) (JEP 313) – The *javah* tool was used to create native header files for JNI. The functionality was integrated into the Java compiler *javac* in Java 10.

Additional Unicode Language-Tag Extensions (JEP 314) – So-called "language tag extensions" allow additional information about the currency, the first day of the week, the region, and the time zone to be stored in a *Locale* object.

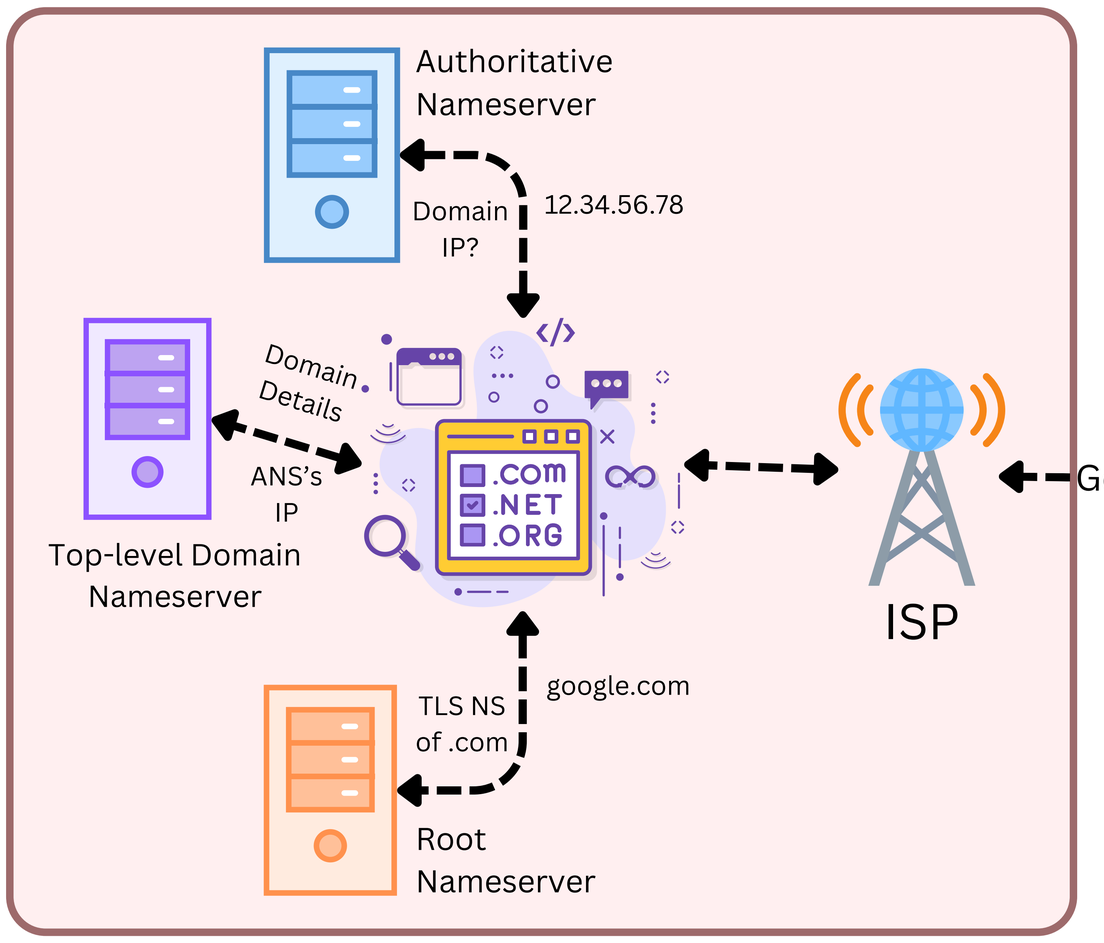
Heap Allocation on Alternative Memory Devices (JEP 316) – As of Java 10, the Java heap can also be created on an alternative memory device, e.g., on NV-DIMM (non-volatile memory), instead of on conventional RAM.

Experimental Java-Based JIT Compiler (JEP 317) – Since Java 9, the Graal compiler (a Java compiler written in Java) has been included as an experimental ahead-of-time (AOT) compiler. From Java 10 onwards, Graal can also be used as a just-in-time (JIT) compiler – initially on the Linux/x64 platform.

Root Certificates (JEP 319) – Root certificates previously only included in the Oracle JDK are now available in the OpenJDK, too.

Time-Based Release Versioning (JEP 322) – Version information now includes the release date and whether the version is a long-term support release. Examples: *java version "10.0.2" 2018-07-17*, *java version "11.0.11" 2021-04-20 LTS*

System Design Blueprint: The Ultimate Guide



Metrics

Security

Options / Algo

ByteByteGo

Request

Response

- Gzip, deflate
- Request ID
- Idempotent Key
- Many more...

- 4XX, 5XX, 2XX
- Pagination
- Expiry Headers
- Mime
- Cookie
- many more...

- Response Time
- Failed Status Codes
- Error Codes / Message

LOAD BALANCING

- Input Validation
- Authorization / Authentication
- Rate Limiting / Throttling
- Whitelist / Blacklisting
- Flow Control
- Authentication (OAuth 2.0)
- Request Header Validations
- TLS Termination
- Request Deduplication
- Metering / Usage data collection
- Request Dispatching

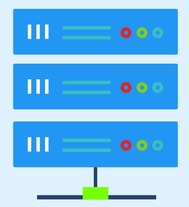
API Gateway

Load Balancer

#MultiPrimary

- Access Logs
- Status Codes
- Number of Requests
- Active Connections

- Round Robin
- Weighted Round Robin
- Least Connections
- Hash / Server Stickiness
- Random



Frontend Servers

#InMemoryConnections

For Live Streaming / Chat

- RTMP
- WebRTC
- Websocket
- SSE (Server Side Events)
- HTTP Short Polling
- HTTP Long Polling
- Webhook
- Stream API

User	Connection
user_1	ada_inst_obj

- Bandwidth
- Topics

Dispatch Messages to other FE Servers

CDN / Edge Servers

#Global #Regional #Static

Static / Stream with ABR

- Cache Miss Outs
- Active Resources
- Total Resources

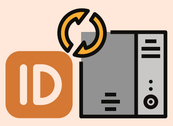
- Push Hot Resource
- Pull Rare Resource
- Hybrid

Serves Static Content

Concurrency:
- Serially
- Serially Batch
- Pessimistic Locking
- Optimistic Locking

- Vertical Scale
- Horizontal

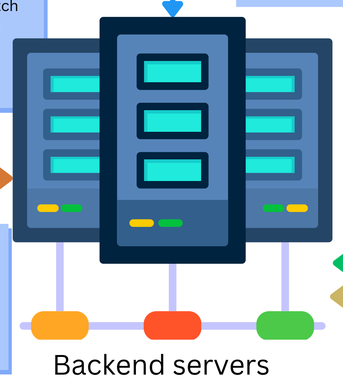
- UUID
- Auto Increment
- Auto Incr. Multiple Servers (Odd / Even)
- Twitter's SnowFlakes
- Offline Generations
- Baidu UID generator
- Sonyflake



Distributed ID Generator

- Redlock (Redis)
- Google Chubby
- Apache Zookeeper

Distributed Resource Locking



- Leaderless (#Gossip Protocol)
- Leader + Followers
- Multi-Leaders

- DiskIO (if buffering)
- CPU
- Memory
- Latency
- Bandwidth

Backend servers

#ClusterOfServers

#MicroService

Server	Topic / Obj ID
server_1	asdf23dsf3oj23098asfdf3

Message Dispatcher

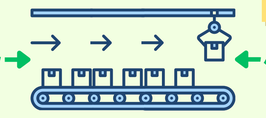
Object Storage

#S3 #Chunk #Raw

For every new chunk object

- Count of messages
- Consumption Rate
- In-Transit (Waiting for Ack)
- Queue limit

- Broker
- Pub/Sub



Message Queue

- Count of messages
- Consumption Rate
- In-Transit (Waiting for Ack)
- Queue limit

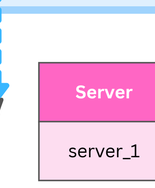
Validate Checksum

Processed / Encoded Storage

#Lossless #Lossy #Compression

- Compression Ratio
- Storage Consumed
- Object Count

- Compute Time
- Failed Count
- CPU / Disk, etc



Processing Workers

Validate checksum

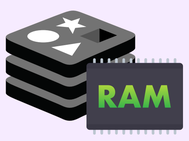
Push Processed Media Files

UPLOAD MEDIA

Key	Checksum	Timestamp
chunk_1	asdf23dsf3oj23098asfdf3	1532873423

STREAMING CHUNK METADATA TABLE

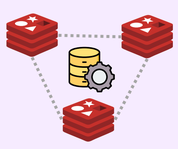
- Write-Through
- Read-Through
- Write-Around
- Write-Back



In-Memory Cache

#Fast

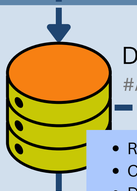
- No. of items
- Cache Miss & Hit
- Disk & Memory Usage



Distributed Cache

#Eviction #Invalidation

Eviction:
- LRU (Least Recently used)
- LFU (Least Freq. Used)
- FIFO
- MRU
- Random Eviction
- Least Used
- On-Demand Expiration
- Garbage Collector



Databases

#ACID #BASE

- RDMS
- Column Wide
- Document
- Key-Value
- Graph
- QuadTree
- Time-series

Cold Storage

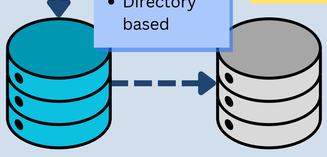
#OldRecords

- Consistent Hashing
- Mod Hashing / Bloom Filter

- RBAC (Role Based Access Control)
- Data Encryption
- Audit Trail

- Range based
- Hash based
- Geographical
- Directory based

- Query response time
- CPU / Disk Usage
- Network Throughput
- Active Connections
- Storage



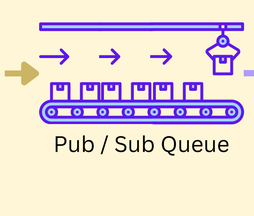
Shards

#HotNode #Hash

Replicas

#CrossGeo

- Quorum (Read / Write)
- Hinted-off
- Merkle Tree (Algo)



Pub / Sub Queue

Recommendation Service

Log Processing Service

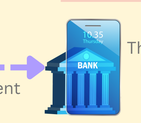
Search Service

Analytics Service

Payment Charge Service

Filtering:
- Content-Based
- Collaborative

- Expired / Blocked / Invalid Card
- Service Down
- Insufficient Balance



With Idempotent Key

Third-Party Banking Service

Notification Service

- Spamming
- Stop Words
- Dedup
- Retry with Idempotent Key

COMMON FAN-OUT SERVICES

JVM Options Cheat Sheet

JRebel | XRebel

STANDARD OPTIONS

`$ java`
List all standard options.

