Spis treści

[1. Java 8 4](#_Toc5447974)

[1.1. Modyfikatory dostępu 4](#_Toc5447975)

[1.2. Interfejsy i klasy abstrakcyjne 4](#_Toc5447976)

[1.3. Interfejsy funkcyjne 4](#_Toc5447977)

[1.4. Metody domyślne (defaultowe) 4](#_Toc5447978)

[1.5. Klasa Immutable 5](#_Toc5447979)

[1.6. Hashcode & Equals 5](#_Toc5447980)

[1.7. Obiekty związane z datami 5](#_Toc5447981)

[1.7.1. LocalDate 6](#_Toc5447982)

[1.7.2. LocalTime 6](#_Toc5447983)

[1.7.3. LocalDateTime 6](#_Toc5447984)

[1.7.4. TemporalAdjuster 6](#_Toc5447985)

[1.8. Kolekcje (sortowanie kolekcji, comparator, comparable) 7](#_Toc5447986)

[1.9. Operacje na Stringach 8](#_Toc5447987)

[ Transformacja char na String 8](#_Toc5447988)

[1.10. Wyrażenia lambda 10](#_Toc5447989)

[1.11. Stream i jego własności 10](#_Toc5447990)

[1.12. Wzorce projektowe 11](#_Toc5447991)

[1.12.1. Strategy 11](#_Toc5447992)

[1.12.2. Observer 12](#_Toc5447993)

[1.12.3. Factory 12](#_Toc5447994)

[1.12.4. Decorator 12](#_Toc5447995)

[1.12.5. Singleton 12](#_Toc5447996)

[1.12.6. Builder 12](#_Toc5447997)

[1.12.7. Prototype 12](#_Toc5447998)

[1.12.8. Facade 13](#_Toc5447999)

[1.12.9. Adapter 13](#_Toc5448000)

[1.13. JVM – zarządzanie pamięcią, stack, heap 13](#_Toc5448001)

[1.14. Gradle i Maven 13](#_Toc5448002)

[1.14.1. Maven 13](#_Toc5448003)

[1.14.2. Gradle 14](#_Toc5448004)

[1.14.3. Maven Central 14](#_Toc5448005)

[1.15. Garbage Collector 14](#_Toc5448006)

[1.16. Wyrażenia regularne – regular expression 15](#_Toc5448007)

[2. SQL 17](#_Toc5448008)

[2.1. Typy danych w MySQL 17](#_Toc5448009)

[2.2. Zapytania i komendy (SELECT, DELETE, JOIN) 18](#_Toc5448010)

[2.2.1. Tworzenie tabel 18](#_Toc5448011)

[2.2.2. INSERT 18](#_Toc5448012)

[2.2.3. COMMIT 19](#_Toc5448013)

[2.2.4. SELECT 19](#_Toc5448014)

[2.2.5. WHERE 19](#_Toc5448015)

[2.2.6. Złączenia tabel – komenda JOIN 19](#_Toc5448016)

[2.2.7. UPDATE i SET – aktualizowanie danych 21](#_Toc5448017)

[2.2.8. DELETE 21](#_Toc5448018)

[2.3. Czym jest indeks i jak go tworzyć 21](#_Toc5448019)

[3. Spring 23](#_Toc5448020)

[3.1. Bean 23](#_Toc5448021)

[3.1.1. Rodzaje komponentów 23](#_Toc5448022)

[3.1.2. @Configuration 24](#_Toc5448023)

[3.1.3. Scope 24](#_Toc5448024)

[3.1.4. Automatyczna konfiguracja beanów 24](#_Toc5448025)

[3.2. SpringBoot 24](#_Toc5448026)

[3.3. Wzorzec projektowy MVC 24](#_Toc5448027)

[3.4. REST API, zapytania HTTP 25](#_Toc5448028)

[3.4.1. JSON 26](#_Toc5448029)

[3.4.2. Walidacja danych 26](#_Toc5448030)

[3.4.3. Endpointy 27](#_Toc5448031)

[3.5. Aspekty 27](#_Toc5448032)

[3.5.1. Porady 27](#_Toc5448033)

[3.5.2. Punkty złączenia 28](#_Toc5448034)

[3.5.3. Punkty przecięcia 28](#_Toc5448035)

[4. Hibernate 29](#_Toc5448036)

[4.1. ORM – Mapowanie obiektowo-relacyjne 29](#_Toc5448037)

[4.2. Różnica między Hibernate a JPA 29](#_Toc5448038)

[4.3. Encje 29](#_Toc5448039)

[4.3.1. Mapowanie na tabele, kolumny 30](#_Toc5448040)

[4.3.2. Relacje między encjami 30](#_Toc5448041)

[4.4. HQL 31](#_Toc5448042)

[4.4.1. NamedQuery czyli zapytania HQL 31](#_Toc5448043)

[4.5. API Criteria 31](#_Toc5448044)

[4.6. @Transactional 32](#_Toc5448045)

[4.6.1. Optymistic and Pesymistic Locking 32](#_Toc5448046)

[4.7. Cash pierwszego i drugiego poziomu 32](#_Toc5448047)

[5. Git 33](#_Toc5448048)

[5.1. Git jako „migawka” projektu 33](#_Toc5448049)

[5.2. Branch, merge, cherry-pick 34](#_Toc5448050)

# Java 8

## Modyfikatory dostępu

Private – Widoczność w obrębie klasy

Protected – Widoczność w pakiecie i wszystkich podklasach

Default – widoczność w obrębie pakietu (niefortunne, niestosowane)

Public – Widoczność wszędzie

## Interfejsy i klasy abstrakcyjne

Interfejs – Zestaw wymagań, które muszą być spełnione, aby klasa została uznana za zgodną z danym interfejsem. Wszystkie metody w interfejsie są od razu publiczne. Interfejs jest też gwarantem implementacji.

Klasa abstrakcyjna – jest to klasa, której obiektu nie można stworzyć, stanowi ona podstawę do tworzenia innych klas poprzez zaimplementowanie w niej podstawowej funkcjonalności.

Różnica – klasa może rozszerzać tylko jedną klasą, ale za to może implementować wiele interfejsów

## Interfejsy funkcyjne

Interfejsy posiadające tylko jedną metodę abstrakcyjną. Pozwalają na wykorzystanie wyrażeń lambda w miejsce, gdzie kompilator oczekuje zadanego typu danych. Można go stworzyć za pomocą adnotacji @FunctionalInterface. Przykładami interfejsów funkcyjnych są: Comparator, EventHandler, ActionListener.

* Comparator – interfejs porównujący ze sobą obiekty
* EventHandler - zapewnia wsparcie dla dynamicznego generowania zdarzeń, których metody wykonują prostą instrukcję obejmującą przychodzący obiekt zdarzenia i obiekt docelowy.
* ActionListener – interfejs nasłuchujący do odbierania zdarzeń.

## Metody domyślne (defaultowe)

Każda metoda opisana w interfejsie może mieć interpretację domyślną, którą należy oznaczyć modyfikatorem default. Wtedy w klasie implementującej interfejs nie należy jej definiować w klasie implementującej interfejs.

## Klasa Immutable

Jest to klasa, na podstawi której stworzony obiekt jest niezmienny przez całe jego życie w programie.

Zalety:

* Są łatwiejsze w użyciu i testowaniu
* Można je bezpiecznie używać w Setach lub jako klucz w Mapach
* Mogą być łatwo cachowane
* Immutable object mogą być bezpiecznie używane w programowaniu wielowątkowym. Stan tych obiektów nie może ulec zmianie, więc mamy pewność, że każdy wątek widzi aktualny stan obiektu

Wady:

* Inicjalizacja wszystkich pól przez konstruktory. Ponieważ wszystkie nasze pola są oznaczone jako final, muszą więc zostać zainicjalizowane w konstruktorze.
* Problem z wydajnością – za każdym razem, gdy chcemy wprowadzić zmianę w naszym obiekcie to sprowadza się to do utworzenia nowego obiektu. Może to być odczuwalne zarówno w czasie działania aplikacji, jak i zużyciu pamięci.

## Hashcode & Equals

Hashcode – Metoda hashCode służy w Javie do zwrócenia unikalnej wartości liczbowej (typu int) dla każdego unikalnego obiektu. Skróty łańcuchów liczbowych są pochodnymi ich zawartości.

Equals – Metoda Equals klasy Object porównuje dwa obiekty i sprawdza czy są sobie równe.

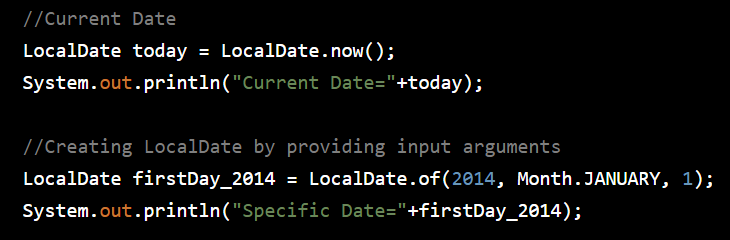
Kontrakt Hashcode i Equals – Jeżeli wartość hashCode obiektów jest taka sama, to obiekty mogą być równe lub różne (czyli equals zwróci true lub false), natomiast jeśli hashcode zwróci false to equals na pewno zwróci false.

## Obiekty związane z datami

W Javie 8 najważniejszym pakietem odnośnie czasu i dat jest pakiet java.time. Poniżej przedstawione są wybrane klasy z tego pakietu.

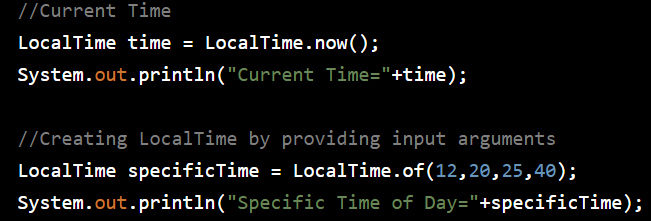
### LocalDate

Niemutowalna klasa reprezentująca datę w formacie yyyy-MM-dd. Można użyć jej metody now(), aby uzyskać obecną datę. Można też do niej wprowadzić własne argumenty, aby utworzyć obiekt przechowujący daną datę.



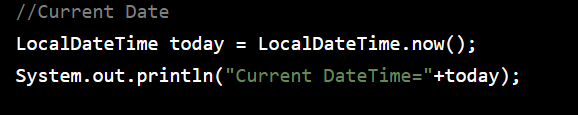
### LocalTime

Podobnie jak LocalDate niemutowalna klasa reprezentująca czas w formacie hh:mm:ss:zzzzz. Również można zastosować metodę now() lub ustawić konkretny czas.



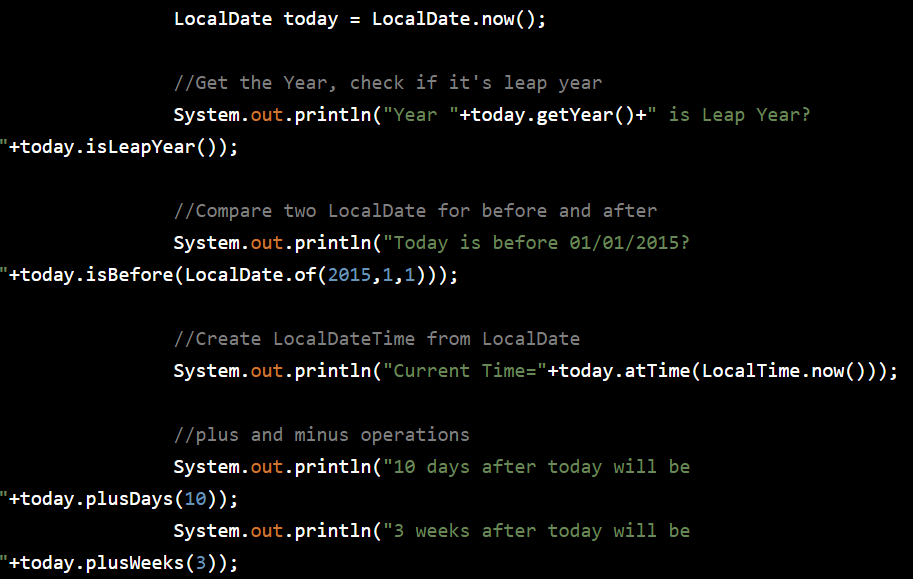
### LocalDateTime

Połączenie dwóch powyższych klas.



### TemporalAdjuster

W tym pakiecie są dostępne różne metody sprawdzające np. czy data dzisiejsza jest przed określoną datą lub czy zadany rok jest przestępny.



## Kolekcje (sortowanie kolekcji, comparator, comparable)

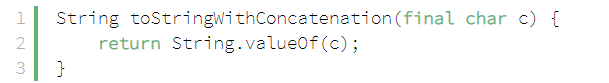
* Interfejs Iterable – Umożliwia użycie na klasach go implementujących pętli for each

Interfejs Collection – umożliwia wykonywanie podstawowych operacji na kolekcjach (np. dodawanie, usuwanie, sprawdzanie rozmiaru itd.)

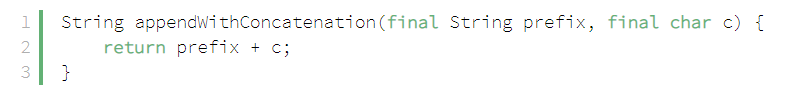
* List – umożliwia wykonywanie na kolekcjach zorganizowanych (np. pobieranie obiektu z konkretnej pozycji z listy)
* ArrayList – Najwygodniejsza przy małych zbiorach danych, gdyż bardzo szybko można odczytać z niej dane. Przy dużych zbiorach bardzo długo zajmuje wstawianie nowych i usuwanie starych elementów. Dziedziczy po wcześniejszych interfejsach,
* LinkedList – Elementy w tej liście zawierają referencje do obiektów poprzedzających i następnych na liście. Odczyt danych z niej jest dużo dłuższy niż w ArrayList, lecz usuwanie i dodawanie obiektów następuje dużo szybciej, Dziedziczy również po interfejsach Iterable, Collection i List.
* Map – Struktura danych przechowująca kolekcje par typu key-value. Zarówno klucz jak i wartość to są to obiekty, a nie typy prymitywne. Dany klucz daje dostęp do wartości pojedynczego entry (entry – jedna para key-value).
* HashMap – Jest to mapa opierająca się na założeniu, że cała kolekcja par zawartych w mapie dzieli się na mniejsze zbiory zwane kubełkami. Kubełki są dzielone na kategorie na podstawie wartości obiektów Key, a nie Value. Jest to swoiste połączenie ArrayList i LinkedList, gdyż tablica kubełków jest ArrayListą, natomiast lista par, kryjąca się pod danym kubełkiem jest LinkedListą. HashMap dzieli pary zawarte w kolekcji na poszczególne kategorie za pomocą metody hashCode. Należy tą metodę nadpisać tak aby spełniała założenia aplikacji, gdyż dziedziczona jest z klasy nadrzędnej Object. Również po jednym kubełkiem nie mogą się kryć dwie wartości pod jednym kluczem, co blokuje metoda equals, porównująca czy dane obiekty są identyczne.
* HashSet – analogicznie do HashMap posiada on listę kubełków i obiektów do nich przypisanych, lecz w HashSet nie ma już par obiektów.
* Interfejs Comparable – przeznaczony jest dla obiektów z „Naturalnym uporządkowaniem”, czyli obiekt sam musi wiedzieć w jakim porządku musi stać (np. uczniowie stojący w rzędzie po numerze dziennika). Interfejs ten porównuje własność this obiektu z innym obiektem w nadpisanej metodzie compareTo.
* Interfejs Comparator – Comparator jest definiowany przez osobną klasę, w której porównywane są dwa obiekty.
* Jeśli jakakolwiek klasa implementuje interfejs Comparable lub zdefiniowany jest w innej klasie Comparator, to zbiór danych obiektów (np. List, Set), może być sortowany automatycznie za pomocą metody statycznej Collections.sort(), a obiekty będą sortowane wg naturalnej kolejności (wg metody CompareTo klasy z zaimplementowanym interfejsem Comparable) lub wg tej zdefiniowanej w komparatorze z zaimplementowanym intefejsem Comparator. Podsumowując, jeśli sortowanie obiektów musi opierać się na naturalnym porządku, należy użyć interfejsu Comparable, podczas gdy sortowanie należy wykonać na atrybutach różnych obiektów, a następnie użyć interfejsu Comparator.