`-Dblog=JRebelBlog`
Sets a 'blog' system property to 'JRebelBlog'. Retrieve/set it during runtime like this:

```
System.getProperty("blog");
//JRebelBlog

System.setProperty("blog", "JR");
```

`-javaagent:/path/to/agent.jar`
Loads the java agent in agent.jar.

`-agentpath:pathname`
Loads the native agent library specified by the absolute path name.

`-verbose:[class/gc/jni]`
Displays information about each loaded class/ gc event/JNI activity.

NON-STANDARD OPTIONS

`$ java -X`
List all non-standard options.

`-Xint`
Runs the application in interpreted-only mode.

`-Xbootclasspath:path`
Path and archive list of boot class files.

`-Xloggc:filename`
Log verbose GC events to filename.

`-Xms1g`
Set the initial size (in bytes) of the heap.

`-Xmx8g`
Specifies the max size (in bytes) of the heap.

`-Xnoclassgc`
Disables class garbage collection.

`-Xprof`
Profiles the running program.

ADVANCED OPTIONS

BEHAVIOR

`-XX:+UseConcMarkSweepGC`
Enables CMS garbage collection.

`-XX:+UseParallelGC`
Enables parallel garbage collection.

`-XX:+UseSerialGC`
Enables serial garbage collection.

`-XX:+UseG1GC`
Enables G1GC garbage collection.

DEBUGGING

`-XX:ErrorFile=file.log`
Save the error data to file.log.

`-XX:+HeapDumpOnOutOfMemoryError`
Enables heap dump when `OutOfMemoryError` is thrown.

`-Xlog:gc`
Enables printing messages during garbage collection.

`-XX:+TraceClassLoading`
Enables Trace loading of classes.

`-XX:+PrintClassHistogram`
Enables printing of a class instance histogram after a `Control+C` event `(SIGTERM)`.

PERFORMANCE

`-XX:ThreadStackSize=256k`
Sets Thread Stack Size (in bytes).
(Same as -Xss256k)

`-XX:G1HeapRegionSize=4m`
Sets the sub-division size of G1 heap (in bytes).

`-XX:MaxGCPauseMillis=n`
Sets a target for the maximum GC pause time.

`-XX:MaxNewSize=256m`
Max size of new generation (in bytes).

`-XX:OnError="<cmd args>"`
Run user-defined commands on fatal error.

spring Annotations Cheat Sheet

JRebel

Spring Boot and Web Annotations

Use annotations to configure your web application.

T **@SpringBootApplication** - uses @Configuration, @EnableAutoConfiguration and @ComponentScan.

T **@EnableAutoConfiguration** - make Spring guess the configuration based on the classpath.

T **@Controller** - marks the class as web controller, capable of handling the requests.

T **@RestController** - a convenience annotation of a @Controller and @ResponseBody.

M T **@ResponseBody** - makes Spring bind method's return value to the web response body.

M **@RequestMapping** - specify on the method in the controller, to map a HTTP request to the URL to this method.

P **@RequestParam** - bind HTTP parameters into method arguments.

P **@PathVariable** - binds placeholder from the URI to the method parameter.

Spring Framework Annotations

Spring uses dependancy injection to configure and bind your appliation together.

T **@Configuration** - used to mark a class as a source of the bean definitions.

T **@ComponentScan** - makes Spring scan the packages configured with it for the @Configuration classes.

T **@Import** - loads additional configuration. This one works even when you specify the beans in an XML file.

T **@Component** - turns the class into a Spring bean at the auto-scan time.

T **@Service** - tells Spring that it's safe to manage @Components with more freedom than regular components.

C F M **@Autowired** - wires the application parts together, on the fields, constructors, or methods in a component.

M **@Bean** - specifies a returned bean to be managed by Spring context. The returned bean has the same name as the factory method.

M **@Lookup** - tells Spring to return an instance of the method's return type when we invoke it.

T M **@Primary** - gives higher preference to a bean when there are multiple beans of the same type.

C F M **@Required** - shows that the setter method must be configured to be dependency-injected with a value at configuration time.

C F M **@Value** - used to assign values into fields in Spring-managed beans. It's compatible with the constructor, setter, and field injection.

T M **@DependsOn** - makes Spring initialize other beans before the annotated one.

T M **@Lazy** - makes beans to initialize lazily. @Lazy annotation may be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean.

T M **@Scope** - used to define the scope of a @Component class or a @Bean definition and can be either singleton, prototype, request, session, globalSession, or custom scope.

T **@Profile** - adds beans to the application only when that profile is active.

Legend: T - Class F - Field Annotation C - Constructor Annotation M - Method P - Parameter

docker CHEATSHEET

definitions

image — *a static snapshot of container's configuration.*

container — *an application sandbox. each container is based on an image.*

layer — *image is composed of read-only file system layers. container creates single writable layer.*

docker registry — *remote server for storing Docker images*

Dockerfile — *a configuration file wih build instructions for Docker images*

docker engine — *Docker platform installation running on a given host*

docker client — *client application that talks to local or remote Docker daemon*

docker daemon — *service process that listens to Docker client commands over local or remote network*

docker host — *server that runs Docker engine*

volume — *directory shared between host and container*

docker run

`docker run [OPTIONS] IMAGE[:TAG] [COMMAND]`

Run a command in a new container.

metadata

`--name=CNTNR_NAME`	*Assign a name to the container.*
`-l, --label NAME[=VALUE]`	*Set metadata on the container.*

process

`-d, --detach`	*Run in the background.*
`-i, --interactive`	*Keep STDIN open.*
`-t, --tty`	*Allocate a pseudo-TTY.*
`--rm`	*Automatically remove the container when process exits.*
`-u USER`	*Run as username or UID.*
`--privileged`	*Give extended privileges.*
`-w DIR`	*Set working directory.*
`-e NAME=VALUE`	*Set environment variable.*
`--restart=POLICY`	*Restart policy.* `no` `on-failure[:RETRIES]` `always` `unless-stopped`

network

`-P, --publish-all`	*Publish all exposed ports to random ports.*
`-p HOST_PORT:CNTNR_PORT`	*Expose a port or a range of ports.*
`--network=NETWORK_NAME`	*Connect container to a network.*
`--dns=DNS_SERVER1[,DNS_SERVER2]`	*Set custom dns servers.*
`--add-host=HOSTNAME:IP`	*Add a line to /etc/hosts.*

file system

`--read-only`	*Mount the container's root file system as read only.*
`-v, --volume [HOST_SRC:]CNTNR_DEST`	*Mount a volume between host and the container file system.*
`--volumes-from=CNTNR_ID`	*Mount all volumes from another container.*

Dockerfile

`FROM <image_id>`
base image to build this image from

`RUN <command>` *shell form*

`RUN ["<executable>", "<param1>", ..., "<paramN>"]` *exec form*
executes command to modify container's file system state

`MAINTAINER <name>`
provides information about image creator

`LABEL <key>=<value>`
*adds searchable **metadata** to image*

`ARG <name>[=<default value>]`
*defines overridable **build-time parameter**:*
`docker build --build-arg <name>=<value> .`

`ENV <key>=<value>`
*defines **environment variable** that will be visible during image build-time and container run-time*

`ADD <src> <dest>`
*copies files from `<src>` (**file, directory or URL**) and adds them to container file system under `<dst>` path*

`COPY <src> <dest>` *similar to `ADD`, does not support URLs*

`VOLUME <dest>`
*defines **mount point** to be shared with host or other containters*

`EXPOSE <port>`
*informs Docker engine that container listens to **port** at run-time*

`WORKDIR <dest>`
sets build-time and run-time working directory

`USER <user>`
defines run-time user to start container process

`STOPSIGNAL <signal>`
defines signal to use to notify container process to stop

`ENTRYPOINT` *shell form* or *exec form*
*defines run-time **command prefix** that will be added to all run commands executed by `docker run`*

`CMD` *shell form* or *exec form*
*defines run-time **command** to be executed by default when `docker run` command is executed*

extreme automation

Find more cheatsheets at https://extremeautomation.io/cheatsheets

v1.5

docker Cheat Sheet

JRebel | XRebel

GLOSSARY

Layer
A set of read-only files to provision the system.

Image
A read-only layer that is the base of your container. Might have a parent image.

Container
A runnable instance of the image.

Registery / Hub
Central place where images live.

Docker machine
A VM to run Docker containers (Linux does this natively).

Docker compose
A utility to run multiple containers as a system.

USEFUL ONE-LINERS

Download an image
```
docker pull image_name
```

Start and stop the container
```
docker [start|stop] container_name
```

Create and start container, run command
```
docker run -ti --name container_name
    image_name command
```

Create and start container, run command, destroy container
```
docker run --rm -ti image_name command
```

Example filesystem and port mappings
```
docker run -it --rm -p 8080:8080 -v
/path/to/agent.jar:/agent.jar -e
JAVA_OPTS="-javaagent:/agent.jar"
tomcat:8.0.29-jre8
```

DOCKER CLEANUP COMMANDS

Kill all running containers
```
docker kill $(docker ps -q)
```

Delete dangling images
```
docker rmi $(docker images -q -f
    dangling=true)
```

Remove all stopped containers
```
docker rm $(docker ps -a -q)
```

DOCKER MACHINE COMMANDS

Use docker-machine to run the containers

Start a machine
```
docker-machine start machine_name
```

Configure docker to use a specific machine
```
eval "$(docker-machine env machine_name)"
```

DOCKER COMPOSE SYNTAX

docker-compose.yml file example
```
version: "2"
services:
web:
  container_name: "web"
  image: java:8 # image name
  # command to run
  command: java -jar /app/app.jar
  ports: # map ports to the host
    - "4567:4567"
  volumes: # map filesystem to the host
    - ./myapp.jar:/app/app.jar
mongo: # container name
  image: mongo # image name
```

Create and start containers
```
docker-compose up
```

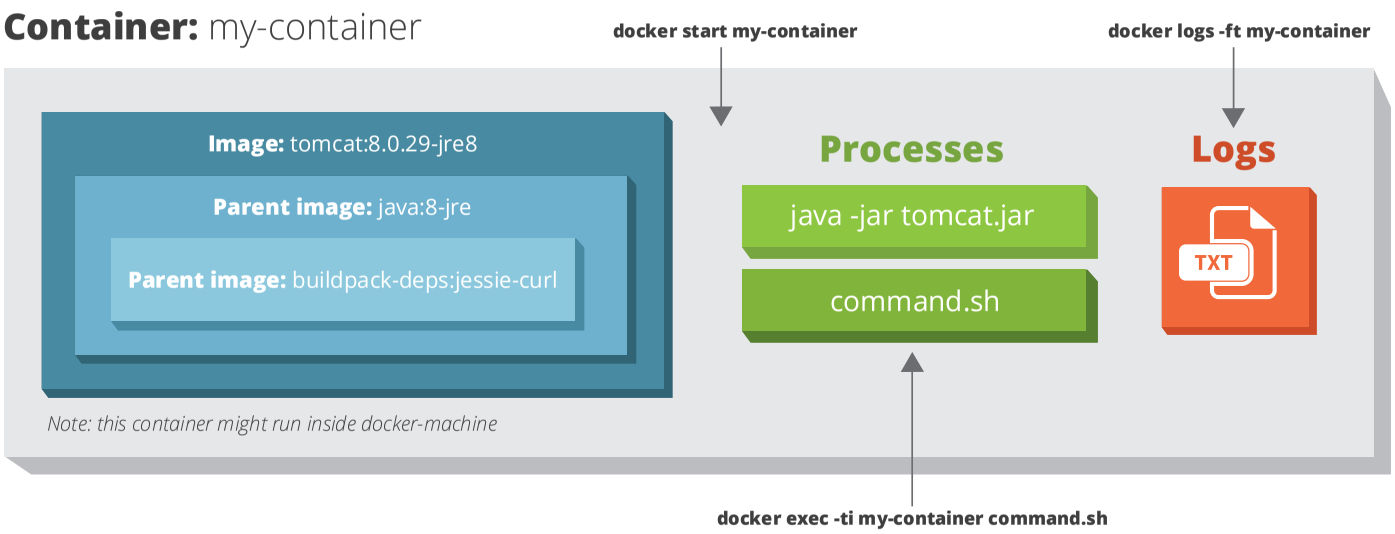
INTERACTING WITH A CONTAINER

Run a command in the container
```
docker exec -ti container_name command.sh
```

Follow the container logs
```
docker logs -ft container_name
```

Save a running container as an image
```
docker commit -m "commit message" -a "author"
    container_name username/image_name:tag
```

Container: my-container



Git Cheat Sheet

JRebel

Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

Observe a Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

Working With Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new_branch

```
$ git branch new_branch
```

Delete the branch called my_branch

```
$ git branch -d my_branch
```

Merge branch_a into branch_b

```
$ git checkout branch_b
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

Make a Change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

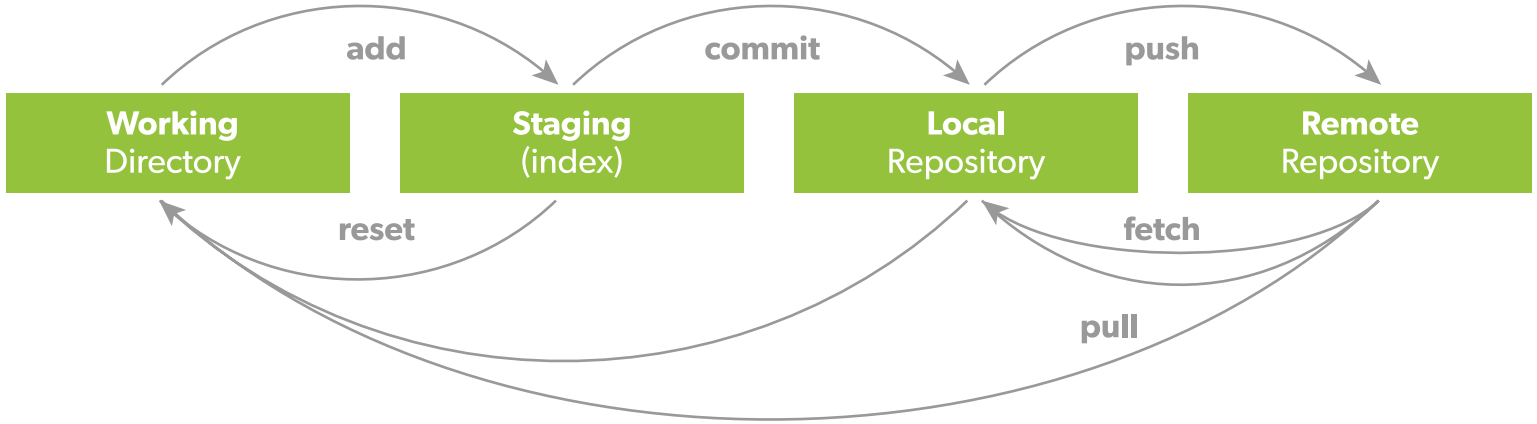
```
$ git push
```

Finally!

When in doubt, use git help

```
$ git [command] –help
```

Or visit training.github.com for official GitHub training.



Java 8 Streams Cheat Sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Definitions

- ☑ A stream **is** a pipeline of functions that can be evaluated.
- ☑ Streams **can** transform data.
- ☒ A stream **is not** a data structure.
- ☒ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
     .map(book -> book.getAuthor())
     .filter(author -> author.getAge() >= 50)
     .distinct()
     .limit(15)
     .map(Author::getSurname)
     .map(String::toUpperCase)
     .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
     .map(Book::getAuthor)
     .filter(a -> a.getGender() == Gender.FEMALE)
     .map(Author::getAge)
     .filter(age -> age < 25)
     .reduce(0, Integer::sum):
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
     groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
     .map(member -> member.getFollowers())
     .flatMap(followers -> followers.stream())
     .collect(toList());
```

Pitfalls

☒ Don't update shared mutable variables i.e.

```
List<Book> myList =
new ArrayList<>();
library.stream().forEach
(e -> myList.add(e));
```

☒ Avoid blocking operations when using parallel streams.

Java Generics cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Basics

Generics don't exist at runtime!

```
class Pair<T1, T2> { /* ... */ }
```
-- the type parameter section, in angle brackets, specifies type variables.

Type parameters are substituted when objects are instantiated.

```
Pair<String, Long> p1 = new
  Pair<String, Long> ("RL", 43L);
```

Avoid verbosity with the diamond operator:

```
Pair<String, Long> p1 =
  new Pair<>("RL", 43L);
```

Wildcards

`Collection<Object>` - heterogenous, any object goes in.
`Collection<?>` - homogenous collection of arbitrary type.

Avoid using wildcards in return types!

Intersection types

```
<T extends Object &
  Comparable<? super T>> T
max(Collection<? extends T> coll)
```

The return type here is ***Object***!

Compiler generates the bytecode for the most general method only.