## Operacje na Stringach

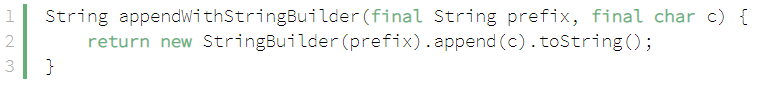
### Transformacja char na String



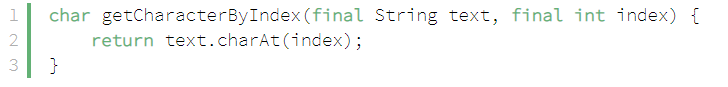
* „Dodawanie” Stringów (appending)
* Ze znakiem +



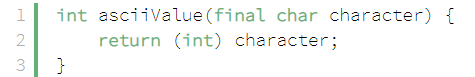
* Za pomocą klasy StringBuilder



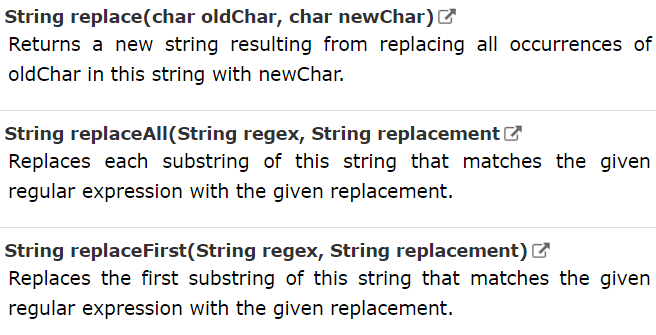
* Pobieranie znaku za pomocą indeksu



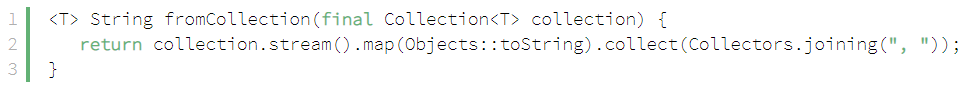
* Pobieranie wartości ASCII



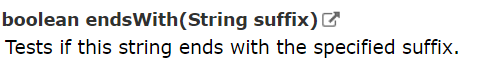
* Zmiana zadanych wartości na inne – metoda replace



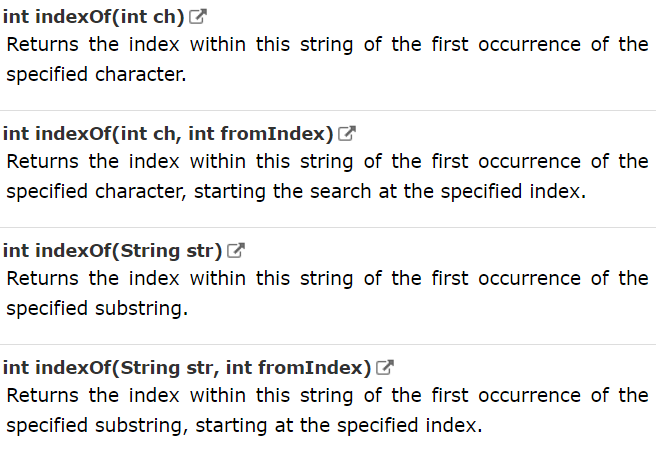
* Collectors Joining to String



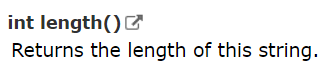
* Sprawdzenie czy String kończy się daną wartością



* Zwraca index danego znaku lub danej części Stringa



* Zwraca długość stringa



* Przekształca stringa na małe bądź duże litery



## Wyrażenia lambda

Dla uproszczenia, można powiedzieć, że wyrażenie lambda jest metodą przypisaną do zmiennej. Można ją też wywołać lub przekazać jako argument do innej metody. Wyrażenie lambda ma następującą składnię:

Wyrażenia lambda są bardzo pomocne na przy operacji na kolekcjach. Ich zaletą jest również duża zwięzłość (kod zajmuje mniej miejsca przez co jest bardziej czytelny).

Operator :: powoduje odwołanie do metod dla każdego elementu w kolekcji.

## Stream i jego własności

Stream służy do przetwarzania danych. Można w nim zawrzeć praktycznie dowolny zestaw danych. Metoda stream() jest metodą domyślną zaimplementowaną w interfejsie Collection. Pozwala ona na utworzenie strumienia na podstawie danych znajdujących się w danej kolekcji. Z kolei interfejs Stream jest interfejsem generycznym, przechowującym informacje o typie danych, które znajdują się w strumieniu.

Tworzenie strumieni:

* Strumień na podstawie kolekcji
* Strumień na podstawie tablicy
* Strumień typów prostych – IntStream, DoubleStream, LongStream
* Strumień danych losowych
* Strumień danych z pliku

Najczęstsze operacje na strumieniach:

* Filter – zwraca strumień zawierający tylko te elementy, dla których filtr zwrócił wartość true
* Map – każdy z elementów może zostać zmieniony do innego typu, nowy obiekt jest zawarty w nowym strumieniu
* Peek – pozwala przeprowadzić operację na każdym elemencie w strumieniu, zwraca strumień z tymi samymi elementami
* Limit – Zwraca strumień ograniczony do zadanej liczby elementów, pozostałe są ignorowane.
* FlatMap – dokonuje „spłaszczenia” strumienia, czyli podczas działania strumienia na pojedynczych obiektach, czasem te obiekty posiadają kolekcje jako właściwości. FlatMap udostępnia iterowanie po kolekcjach zagnieżdżonych w kolekcji, tzn. zwraca obiekty z zagnieżdżonej kolekcji i przenosi je do głównego strumienia.

Kończenie Strumieni:

* forEach – wykonuje żądane zadanie dla każdego elementu
* count – zwraca liczbę elementów w strumieniu
* allMatch – zwraca flagę informującą czy wszystkie elementy w strumieniu spełniają zadany warunek, przestaje sprawdzać po pierwszym elemencie, który danego warunku nie spełnia.
* Sum – zwraca sumę wartości w strumieniu
* Collect – pozwala na utworzenie kolekcji wynikowej po przetworzeniu danych przes strumień
  + Collectors.toList() – tworzy kolekcję w postaci listy
  + Collectors.toMap(key,value) – tworzy kolekcję w postaci mapy o zadanym kluczu i wartości
  + Collectors.joining(String delimeter, String prefix, String suffix) – tworzy String, który jest konkatenacją (połączeniem) wszystkich obiektów w strumieniu w jeden ciąg znaków. Będzie on poprzedzony prefixem, zakończony suffixem, a obiekty będą oddzielone delimetrem.

## Wzorce projektowe

### Strategy

Wzorzec strategia definiuje rodzinę algorytmów, pakuje je jako osobne klasy i powoduje, że są one w pełni wymienne. Zastosowanie tego wzorca pozwala na to, aby zmiany w implementacji algorytmów przetwarzania były całkowicie niezależne od strony klienta, który z nich korzysta ( Rusz głową – symulator kaczki)

### Observer

Wzorzec obserwator definiuje pomiędzy obiektami relację jeden do wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan to wszystkie jego obiekty zależne zostają o tym poinformowane i automatycznie zaktualizowane (Rusz głową – stacja pogodowa).

### Factory

Wzorzec metody fabrykującej definiuje interfejs pozwalający na tworzenie obiektów, ale pozwala klasom podrzędnym decydować, jakiej klasy obiekt zostanie utworzony. Wzorzec ten przekazuje odpowiedzialność za tworzenie obiektów do klas podrzędnych (Rusz głową – fabryka pizzy)

### Decorator

Wzorzec dekorator pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań. Dekoratory daj a elastyczność podobną do tej, jaką daje dziedziczenie, oferując jednak znacznie rozszerzoną funkcjonalność (Rusz głową – kawiarnia)

### Singleton

Wzorzec singleton zapewnia to, że dana klasa będzie miała tylko i wyłącznie jedną instancję obiektu, i zapewnia globalny punkt dostępu do tej instancji (Rusz głową – czekoladowy kocioł).

### Builder

Umożliwia tworzenie obiektów w procedurze wielokrokowej i nie narzuca tej procedury. Ukrywa wewnętrzną reprezentację produktu przed klientem. Implementuje produkty, które mogą być wymieniane, ponieważ klient korzysta z abstrakcyjnego interfejsu (Kodilla – Pizza).

### Prototype

Prototyp umożliwia tworzenie nowych egzemplarzy i kopiowane tych, które już istnieją (oznacza to użycie metody clone()). Kluczowym aspektem jest to, że klient może tworzyć nowe egzemplarze bez wiedzy o tym, którym egzemplarzem naprawdę operuje, czyli zapewnia klientowi możliwość generowania obiektów, których typ niekoniecznie jest znany. W pewnych warunkach również, kopiowanie obiektów, może być wydajniejsze niż tworzenie nowych.

### Facade

Zapewnia jeden zunifikowany interfejs dla całego zestawu interfejsów określonego podsystemu. Tworzy nowy interfejs wysokiego poziomu.

### Adapter

Dokonuje konwersji interfejsu danej klasy do postaci innego interfejsu, zgodnego z oczekiwaniami klienta. Pozwala na wzajemną współpracę klas, które ze względu na niekompatybilne interfejsy wcześniej nie mogły ze sobą współpracować.

## JVM – zarządzanie pamięcią, stack, heap

JVM – Java Virtual Machine – Wirtualna Maszyna Javy to pomocniczy program, który zrozumie kod bajtowy wyrzucony przez kompilator. Jest to medium, w którym kod bajtowy otrzymany od kompilatora jest zrozumiały i uruchomi na maszynie program w takiej formie, że ta będzie wiedzieć jak go wykonywać. JVM również zarządza pamięcią

Heap(Sterta) – na stercie alokowane są wszystkie obiekty jakie tworzymy i dostęp do sterty odbywa się poprzez wskazanie komórki pamięci (przez referencje). Stertę współdzielą ze sobą wszystkie wątki programu. Przechowuje ona również typy złożone jak np. String, Array, a także zmienne statyczne inicjalizowane przy starcie programu.

Stack(Stos) – Na stosie odkładane są zmienne lokalne (typy proste jak i referencyjne) oraz tworzone są ramki wywołań (czyli wywołań metod danego wątku). Każdy z wątków ma swój własny oddzielny stos. Dlatego zmienne zawarte na stosie nie mogą być współdzielone przez wątki.

StringPool –  Jest różnica pomiędzy String s = "kot" a String s1 = new String("kot"). StringPool to właśnie element na stercie do którego trafiają zmienne łańcuchowe bez słówka new.

## Gradle i Maven

### Maven

Narzędzie, którego sterowanie oparte jest na języku XML. Maven ma sporo możliwości jak np. budowanie projektu, tworzenie dokumentacji, pobieranie zewnętrznych bibliotek, wykonywanie testów automatycznych. Główny plik Mavena – pom.xml składa się z kilku cząści:

* Nagłowek – standardowy nagłówek pliku XML
* Deklaracja podstawowych informacji o projekcie :
  + groupId – główny pakiet projektu
  + artifactId – nazwa projektu
  + version – wersja projektu
* Rodzic projektu – pewnego rodzaju dziedziczenie po innym projekcie
* Właściwości – informacje o projekcie (np. narzucenie konkretnej wersji Javy dla projektu).
* Zależności – definiowanie zewnętrznych bibliotek, które będą użyte w projekcie
* Build – informacja w jaki sposób projekt ma być zbudowany

### Gradle

Gradle jest to bardzo podobne narzędzie do Mavena, lecz dużo bardziej przejrzyste. Odpowiednikiem pliku pom.xml dra gradle jest plik build.gradle składający się z:

* BulidScript – blok wymagań dla pliku, definiujący z czego można skorzystać w obrębie projektu
* Plugins – rozszerzenia projektu
* Właściwości – informacje o projekcie
* Repozytoria – deklaracje repozytoriów, w których można szukać zależności
* Zależności - definiowanie zewnętrznych bibliotek, które będą użyte w projekcie

### Maven Central

Jest to repozytorium, będące głównym miejscem składowania różnego rodzaju bibliotek. MAVEN CENTRAL TO NIE MAVEN!!!!

## Garbage Collector

Jest to po prostu odśmiecanie pamięci. Jedna z metod automatycznego zarządzania przydzieloną pamięcią, w której za proces jej zwalniania odpowiedzialny jest nie programista tylko właśnie programowy zarządca nazwany garbage collector.

Przykładowe metody odśmiecania:

* Zliczanie referencji – w metodzie tej alokowane są obiekty posiadające dodatkowe pola, wykorzystywane do zliczania odwołań do danego obiektu.  Podczas alokowania obiektu pole to ustawiane jest na 1, następnie za każdym razem, gdy do obiektu dodawane jest odwołanie, licznik ten jest zwiększany o jeden, a gdy odwołanie jest usuwane, licznik jest zmniejszany o jeden. Wyzerowanie licznika oznacza, że w programie nie istnieje żadne odwołanie do tego obiektu – nie jest on używany oraz nie ma możliwości ponownego, poprawnego odwołania się do niego, w związku z czym przydzielona mu pamięć może zostać zwolniona
* Mark and Sweep -  Przy zastosowaniu tej metody z każdym dynamicznie zaalokowanym obiektem związany jest tzw. markbit, określający czy dany obiekt jest jeszcze używany (markbit ustawiony na 1) czy też jest już niepotrzebny (ustawiony na 0). W metodzie tej pamięć nie jest odzyskiwana bezpośrednio po stwierdzeniu, że obiekt jest już niepotrzebny, lecz dopiero w momencie przekroczenia pewnego progu wykorzystania pamięci. Działanie programu jest wtedy zatrzymywane i uruchamiana jest faza odśmiecania. Działanie to dzieli się na dwa etapy jak sama nazwa wskazuje:
  + Mark - garbage collector  identyfikuje wszystkie obiekty, które są jeszcze wykorzystywane i ustawia ich markbity na 1
  + Sweep - garbage collector usuwa nieużywane obiekty (te, których markbit był ustawiony na 0) oraz resetuje markbit obiektów pozostawionych
* Odśmiecanie przez kopiowanie - Ta metoda polega na tym, że wszystko zostaje rekursywnie przekopiowane do innego obszaru w pamięci – kopiowany jest początkowy zestaw, potem wszystko co było przez niego wskazywane. Na końcu zwalniamy początkową pamięć.

W ten sposób oszczędza się na konieczności ustawiania bitów w mark and sweep, i regularnie defragmentuje się pamięć.

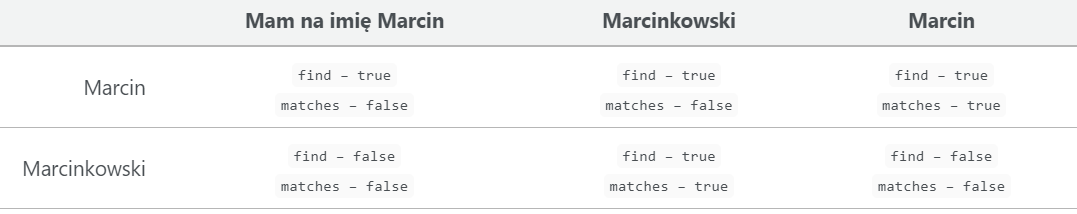
## Wyrażenia regularne – regular expression

Są to wzorce opisujące grupę łańcuchów znaków. Łańcuch znaków pasuje do wzorca, jeżeli wyrażenie regularnie go opisuje. Wzorzec ten jest kompilowany przez silnik wyrażeń regularnych.

Wyrażeń regularnych używa się do pracy z łańcuchami znaków. W praktyce jednym z głównych zastosowań jest weryfikacja czy dany łańcuch znaków pasuje do wzorca, którym właśnie jest wyrażenie regularne.

Wyrażenia regularne są obsługiwane przez dwie klasy z biblioteki standardowej: Pattern i Matcher. Klasa Matcher posiada następujące metody:

* Find() – zwraca true jeśli w łańcuchu znaków znajduje się coś co pasuje do wyrażenia regularnego
* Matches() – metoda zwraca true jeśli łańcuch znaków pasuje w całości do wyrażenia regularnego



# SQL

Strukturalny język zapytań używany do operacji na bazach danych.