Producer Extends Consumer Super (PECS)

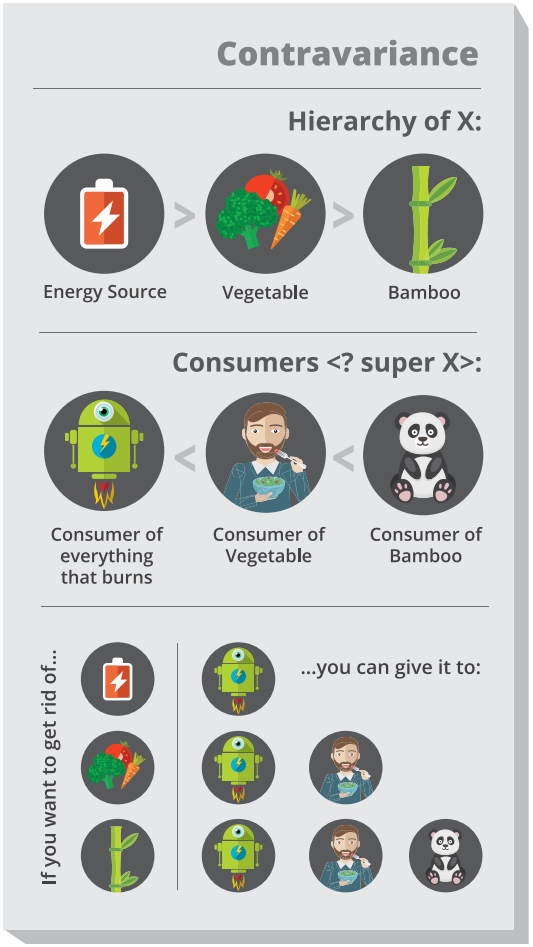
```
Collections.copy(List<? super T> dest, List<? extends T> src)
```

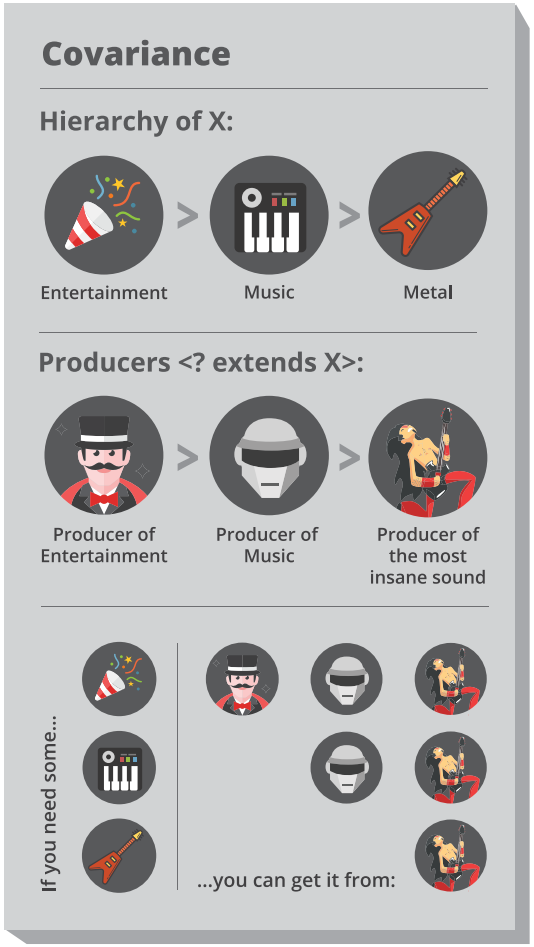
src -- contains elements of type T or its subtypes.

dest -- accepts elements, so defined to use T or its supertypes.

*Consumers are **contravariant** (use super).*

*Producers are **covariant** (use extends).*





Method Overloading

```
String f(Object s) {
  return "object";
}
String f(String s) {
  return "string";
}
<T> String generic(T t) {
  return f(t);
}
```

If called **generic("string")** returns "object".

Recursive generics

Recursive generics add constraints to your type variables. This helps the compiler to better understand your types and API.

```
interface Cloneable<T extends
Cloneable<T>> {
  T clone();
}
```

Now `cloneable.clone().clone()` will compile.

Covariance

```
List<Number> ⊁ ArrayList<Integer>
```

Collections are not covariant!

BROUGHT TO YOU BY JRebel

Java 8 CHEATSHEET

lambdas

*A **lambda expression** is like a method: it provides a list of formal parameters and a body - an expression or block - expressed in terms of those parameters.*

```
(param1, param2, ...) -> expression
(param1, param2, ...) -> { stmt1; stmt2; ... }
```

***Functional interfaces** provide target types for lambda expressions and method references. Each functional interface has a **single abstract method**, to which the lambda expression's **parameters and return types** are matched or adapted.*

Example:

```
@FunctionalInterface
interface Comparator<T> {
  int compare(T o1, T o2);
}

Comparator<Person> c =
  (Person p1, Person p2) ->
    p1.getName().compareTo(p2.getName());
```

functional method · *parameter list* · *implementation* · *lambda expression*

```
Predicate<T>      → boolean test(T t)
Supplier<T>       → T get()
Consumer<T>       → void accept(T t)
Function<T, V>    → V apply(T t)
BiPredicate<T, U> → boolean test(T t, U u)
BiConsumer<T, U>  → void accept(T t, U u)
BiFunction<T, U, V> → V apply(T t, U u)
BinaryOperator<T> → T test(T t1, T t2)
```

stream api

*Stream operations are divided into **intermediate** and **terminal** operations, and are combined to form stream **pipelines**. A stream pipeline consists of a **source** followed by zero or more **intermediate** operations and a **terminal** operation.*

***Intermediate** operations return a new stream. They are always **lazy**! **Terminal** operations may traverse the stream to produce a **result** or a **side-effect**.*

Example:

source · *lambda expression* · *intermediate* · *method reference* · *terminal*

```
people
  .stream()
  .map(person -> person.getAddress())
  .filter(address -> address.isVerified())
  .map(Address::getCity)
  .map(String::toUpperCase)
  .distinct()
  .limit(10)
  .collect(Collectors.toList());
```

Intermediate operations:

input	method	output
1 2 3 4 5	`map(function)` *1 → a*	a b c d e
1 2 3 4 5	`flatMap(function)` *1 → [a, b]*	a b c d e f g h i j
1 2 3 4 5	`filter(predicate)` *x≠2 && x≠5*	1 3 4
1 2 3 4 5	`peek(consumer)` *side effect*	1 2 3 4 5
1 2 3 4 5	`limit(int)` *3*	1 2 3
1 2 3 4 5	`skip(int)` *2*	3 4 5
1 1 2 1 2	`distinct()`	1 2
4 2 1 5 3	`sorted(comparator)`	1 2 3 4 5

Terminal operations:

input	method	result
1 2 3 4 5	`findAny(predicate)` *x % 2 == 0*	2 *not guaranteed*
1 2 3 4 5	`findFirst(predicate)` *x % 2 == 0*	2 *guaranteed*
1 2 3 4 5	`allMatch(predicate)` *x≠6*	`true`
1 2 3 4 5	`noneMatch(predicate)` *x % 2 == 0*	`false`
1 2 3 4 5	`anyMatch(predicate)` *x % 2 == 0*	`true`
1 2 3 4 5	`count()`	5
1 2 3 4 5	`min(comparator)`	`Optional(1)`
1 2 3 4 5	`max(comparator)`	`Optional(5)`
1 2 3 4 5	`collect(collector)` `Collectors.toList()`	`List(...)`
1 2 3 4 5	`forEach(consumer)` *side effect*	`void`
1 2 3 4 5	`reduce(binaryOperator)` `Integer::sum`	`Optional(15)`

lambda examples

```
()                -> {}                    // no parameters, result is void
()                -> 42                    // expression body
()                -> null                  // expression body
()                -> { return 42; }        // block body
()                -> { System.gc(); }      // block body
(int x)           -> x + 1
(int x)           -> { return x + 1; }     // block body
(x)               -> x + 1
 x                -> x + 1                 // parenthesis are optional for single inferred-type parameter
(int x, int y)    -> x + y
(x, y)            -> x + y
(String s)        -> s.length()
(Thread t)        -> { t.start(); }
```

extreme automation

JUnit cheat sheet

For more awesome cheat sheets visit rebellabs.org!

REBELLABS by ZEROTURNAROUND

Assertions and assumptions

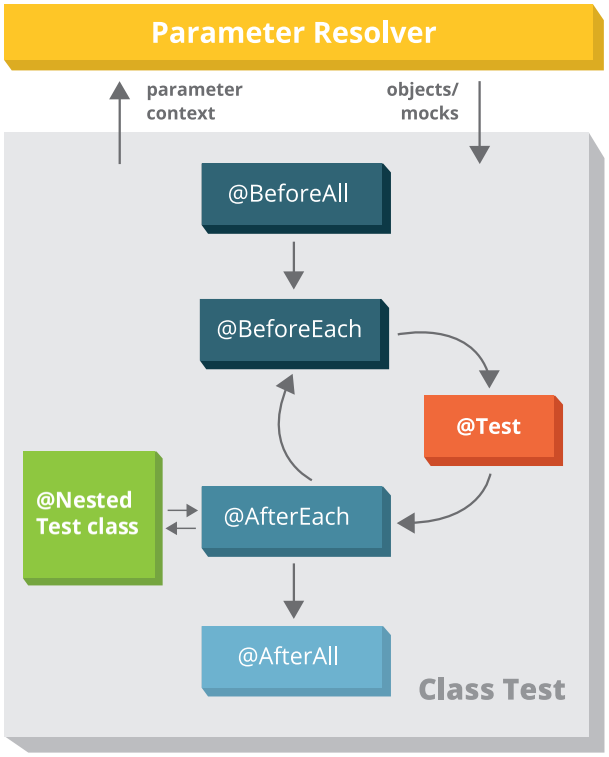
Use assertions to verify the behavior under test:

```
Assertions.assertEquals(Object expected,
Object actual, Supplier<String> message)

// Group assertions are run all together and
reported together.
Assertions.assertAll("heading",
  () -> assertTrue(true),
  () -> assertEquals("expected",
            objectUnderTest.getSomething());

// To check for an exception:
expectThrows(NullPointerException.class,
  () -> { ((Object) null).getClass();});
```