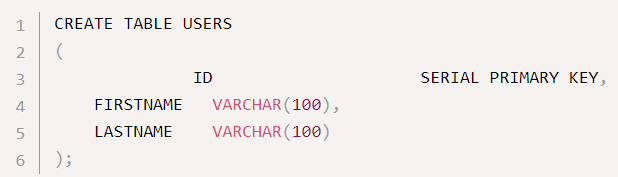
Bazy danych można określić jako tabele. Tabele posiadają kolumny oraz wiersze. Każdy wiersz określa jego klucz główny (id), który musi mieć unikalną wartość dla każdego wiersz (rekordu) w tabeli. Mogą również też posiadać klucz obcy czyli po prostu klucz główny z innej tabeli. Dzięki temu można określić relacje między tabelami czyli: jeden do jednego (każdemu rekordowi z tabeli A odpowiada dokładnie jeden rekord z tabeli B), jeden do wielu (każdemu rekordowi z tabeli A odpowiada kilka rekordów z tabeli B, lecz rekordom z tabeli B odpowiada tylko jeden rekord z tabeli A), a także wiele do wielu (czyli rekordom z tabeli A jest przypisane wiele rekordów z tabeli B i odwrotnie). W trakcie tworzenia tabel dodaje się do nich ograniczenia, czyli tzw. constrainty.

## Typy danych w MySQL

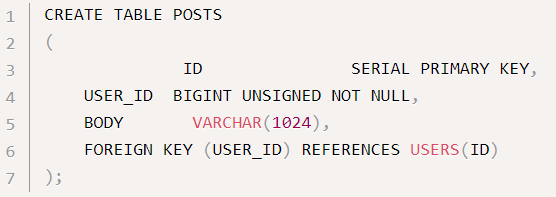
* Typy numeryczne
  + Unsigned – tylko dodatnie
  + Signed – ze znakiem (stosowany domyślnie)
  + Do typów numerycznych przypisuje się typy liczbowe jak: int, bigint, decimal, double
* Typ logiczny – boolean
* Typy tekstowe
  + Char – typ znakowy przechwujący od 0 do 255 znaków
  + Varchar – typ przechowujący od 0 do 65535 znaków
  + Enum – Typ znakowy umożliwiający wcześniej zdefiniowane wartości tekstowe
* Typy do przechowywania dat i czasu
  + Date – data w formacie YYYY-MM-DD
  + Time – Czas w formacie HH-MM-SS
  + Datetime – połączenie typów date i time

## Zapytania i komendy (SELECT, DELETE, JOIN)

### Tworzenie tabel



* CREATE TABLE <nazwa tabeli> - tworzenie tabeli o zadanej nazwie
* ID jest typu SERIAL co jest skrótem dla zapisu BIGINT UNSIGNET NOT NULL AUTO\_INCREMENT (rekordy automatycznie otrzymują kolejne wartości) UNIQUE, a także poprzez zapis PRIMARY KEY jest kluczem w tabeli
* Następnie deklarowane są nazwy poszczególnych kolumn i ich typów (w tym przypadku są to kolumny FIRSTNAME i LASTNAME typu varchar przechowującej nie więcej niż 100 znaków)



* FOREIGN KEY (<nazwa\_wcześniej\_zadeklarowanej\_kolumny>) REFERENCES <nazwa\_tabeli>(<nazwa\_kolumny\_w\_tej\_tabeli>) – komenda ta określa, że jedna kolumna w tej tabeli będzie kluczem obcym z innej tabeli i będzie przyjmowała wartości z kolumny z tabeli, do której się odwołuje.

### INSERT



INSERT INTO <nazwa\_tabeli> (nazwa\_kolumny\_1, …, nazwa\_kolumny\_N) VALUES (wartość\_1, …, wartość\_N) – wstawia do tabeli dane wartości pod odpowiednimi kolumnami.

### COMMIT

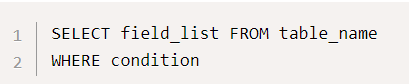
Wykonanie operacji COMMIT powoduje zatwierdzenie wcześniej wprowadzonych modyfikacji, i ich rzeczywiste wprowadzenie do baz danych.

### SELECT

Instrukcja SELECT pozwalająca na wyszukiwanie i pobieranie danych z bazy danych jest najczęściej stosowaną instrukcją języka SQL. Ważne jest to, że instrukcja SELECT nie modyfikuje danych, nie otwiera więc transakcji i nie trzeba używać COMMIT.

### WHERE

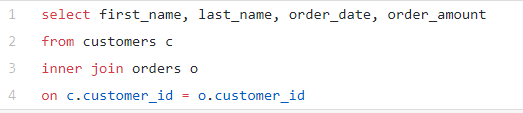
Instrukcja WHERE jest to po prostu instrukcja warunkowa. Występuje w połączeniu z instrukcją SELECT.

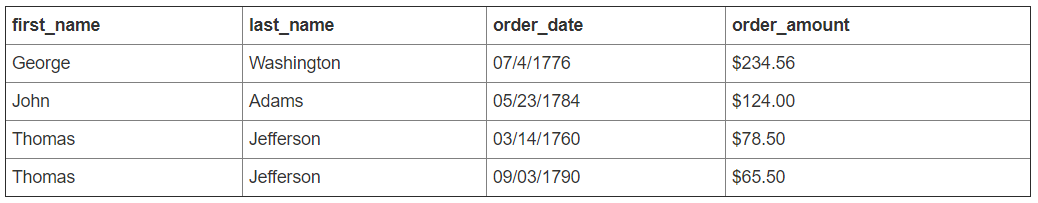


### Złączenia tabel – komenda JOIN

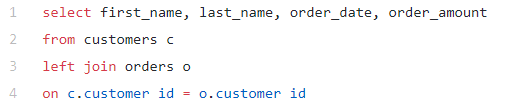
Komenda JOIN łączy tabele powiązane jakimiś relacjami. Komenda JOIN przypisuje wybrane rekordy z jednej tabeli i drugiej tabeli, wyświetlając je odpowiedznio je przyporządkowując. JOIN występuje w 4 wariantach: Inner, left, right i full.

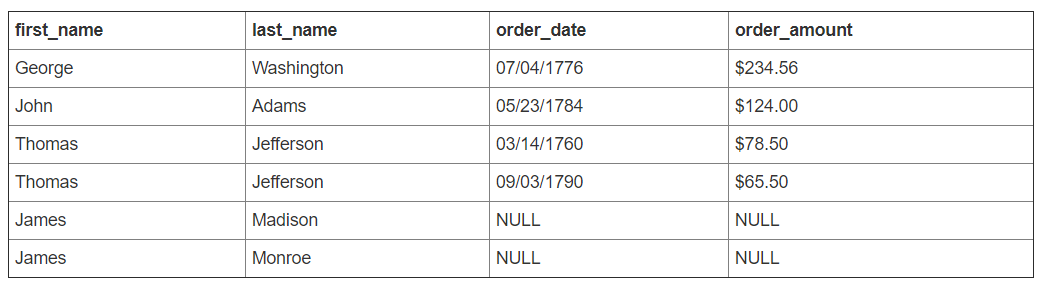
* Inner join – zwraca połączone rekordy dwóch tabel, lecz zwraca tylko rekordy, pomiędzy którymi istnieje jakaś relacja



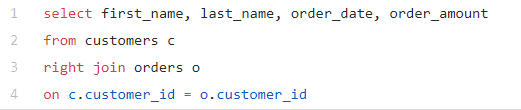


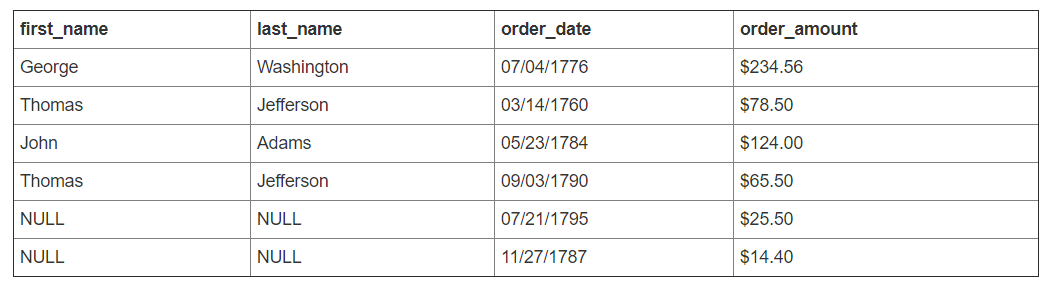
* Left join – zwraca połączone rekordy dwóch tabel, lecz zwraca wszystkie rekordy z tabeli A nawet jeśli nie są powiązane z rekordami z tabeli B, a do wartości w kolumnach z tabeli B wstawia wartość NULL



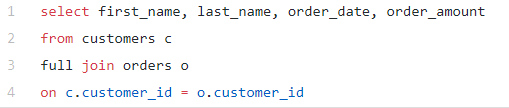


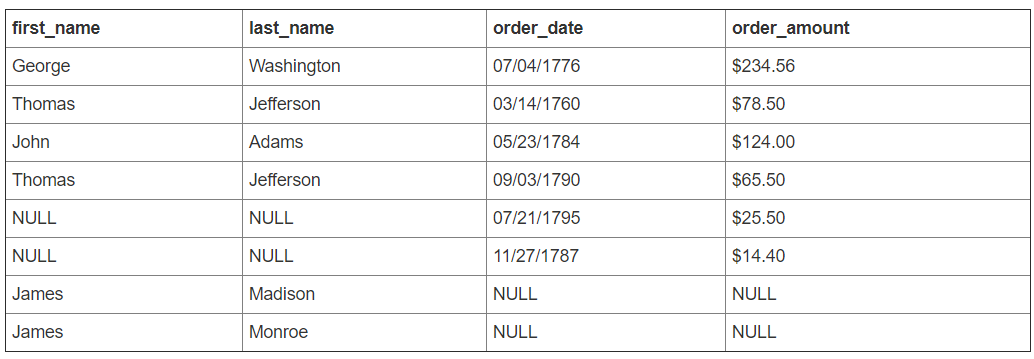
* Right join – analogicznie do left join, zwraca wszystkie rekordy z tabeli B, a do wartości w kolumnach wybranych z tabeli A przypisuje NULL, jeśli nie mają one powiązań



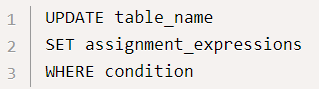


* Full join – Całkowite połączenie obydwu tabel

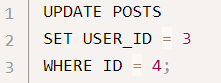




### UPDATE i SET – aktualizowanie danych



Chodzi o zaktualizowanie tabeli o nazwie table\_name, wartością assignment\_expressions(np. USER\_ID = 3), dla warunku condition.



Powyżej następuje aktualizacja tabeli POSTS, gdzie ustawiana jest wartość USER\_ID równe 3, dla rekordu o ID = 4.

### DELETE



Usuwanie rekordu z tabeli table\_name, który spełnia warunek condition.



Powyżej, następuje usunięcie rekordu o ID równym 4 z tabeli POSTS.

## Czym jest indeks i jak go tworzyć

Indeks w bazach danych to struktura ułatwiająca wyszukiwanie danych. Jest on tworzony za pomocą instrukcji CREATE INDEX. Wymaga własnej przestrzeni dyskowej, gdzie przechowuje kopię indeksowanych danych w tabeli. Oznacza to, że indeks to nic innego jak redundancja (nadmiarowość w celu zabezpieczenia na wypadek uszkodzenia systemu, w tym wypadku bazy danych). Tworzenie indeksu nie zmienia danych w tabeli tylko tworzy nową strukturę danych, która odnosi się do tabeli. Indeks bazy danych można porównać do indeksu na końcu książki: zajmuje własną przestrzeń, jest nadmiarowy i odnosi się do rzeczywistych informacji przechowywanych w innym miejscu.

Indeksowanie ułatwia wyszukiwanie danych, ponieważ uporządkowuje zestaw danych (porządek sortowania określa pozycję każdego rekordu w bazie), przez co samo wyszukiwanie jest dużo szybsze.

Indeks bazy danych jest dużo bardziej złożony niż drukowany katalog, ponieważ podlega on ciągłym zmianom. Aktualizacja drukowanego katalogu jest jedynie możliwa poprzez wydrukowanie całości od nowa. Baza danych SQL nie może czekać tak długo. Musi egzekwować wyrażenia typu insert, delete czy update praktycznie natychmiast, zachowując porządek indeksu bez potrzeby przenoszenia dużych ilości danych.

# Spring

Spring opiera się na pisaniu programów przez podejście Inversion os Control (IoC), i jego wzorca Dependency Injection (wstrzykiwanie zależności). Wstrzykiwanie polega na podawaniu obiektom ich zależności zamiast zmuszaniu ich by te zależności uzyskiwały.

IoC to pojęcie bardzo ogólne. Przedstawia koncepcję, gdzie jakiś osobny byt jest odpowiedzialny za tworzenie zależności, a nie jak do tej pory robił to programista.

DI jest jednym ze wzorców IoC. Obiekt nie tworzy obiektów, które są wykorzystywane wewnątrz. Dzięki temu program nie jest związany konkretną implementacją (gdzie najlepiej operować na interfejsach), a także nie trzeba znać parametrów konstruowanego obiektu.

Wstrzykiwanie może odbywać się poprzez wstrzykiwanie zależności do konstruktora lub poprzez deklarację tzw. Beanów.

## Bean

Bean to dowolny obiekt, którego cyklem życia zarządza Spring. Jeśli Spring utworzy obiekt w oparciu o stworzone klasy to ten obiekt automatycznie staje się beanem.

Beany są przechowywane w tzw. Kontenerze. Jest to miejsce, w którym Spring tworzy beany. Przy użyciu frameworka Spring nie tworzy się obiektów samodzielnie, zajmuje się tym właśnie Spring (przy tym właściwie ustawia zależności między obiektami – dokonuje wstrzykiwania zależności).

Deklarowanie komponentów na podstawi których Spring tworzy obiekty dokonuje się za pomocą adnotacji @Component przed nazwą klasy. Innym sposobem jest adnotacja @Bean przed metodą tworzącą obiekt.

### Rodzaje komponentów

* @Component – podstawowa adnotacja oznaczająca klasą mającą stać się Beanem.
* @Service – adnotacja dla klas bardziej złożonych
* @Controller – adnotacja używana do pisania kontrolerów dla aplikacji www
* @Repository – oznacza Beany odpowiedzialne za przechowywania danych w trwałych zbiorach danych

### @Configuration

Wskazuje, że klasa deklaruje jedną lub więcej metod tworzących Beany i może być przetwarzana przez kontener Spring w celu wygenerowania komponentu bean i żądań usług dla tych komponentów w czasie wykonywania.

### Scope

Zasięg beanów opisuje sposób tworzenia i widoczności beanów w poszczególnych miejscach programu. Zasięg opisuje się za pomocą adnotacji @Scope.

Zasięgi beanów:

* Singleton – Domyślny zasięg, oznacza., że w kontenerze tworzony jest tylko jeden bean danego typu.
* Prototype – zasięg pozwalający na utworzenie wielu instancji beanów danego typu w kontenerze.

### Automatyczna konfiguracja beanów

Adnotacja @Autowired – W sytuacji, gdy klasa posiada więcej niż jeden konstruktor potrzebne jest wskazanie którego z nich ma użyć do wstrzyknięcia beanów. Można tę użyć ją przy setterze, lub też przy polu w klasie.