Lifecycle of standard tests



Parameter resolution

ParameterResolver - extension interface to provide parameters

```
public class MyInfoResolver implements ParameterResolver {
  public Object resolve(ParameterContext paramCtx,
                        ExtensionContext extCtx) {
    return new MyInfo();
  }
}
```

Extend your tests with your parameter resolver.

```
@ExtendWith(MyInfoResolver.class)
class MyInfoTest { … }
```

Useful code snippets

```
@TestFactory
Stream<DynamicTest> dynamicTests(MyContext ctx) {
// Generates tests for every line in the file
 return Files.lines(ctx.testDataFilePath).map(l ->
dynamicTest("Test:" + l, () -> assertTrue(runTest(l)));
}

@ExtendWith({ MockitoExtension.class,
DataParameterProvider.class })
class Tests {
  ArrayList<String> list;

  @BeforeEach
  void init() { /* init code */ }

  @Test
  @DisplayName("Add elements to ArrayList")
  void addAllToEmpty(Data dep) {
    list.addAll(dep.getAll());
    assertAll("sizes",
      () -> assertEqual(dep.size(), list.size(),
              () -> "Unexpected size:" + instance),
      () -> assertEqual(dep.getFirst(), list.get(0),
              () -> "Wrong first element" + instance));
  }
}
```

Useful annotations

@Test - marks a test method
@TestFactory - method to create test cases at Runtime
@DisplayName - make reports readable with proper test names
@BeforeAll/@BeforeEach - lifecycle methods executed prior testing
@AfterAll/AfterEach - lifecycle methods for cleanup
@Tag - declare tags to separate tests into suites
@Disabled - make JUnit skip this test.

Use **@Nested** on an inner class to control the order of tests.

Use **@ExtendWith()** to enhance the execution: provide mock parameter resolvers and specify conditional execution.

Use the lifecycle and **@Test** annotations on the default methods in interfaces to define contracts:

```
@Test
interface HashCodeContract<T> {
  <T> getValue();
  <T> getAnotherValue();
  @Test
  void hashCodeConsistent() {
    assertEquals(getValue().hashCode(),
                 getAnotherValue().hashCode(),
                 "Hashes differ");
}
```

"Never trust a test you haven't seen fail."

— Colin Vipurs

JVM internals page 1 CHEAT SHEET

definitions

*A **Java Virtual Machine** (JVM) is an abstract computing machine that enables a computer to run a Java program.*

***Java Runtime Environment** (JRE) is a software package that contains what is required to run a Java program.*

*The **class loader** is a part of the JRE that dynamically loads Java classes into the JVM.*

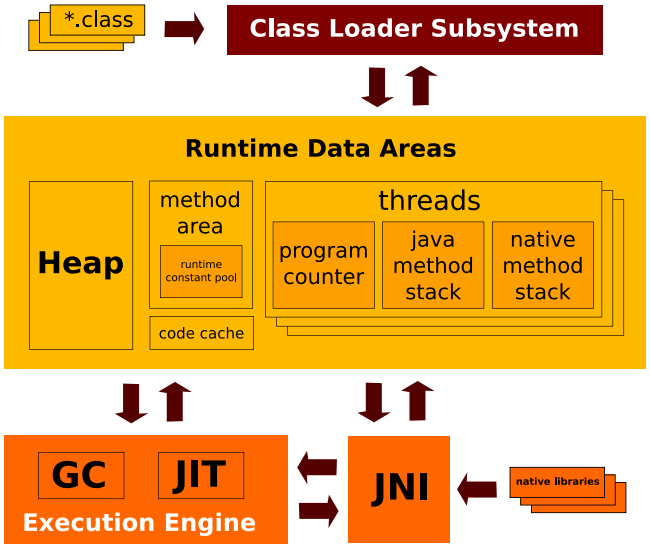
*The JVM defines various **runtime data areas** that are used during execution of a program.*

*The **heap** is the run-time data area from which memory for all class instances and arrays is allocated. Heap storage for objects is reclaimed by an automatic storage management system (known as a **garbage collector**).*

*The **method area** is the storage area for compiled bytecode.*

*Each JVM **thread** has a private **stack**, created at the same time as the thread. A stack stores **frames**. A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions. Each thread has its own **program counter** register. The PC register contains the address of the JVM instruction currently being executed.*

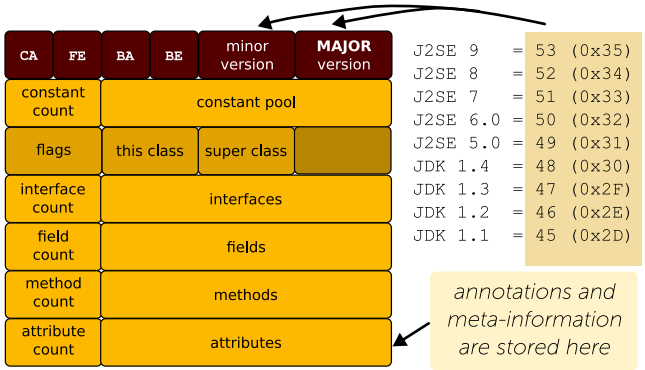
overview



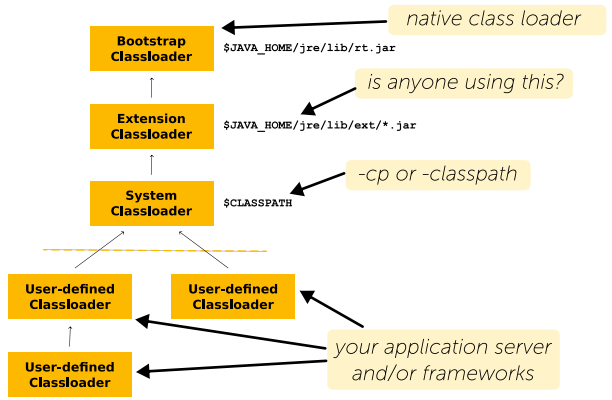
class file

*A **class file** is a precisely defined binary format for Java programs. Each class file represents a complete description of **one** class or interface.*

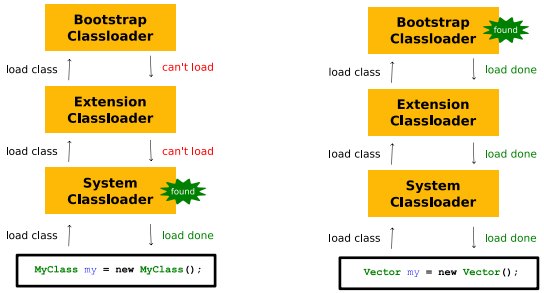
*Class file format remains **backward-compatible**.*



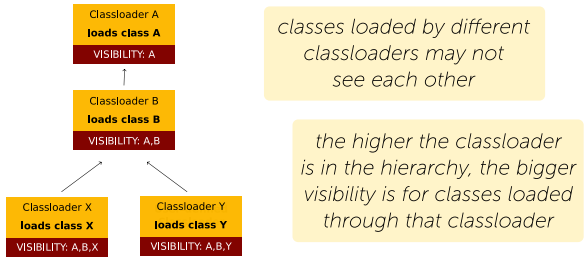
class loaders



Delegation principle:



Visibility principle:



extreme automation

JVM internals page 2 CHEAT SHEET

definitions

*The **heap** is the run-time data area from which memory for all class instances and arrays is allocated. Heap storage for objects is reclaimed by an automatic storage management system (known as a **garbage collector**).*

***HotSpot** JVM distinguishes the following memory areas:*

***Eden space** is a memory pool in which memory is initially allocated.*

***Survivor space** is a memory pool which contains objects that survived garbage collection in Eden space.*

***Young generation** consists of eden and two equally sized survivor spaces.*

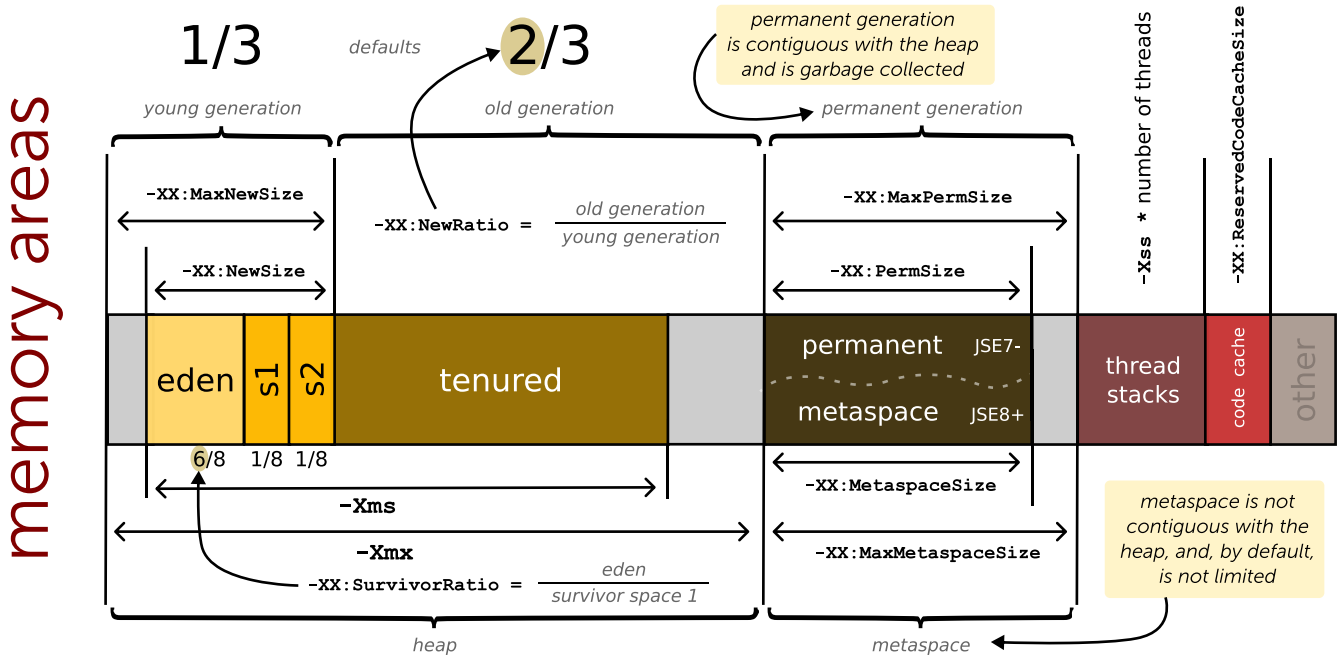
***Tenured (old) generation** contains objects that passed several garbage collections in the survivor spaces.*

***Permanent generation** contains class and method data. It is NOT part of the heap, but is contiguous with the heap. It was replaced with metaspace in JDK 8.*

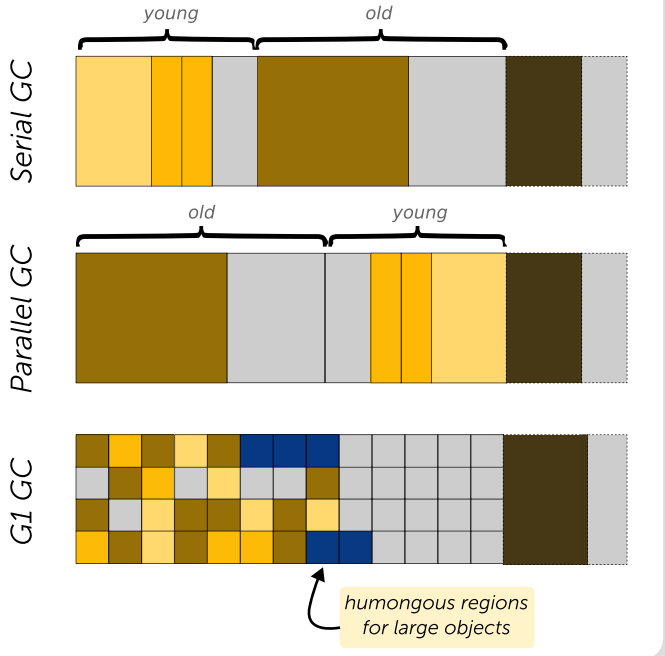
***Metaspace** is not contiguous with the heap and has different management mechanics. It is not limited by default, and, therefore, theoretically, can fill all available OS memory.*

***Code cache** contains native code compiled by **JIT** compiler at runtime.*

memory areas



memory layout



collectors

*A **serial collector** uses a single CPU core to perform garbage collection.*

*A **parallel collector** uses multiple threads that can perform collection work at the same time, and can use of more than one CPU core.*

*A **stop-the-world (STW) collector** performs garbage collection during a pause when the application is not running any code that could mutate the heap.*

*A **concurrent collector** performs garbage collection concurrently with program execution, i.e. allowing program execution to proceed whilst collection is carried out.*

extreme automation

KUBERNETES CHEAT SHEET

KUBERNETES

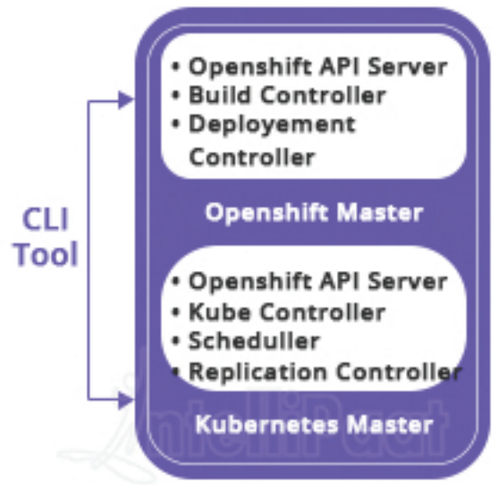
• It is an open source platform for automating deployment and scaling of containers across clusters of hosts providing container centric infrastructure.

• It is a container orchestrator and can run Linux containers:

• Launch container.

• Maintain and monitor container site.

• Performs container-oriented networking



Master(s)
Manage,Build,Deploy & Operate

NODE(s)
Run Containers and registries

Key Concepts

Now let's discuss the key points of this architecture.

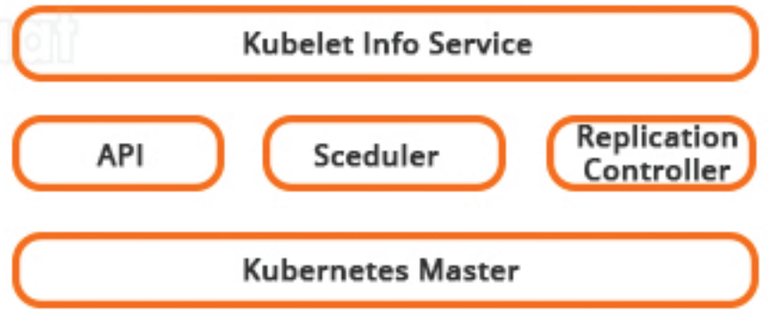
• Pod: These are the group of containers.
• Labels: These are used to identify the pods.
• Kubelet: They are container agents, responsible for maintaining the set of pods.
• Proxy: They are the Load balancer for pods, helping in distributing tasks across the pods.
• ETCD: A Metadata service.

• Cadvisor: For resource usage and performance stats.
• Replication controller: It manages pod replication.
• Scheduler: Used for pod scheduling in worker nodes.
• API server: Kubernetes API server.

Now let's understand the role Master and Node play in the Kubernetes Architecture.

Master

• It is responsible for maintaining the desired state for the cluster you are working on.
• "Master" indicates a set of processes that are used to manage the cluster.
• Contains info, API, scheduler, replication controllers, and master.

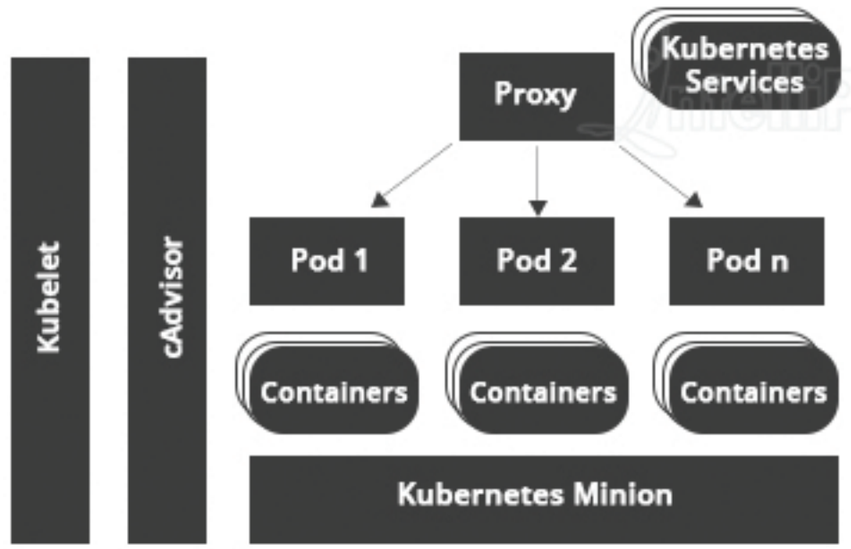


Worker Nodes / Minions

• Also called as a minion. It contains the services necessary to run the pods that are managed by the master.

• Some services include: container runtime, Kubelet, kube-proxy.

• Contains: Kubelet, cAdvisor, services, pods and containers.



Features

• Automated scheduling- provides an advanced scheduler that helps launch container on cluster nodes
• Self healing- reschedule, replace and restart dead containers.
• Automated rollouts and rollbacks- supports rollback for systems incase of a failure. Enables rollout and rollback for the desired state.
• Horizontal scaling- can scale up and down the app as per required. Can also be automated wrt CPU usage.
• Service discovery and load balancing- uses unique ip and dns name to containers. This helps identify them across different containers.