## SpringBoot

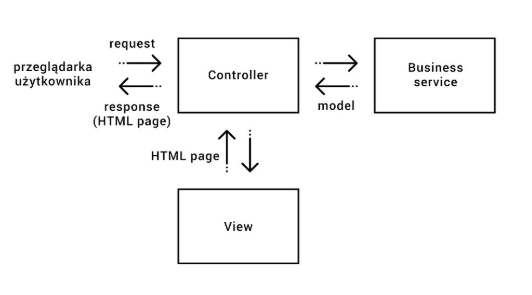
Jest to projekt, który powstał aby ułatwić start ze Springiem, eliminuje tworzenie konfiguracji w języku xml, a także gotową aplikację, można stworzyć i uruchomić za pomocą jednej klasy. Jedyne co należy zrobić to zbudować odpowiednie zależności

## Wzorzec projektowy MVC

MVC – Model – View – Conroller

Jest to frontendowy wzorzec projektowy w aplikacjach WWW. Działa on w następujący sposób:

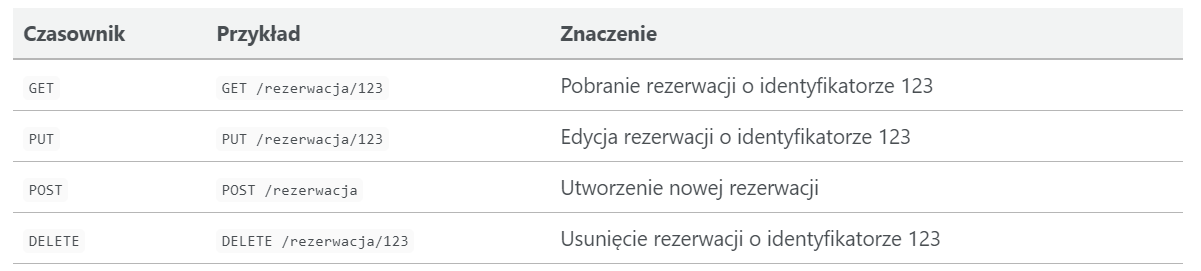
* Użytkownik za pośrednictwem przeglądarki wysyła żądanie do serwera WWW
* Kontroler „wie”, którą usługę biznesową pobrać z systemu i wywołuje metodę z danej usługi
* Usługa generuje dane (model) i przekazuje je do kontrolera
* Kontroler generuje metodę z odpowiedniego generatora widoku, przekazując mu dane z usługi
* Generator widoku tworzy stronę w HTML odpowiednią dla przeglądarki użytkownika i przesyła do kontrolera
* Kontroler odsyła użytkownikowi gotową stronę HTML z oczekiwanymi danymi



## REST API, zapytania HTTP

REST – Representational State Transfer – zbiór praktyk określający w jaki sposób powinny być implementowane usługi sieciowe. W aplikacjach typu REST obiekty to encje (ang. Resource) i reprezentują one obiekty w aplikacji. Operacje na encjach określane są przez zapytania HTTP :

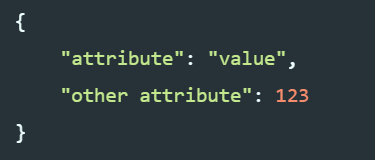
* GET – służy do pobierania danych. Tutaj wystarczy podać odpowiedni endpoint, ewentualnie zmodyfikować nagłówki(headers) zapytania.
* POST – służy tworzeniu i przesłaniu nowych danych. W tym przypadku konieczne jest już stworzenie ciała(body), w którym przekażemy dane do naszego REST API.
* PUT – również służy przesyłaniu danych, lecz najczęściej w celu aktualizacji tych danych. Tutaj również wymagane jest przesłanie danych w ciele
* DELETE – metoda służąca do usuwania danych. Jest to tak zwane soft-delete. Polega to na tym, że kasując dane za pomocą API tak naprawdę tylko dodawane są do encji informacje o tym, że została ona usunięta. W rezultacie dane pozostają nadal w bazie, lecz nie są one dostępne z poziomu API. Mechanizm ten należy już zaimplementować w samym API i nie ma on nic wspólnego z HTTP.



### JSON

JSON – Java Script Object Notation – format tekstowy zapisu i wymiany danych. W tym formacie występuje kilka symboli i słów kluczowych opisujących jego strukturę.

* Łańcuch znaków jest w cudzysłowie ” ”
* Obiekt reprezentowany jest w klamrach { }. Wewnątrz umieszczane są pary w postaci: ”nazwa atrybutu” : <wartość atrybutu>, a pary oddzielone są przecinkiem



* Lista wartości jest reprezentowana przez nawiasy [ ]



* Liczby w formacie JSON wyraża się ciągiem cyfr, kropka służy do oddzielania części całkowitej od dziesiętnej, a także dozwolony jest znak minus
* Literały występujące w tym formacie to true, false i null

### Walidacja danych

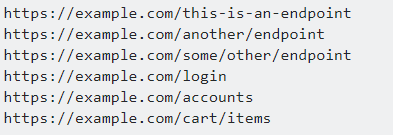
Walidacja to nic innego jak szereg reguł. Tylko poprawne dane spełniają te ograniczenia. Ograniczenia na dane nakładane są poprzez odpowiednie adnotacje.

Najczęściej używane reguły (adnotacje)

* @NotNull – dany element nie może mieć wartości null
* @NotBlank – dany element nie może mieć wartości null i musi zawierać co najmniej jeden znak nie będący spacją, tabulatorem itp.
* @NotEmpty – dany element nie może mieć wartości null i musi zawierać co najmniej jeden znak
* @Min(x) – dany element musi być liczbą i jego wartość musi być co najmniej x
* @Max(x) - dany element musi być liczbą i jego wartość musi być maksymalnie równa x
* @Email – dany element musi zawierać poprawny adres e-mail
* @Pattern(regexp=X) – dany element musi pasować do wyrażenia regularnego X
* @Size(min=X, max=Y) – dany element musi mieć rozmiar określony przez X i Y
* @Future – sprawdza czy podana data jest w przyszłości

### Endpointy

Endpoint jest to koniec kanału komunikacyjnego. Gdy API wchodzi w interakcję z innym systemem punkty kontaktowe tej komunikacji są uznawane za endpointy. Mówiąc inaczej w adresach http endpointy to końcówki adresów.



Gdzie <https://example.com/> - jest domeną natomiast to co jest po niej to adresy danych endpointów.

## Aspekty

Aspekty stanowią alternatywę dla dziedziczenia. Korzystając z technik programowania aspektowego (AOP), można zdefiniować popularne mechanizmy w jednym miejscu, lecz za pomocą deklaracji definiuje się, gdzie i w jaki sposób mechanizmy zostaną zastosowane, bez konieczności modyfikowania samej klasy, dla której zastosowanie mają właśnie te mechanizmy. Aspekt składa się z punktów przecięcia, punktów złączenia i porad.

### Porady

Główna czynność jaką ma wykonać aspekt nazywany jest poradą. Jest ona nawet nie tyle czynnością, która ma zostać wykonana, ale też i definicją właściwego momentu jej wykonania. Aspekty w springu mogą działać z pięcioma rodzajami porad:

* Before – Funkcjonalność porady wykonywana jest przed wywołaniem metody z poradą
* After – Funkcjonalność porady jest wykonywana po zakończeniu działania metody z poradą, niezależnie od wyniku jej działania
* After-returning – Funkcjonalność porady jest wykonywana po prawidłowym zakończeniu metody z poradą
* After-throwing – Funkcjonalność porady jest wykonywana po zgłoszeniu wyjątku przez metodę z poradą
* Around – Funkcjonalność porady jest wywoływana zarówno przed wywołaniem jak i po zakończeniu metody z poradą (Before i After w jednym)

### Punkty złączenia

Jest to takie miejsce w przebiegu wykonywania programu, gdzie może zostać wpięty aspekt. Może to być np.: wywołanie metody zgłoszenie wyjątku, a nawet modyfikacja pola. Czyli jest to pozycja, w których kod aspektu może zostać włączony w normalny przebieg działania aplikacji wzbogacając jej działanie.

### Punkty przecięcia

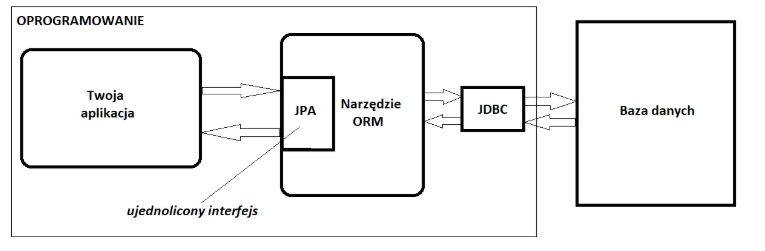
Definiuje miejsce, w którym porada ma być wykonana. Są one często jawnie określane przez wskazanie nazw metod i klas, lub za pomocą wyrażeń regularnych, dopasowujących te nazwy.

# Hibernate

## ORM – Mapowanie obiektowo-relacyjne

ORM jest to koncept (proces) konwertowania danych z języka obiektowego na relacyjny język baz danych i odwrotnie. W Javie odbywa się to za pomocą JPA.

JPA – Java Persistance API – ujednolicony standard zbudowany na potrzeby języka Java ułatwiający korzystanie z narzędzi ORM. Pozwala on na operacje na obiektach (encjach) i zapisywania wyników tych operacji do baz danych, łączących się z bazą danych przez protokół JDBC.