• Pods and Container Introspection

COMMANDS	FUNCTION
Kubectl get pods	Lists all current pods
Kubectl describe pod<name>	Describes the pod names
Kubectl get rc	List all replication controllers
Kubectl get rc --namespace="namespace"	Lists all current pods
Kubectl describe rc <name>	Shows the replication controller name
Kubectl get cvc	Lists the services
Kubectl describe svc <name>	Shows the service name
Kubectl delete pod <name>	Deletes the pod
Kubectl get nodes -w	Watch nodes continuously

• Debugging

COMMANDS	FUNCTION
Execute command on service by selecting container.	Kubectl exec<service><commands>[-c< $container>]
Get logs from service for a container	Kubectl logs -f<name>>[-c< $container>]
Watch the kubelet logs	Watch -n 2 cat/var/log/kublet.log
Show metrics for node	Kubectl top node
Show metrics for pods	Kubectl top pod

• Objects

All	Clusterroleblindings	FUNCTION
cm= conf gmaps c	controllerrevisions	crd=custom resource definition
Cronjobs	cs=component status	csr= certificate signing requests
Deploy=deployments	ds= daemon sets	ep=end points
ev= events	hpa= autoscaling	ing= ingress
jobs	limits=limit ranges	Netpol- network policies
No = nodes	ns= namespaces	pdb= pod
po= pods	Pod preset	Pod templates
Psp= pod security policies	Pv= persistent volumes	pvc= persistent volume claims
quota= resource quotas	rc= replication controllers	Role bindings
roles	rs= replica sets	sa=service account
sc= storage classes	secrets	sts= stateful sets

• Cluster Introspection

COMMANDS	FUNCTION
Get version information	Kubectl version
Get cluster information	Kubectl cluster-info
Get the configuration	Kubectl config g view
Output info about a node	Kubectl describe node<node>

Other Quick Commands

Launch a pod with a name an image : Kubectl run<name> --image=<image-name>

Create a service in <manifest.yaml> : Kubectl create -f <manifest.yaml>

Scale replication counter to count the number of instances : Kubectl scale --replicas=<count>

Map external port to internal replication port : Expose rc<name> --port=<external>--target-port=<internal>

To stop all pod in <n> : Kubectl drain<n>-- delete-local-data--force--ignore-daemonset

Allow master nodes to run pods : Kubectltaintnodes --all-noderole.kuernetes.io/master-

IntelliPaat

Maven cheat sheet

Getting started with Maven

Create Java project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
```

Create web project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
-DarchetypeArtifactId=maven-archetype-webapp
```

Create archetype from existing project

```
mvn archetype:create-from-project
```

Main phases

clean — delete target directory
validate — validate, if the project is correct
compile — compile source code, classes stored in target/classes
test — run tests
package — take the compiled code and package it in its distributable format, e.g. JAR, WAR
verify — run any checks to verify the package is valid and meets quality criteria
install — install the package into the local repository
deploy — copies the final package to the remote repository

Useful command line options

-DskipTests=true compiles the tests, but skips running them

-Dmaven.test.skip=true skips compiling the tests and does not run them

-T - number of threads:
-T 4 is a decent default
-T 2C - 2 threads per CPU

-rf, --resume-from resume build from the specified project

-pl, --projects makes Maven build only specified modules and not the whole project

-am, --also-make makes Maven figure out what modules out target depends on and build them too

-o, --offline work offline

-X, --debug enable debug output

-P, --activate-profiles comma-delimited list of profiles to activate

-U, --update-snapshots forces a check for updated dependencies on remote repositories

-ff, --fail-fast stop at first failure

Essential plugins

Help plugin — used to get relative information about a project or the system.
mvn help:describe describes the attributes of a plugin
mvn help:effective-pom displays the effective POM as an XML for the current build, with the active profiles factored in.

Dependency plugin — provides the capability to manipulate artifacts.
mvn dependency:analyze analyzes the dependencies of this project
mvn dependency:tree prints a tree of dependencies

Compiler plugin — compiles your java code.
Set language level with the following configuration:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</
  artifactId>
 <version>3.6.1</version>
 <configuration>
   <source>1.8</source>
   <target>1.8</target>
 </configuration>
</plugin>
```

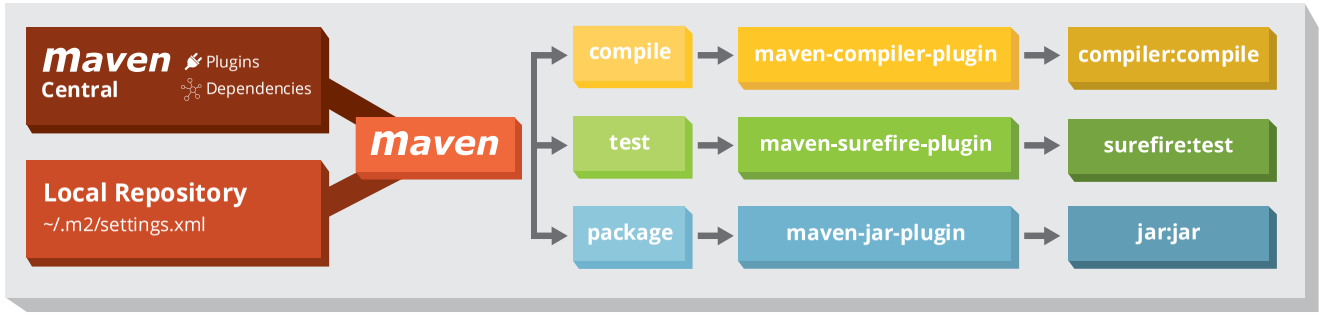
Version plugin — used when you want to manage the versions of artifacts in a project's POM.

Wrapper plugin — an easy way to ensure a user of your Maven build has everything that is necessary.

Spring Boot plugin — compiles your Spring Boot app, build an executable fat jar.

Exec — amazing general purpose plugin, can run arbitrary commands :)

The big picture



RxJava cheat sheet

Basic RxJava classes

`Observable<T>` - emits 0 or n items and terminates with complete or an error.

`Single<T>` - emits either a single item or an error. The reactive version of a method call. You subscribe to a *Single* and you get either a return value or an error.

`Maybe<T>` - succeeds with either an item, no item, or errors. The reactive version of an *Optional*.

`Completable` - either completes or returns an error. It never return items. The reactive version of a *Runnable*.

Creating observables

Create an observable from a value, a collection or *iterable*, or a result of a *callable*:

```
Observable.just("RebelLabs");
Observable.fromIterable(iterable);
Observable.fromCallable(callable);
```

RxBindings

Turns Android UI events into RxJava observables:

```
Button button = (Button)
findViewById(R.id.button);
RxView.clicks(button).subscribe(x -> {
  // do work here
});
```

RxAndroid

Control on which threads you observe and react to events (avoid long computations on the main thread):

```
Observable.just("RebelLabs")
.subscribeOn(Schedulers.newThread())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(anObserver);
```

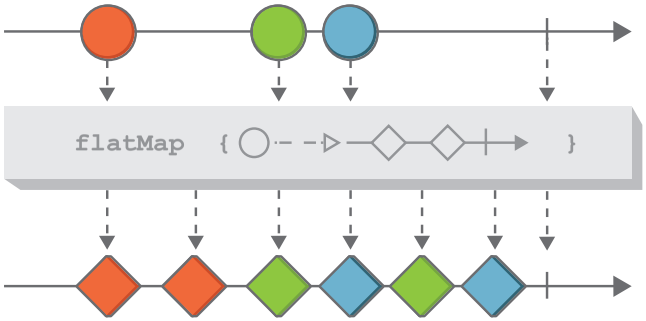
Data processing functions

`map(Function<? super T,? extends R> mapper)` - applies a function to each of items, and emits the returned values.

`filter(Predicate<? super T> predicate)` - emits only the items satisfying a predicate.

`buffer(int count)` - emits lists of the items of the specified size.

`zip(ObservableSource s1, ObservableSource s2, BiFunction<T1, T2, R> f)` - applies a function to the items from multiple observables and emits the returned value.



`flatMap(Function<? super T,? extends ObservableSource<? extends R>> mapper)` - takes a function from items to an *Observable*, emits the items of the resulting *Observables*

.

`groupBy(Function<? super T,? extends K> keySelector)` - emits items grouped by a specified key selector function.

`timeout(long timeout, TimeUnit timeUnit)` - emits items of the original *Observable*. If the next item isn't emitted within the specified timeout, a *TimeoutException* occurs.

Subscribing to observables

Observers provide a mechanism for receiving data and notifications from *Observables* using the following API:

`onNext(T t)` - provides the *Observer* with a new item to observe.

`onError(Throwable e)` - notifies the *Observer* that the Observable has experienced an error condition.

`onComplete()` - notifies the *Observer* that the *Observable* has finished sending push-based notifications.

RxLifecycle

Bind subscription lifecycle to Android components. Destroy subscriptions and avoid memory leaks on destroy / pause events.

```
myObservable.compose(
  RxLifecycleAndroid.bindActivity(lifecycle))
  .subscribe();
```

Testing observables

`TestSubscriber` - a subscriber that records events that you can make assertions upon.

`TestObserver` - an *Observer* that records events that you can make assertions upon.

```
TestSubscriber<Integer> ts =
Flowable.range(1, 5).test();
// assert properties
assertThat(
      ts.values()).hasSize(5));
```

Spring Boot and Web annotations

Use annotations to configure your web application.

T **@SpringBootApplication** - uses @Configuration, @EnableAutoConfiguration and @ComponentScan.

T **@EnableAutoConfiguration** - make Spring guess the configuration based on the classpath.

T **@Controller** - marks the class as web controller, capable of handling the requests. T **@RestController** - a convenience annotation of a @Controller and @ResponseBody.

M T **@ResponseBody** - makes Spring bind method's return value to the web response body.

M **@RequestMapping** - specify on the method in the controller, to map a HTTP request to the URL to this method.

P **@RequestParam** - bind HTTP parameters into method arguments.

P **@PathVariable** - binds placeholder from the URI to the method parameter.

Spring Cloud annotations

Make you application work well in the cloud.

T **@EnableConfigServer** - turns your application into a server other apps can get their configuration from.

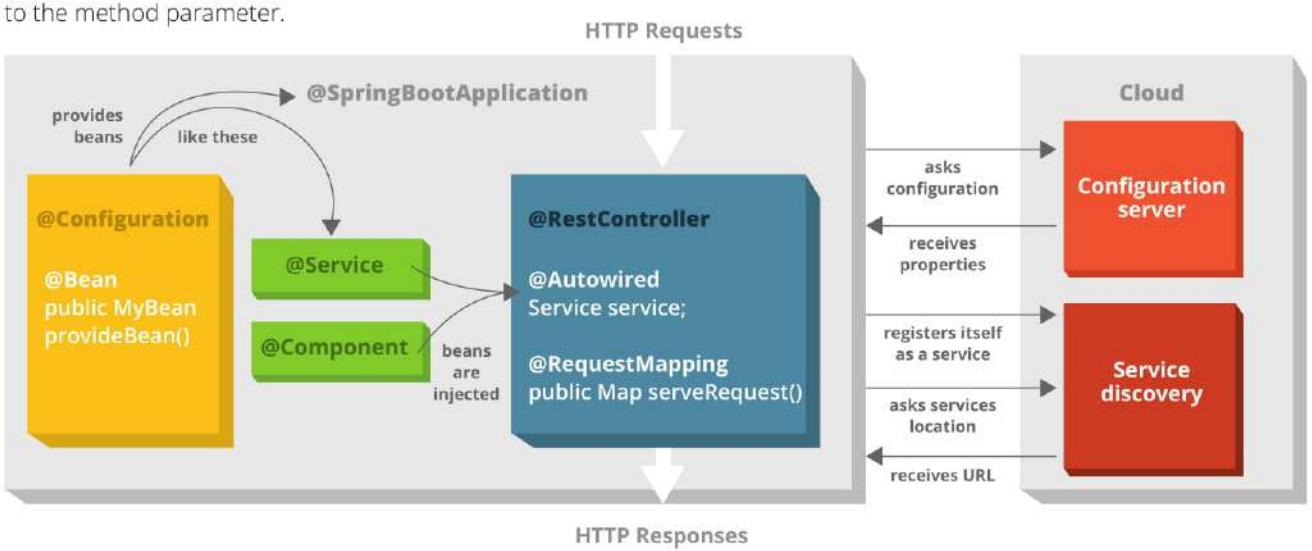
Use `spring.application.cloud.config.uri` in the client @SpringBootApplication to point to the config server.

T **@EnableEurekaServer** - makes your app an Eureka discovery service, other apps can locate services through it.

T **@EnableDiscoveryClient** - makes your app register in the service discovery server and discover other services through it.

T **@EnableCircuitBreaker** - configures Hystrix circuit breaker protocols.

M **@HystrixCommand(fallbackMethod = "fallbackMethodName")** - marks methods to fall back to another method if they cannot succeed normally.



Spring Framework annotations

Spring uses dependency injection to configure and bind your application together.

T **@ComponentScan** - make Spring scan the package for the @Configuration classes.

T **@Configuration** - mark a class as a source of bean definitions.

M **@Bean** - indicates that a method produces a bean to be managed by the Spring container.

T **@Component** - turns the class into a Spring bean at the auto-scan time. T **@Service** - specialization of the @Component, has no encapsulated state.

C F M **@Autowired** - Spring's dependency injection wires an appropriate bean into the marked class member.

T M **@Lazy** - makes @Bean or @Component be initialized on demand rather than eagerly.

C F M **@Qualifier** - filters what beans should be used to @Autowire a field or parameter.

C F M **@Value** - indicates a default value expression for the field or parameter, typically something like `"#{systemProperties.myProp}"`

C F M **@Required** - fail the configuration, if the dependency cannot be injected.

Legend

- T - class
- F - field annotation
- C - constructor annotation
- M - method
- P - parameter

SQL Cheat Sheet

JRebel | XRebel

Basic Queries

- filter your columns
  SELECT col1, col2, col3, ... **FROM** table1
- filter the rows
  WHERE col4 = 1 **AND** col5 = 2
- aggregate the data
  GROUP by ...
- limit aggregated data
  HAVING count(*) > 1
- order of the results
  ORDER BY col2

USEFUL KEYWORDS FOR SELECTS:

DISTINCT - return unique results
BETWEEN a **AND** b - limit the range, the values can be numbers, text, or dates
LIKE - pattern search within the column text
IN (a, b, c) - check if the value is contained among given.

Data Modification

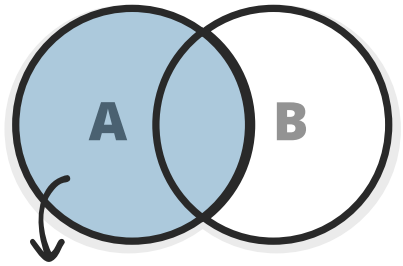
- update specific data with the **WHERE** clause
  UPDATE table1 SET col1 = 1 **WHERE** col2 = 2
- insert values manually
  INSERT INTO table1 **(ID, FIRST_NAME, LAST_NAME)**
  VALUES (1, 'Rebel', 'Labs');
- or by using the results of a query
  INSERT INTO table1 **(ID, FIRST_NAME, LAST_NAME)**
  SELECT id, last_name, first_name **FROM** table2

Views

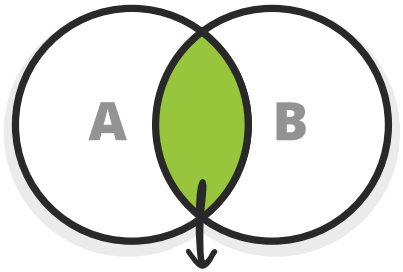
A **VIEW** is a virtual table, which is a result of a query.
They can be used to create virtual tables of complex queries.

```
CREATE VIEW view1 AS
SELECT col1, col2
FROM table1
WHERE ...
```

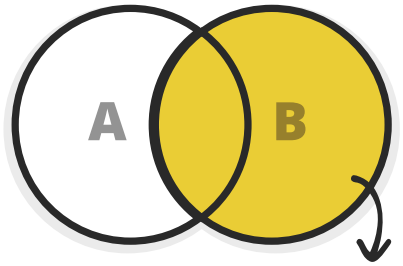
The Joy of JOINs



LEFT OUTER JOIN - *all rows from table A, even if they do not exist in table B*



INNER JOIN - *fetch the results that exist in both tables*



RIGHT OUTER JOIN - *all rows from table B, even if they do not exist in table A*

Updates on JOINed Queries

You can use **JOINs** in your **UPDATEs**

```
UPDATE t1 SET a = 1
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id
WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

NB! Use database specific syntax, it might be faster!

Semi JOINs

You can use subqueries instead of **JOINs:**

```
SELECT col1, col2 FROM table1 WHERE id IN
  (SELECT t1_id FROM table2 WHERE date >
    CURRENT_TIMESTAMP)
```

Indexes

If you query by a column, index it!

```
CREATE INDEX index1 ON table1 (col1)
```

DON'T FORGET:

Avoid overlapping indexes
Avoid indexing on too many columns
Indexes can speed up **DELETE** and **UPDATE** operation

Useful Utility Functions

- convert strings to dates:
  TO_DATE (Oracle, PostgreSQL), **STR_TO_DATE** (MySQL)
- return the first non-**NULL** argument:
  COALESCE (col1, col2, "default value")
- return current time:
  CURRENT_TIMESTAMP
- compute set operations on two result sets

```
SELECT col1, col2 FROM table1
UNION / EXCEPT / INTERSECT
SELECT col3, col4 FROM table2;
```

Union - returns data from both queries
Except - rows from the first query that are not present in the second query
Intersect - rows that are returned from both queries

Reporting

Use aggregation functions

COUNT - return the number of rows
SUM - cumulate the values
AVG - return the average for the group
MIN / MAX - smallest / largest value

Java 8 Best Practices Cheat Sheet

Default methods

Evolve interfaces & create traits

```
//Default methods in interfaces
@FunctionalInterface
interface Utilities {
  default Consumer<Runnable> m() {
    return (r) -> r.run();
  }
  // default methods, still functional
  Object function(Object o);
}
class A implements Utilities { // implement
  public Object function(Object o) {
    return new Object();
  }
  {
    // call a default method
    Consumer<Runnable> n = new A().m();
  }
}
```

Lambdas

Syntax:
(parameters) -> expression
(parameters) -> { statements; }

```
// takes a Long, returns a String
Function<Long, String> f = (l) -> l.toString();
// takes nothing gives you Threads
Supplier<Thread> s = Thread::currentThread;
// takes a string as the parameter
Consumer<String> c = System.out::println;

// use them with streams
new ArrayList<String>().stream().
// peek: debug streams without changes
peek(e -> System.out.println(e)).
// map: convert every element into something
map(e -> e.hashCode()).
// filter: pass some elements through
filter (hc -> (hc % 2) == 0).
// collect all values from the stream
collect(Collectors.toCollection(TreeSet::new))
```

java.util. Optional

A container for possible null values

```
// Create an optional
Optional<String> optional =
Optional.ofNullable(a);
// process the optional
optional.map(s -> "RebelLabs:" + s);
// map a function that returns Optional
optional.flatMap(s -> Optional.ofNullable(s));

// run if the value is there
optional.ifPresent(System.out::println);
// get the value or throw an exception
optional.get();

// return the value or the given value
optional.orElse("Hello world!");
// return empty Optional if not satisfied
optional.filter(s -> s.startsWith("RebelLabs"));
```

Rules of Thumb

Traits: 1 default method per interface
Don't enhance functional interfaces
Only conservative implementations

Expressions over statements
Refactor to use method references
Chain lambdas rather than growing them

Fields - use plain objects
Method parameters, use plain objects
Return values - use Optional
Use *orElse()* instead of *get()*