## Różnica między Hibernate a JPA

JPA jest tylko specyfikacją, co oznacza, że nie ma implementacji. Można dodawać adnotacje JPA, lecz bez implementacji nic się nie stanie. Można stwierdzić, że JPA to tylko wytyczne, które muszą być przestrzegane, kiedy to implementacja JPA Hibernate to kod, który spełnia warunki dla API zdefiniowanego w JPA, a także spełnia jego funkcjonalność. Inaczej rzecz ujmując JPA jest to pewien standard, a narzędziem go implementującym jest Hibernate.

## Encje

Hibernate musi wiedzieć o tym , że obiekty danej klasy muszą być przechowywane w bazie danych. Dlatego klasę, którą Hibernate ma umieścić w bazie danych należy oznaczyć adnotacją @Entity (encja to inaczej tabela).

### Mapowanie na tabele, kolumny

Aby hibernate wiedział jaką nazwę ma mieć dana tabela należy po adnotacji @Entity dodać adnotację @Table(name =) i poprzez argument name podać nazwę, która ma zostać użyta jako nazwa tabeli.

Nazwy pól z reguły są nadawane przez Hibernate według jego strategii. Aby wymusić nazwę pól, należy skorzystać z adnotacji @Column(name = ” ”), gdzie do atrybutu name przypisana zostaje nazwa. Adnotację tą można umieścić bezpośrednio nad deklaracją pola lub nad metodą gettera odczytującego z pola.

Pole o nazwie ID będące z reguły kluczem głównym tabeli należy oznaczyć adnotacją @Id, a także by hibernate sam przypisywał wartości do tego pola, należy oznaczyć je adnotacją @GeneratedValue.

### Relacje między encjami

Encje, które w żaden sposób nie są ze sobą powiązane nazywa się płaskimi encjami. Występują 3 rodzaje relacji: Jeden do jednego (1:1), jeden do wielu (1:N) oraz wiele do wielu (N:N).

* Relacja jeden do jeden – sytuacja, gdy każdemu rekordowi z jednej tabeli odpowiada dokładnie jeden rekord z drugiej tabeli. W hibernate tworzy się ją za pomocą adnotacji @OneToOne.
* Relacja jeden do wielu (wiele do jednego) – najczęściej używany typ połączenia. Występuje wtedy, gdy pojedynczemu rekordowi z jednej tabeli przyporządkowane jest wiele rekordów z drugiej tabeli, lecz temu rekordowi z tabeli drugiej przyporządkowany jest dokładnie jeden rekord z tabeli pierwszej. W hibernate tworzy się ją za pomocą adnotacji @OneToMany i @ManyToOne.
* Relacja wiele do wielu – występuje gdy rekordom z tabel przyporządkowane jest wiele rekordów z innych tabel. W hibernate tworzy się ją za pomocą adnotacji @ManyToMany.

Adnotacje określające relacje między encjami mogą posiadać kilka parametrów:

* Cascade – określa co ma się dziać z rekordami w tabeli zależnej w momencie, gdy zapisywany lub usuwany jest rekord w tabeli uzależniającej (różne funkcje np. CascadeType.REMOVE, CascadeType.PERSIST, CascadeType.ALL).
* Fetch – określa czy podczas wczytywania danej encji, z bazy danych mają być automatycznie odczytywane wszystkie rekordy z tabeli zależnych.
  + FetchType.EAGER – ładuje natychmiastowo dane z tabel zależnych wraz załadowaniem danej encji
  + FetchType.LAZY – ładuje dane z encji zależnej wtedy, gdy użytkownik próbuje się do nich dostać

W relacjach 1:1 można użyć FetchType.EAGER, gdyż podciągany jest tylko jeden rekord, leczy przy relacjach do wielu, lepiej się zastanowić, gdyż powiązanych rekordów może być bardzo dużo.

* TargetEntity – wskazuje na encję, która jest prawą stroną relacji 1:N, lub N:N. Wartością tego argumentu powinien być deskryptor klasy encji
* MappedBy – określa pole encji z prawej strony relacji, która przechowuje referencję do obiektu encji. MappedBY sygnalizuje Hibernate’owi, że relacja jest obustronna.

## HQL

Hibernate używa języka HQL podobnego do języka SQL, lecz w porównaniu z SQLem, HQL jest w pełni zorientowany obiektowo i rozumie pojęcia takie jak dziedziczenie i polimorfizm.

### NamedQuery czyli zapytania HQL

Adnotacja @NamedQuery przed klasą pozwala na użycie zapytań z języka SQL w Javie. Parametr name tej adnotacji określa nazwę klasy, na podstawie której tworzona jest encja, a także nazwę zapytania, czyli nazwę przypominającą nazwę metody.

Z kolej parametr query to trść zapytania w języku HQL

Istnieje również adnotacja @NamedNativeQuery pozwalająca skorzystać bezpośrednio z języka SQL.

## API Criteria

API Criteria to zdefiniowane wcześniej API używane to tworzenia zapytań (query) dla encji. Zapytania te są bezpieczne pod względem typu oraz przenośne i łatwe do modyfikowania poprzez zmianę składni. Główną zaletą API Criteria jest możliwość wykrycia błędów wcześniej podczas kompilacji.

## @Transactional

Transakcja to zbiór operacji zmieniających stan systemu w kilku krokach i tylko wykonanie wszystkich kroków jest uznawane za sukces. Jeśli, któryś z kroków się nie powiedzie, wszystkie zmiany, które nastąpiły w ramach transakcji zostają wycofane. W springu transakcji można dokonywać np. za pomocą adnotacji @Transactional

* Propagation - Definiuje jak transakcje oddziałują na siebie:
* Required – kod zawsze będzie się wykonywał w transakcji, utworzy nową lub powtórzy dostępną
* Requires\_new – kod zawsze wykona się w nowej transakcji i zawiesi obecną jeśli takowa istniej
* Isolation - Definiuje kontrakt wymiany danych pomiędzy transakcjami.

### Optymistic and Pesymistic Locking

Optymistic Locking jest to sytuacja, gdy użytkownik czyta rekord zwraca uwagę na jego właściwości (np. numer wersji) i sprawdza, czy wersja nie zmieniła się przed ponownym zapisaniem rekordu. Po zapisaniu należy przefiltrować aktualizację wersji, by upewnić się czy nie była aktualizowana między chwilą sprawdzenia wersji i zapisania rekordu na dysku (sprawdza czy wersja jest atomowa). Jeśli dany rekord jest nieczysty (dirty) należy odrzucić transakcję i ponownie ją wykonać.

Pesymistic Locking jest to sytuacja występująca, kiedy rekord jest wyłączony z użytku przez innych użytkowników (tylko jeden użytkownik na nim może pracować, póki nie zakończy transakcji). W tym przypadku jest zdecydowanie większa spójność danych niż w przypadku optymistic locking. Aby użyć tej opcji należy mieć bezpośredni dostęp do bazy danych lub zewnętrznie dostępne ID transakcji, które może być użyte niezależnie od połączenia.

## Cash pierwszego i drugiego poziomu

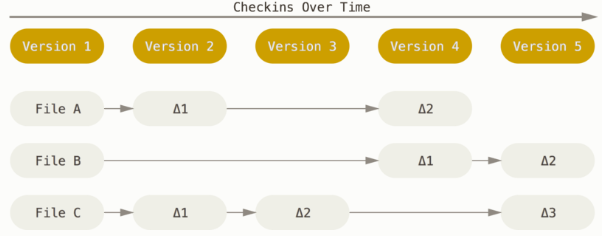
# Git

Git jest to jeden z systemów kontroli wersji. System kontroli wersji śledzi wszystkie zmiany dokonywane na pliku (lub plikach) i umożliwia przywołanie dowolnej wcześniejszej wersji. Używanie systemu kontroli wersji (VCS-Version Control System) jest bardzo rozsądnym rozwiązaniem. Pozwala on przywrócić plik(i) do wcześniejszej wersji, odtworzyć stan całego projektu, porównać wprowadzone zmiany, dowiedzieć się kto jako ostatnio zmodyfikował część projektu powodującą problemy, kto i kiedy wprowadził daną modyfikację. Oprócz tego używanie VCS oznacza, że nawet jeśli popełnisz błąd lub stracisz część danych, naprawa i odzyskanie ich powinno być łatwe.

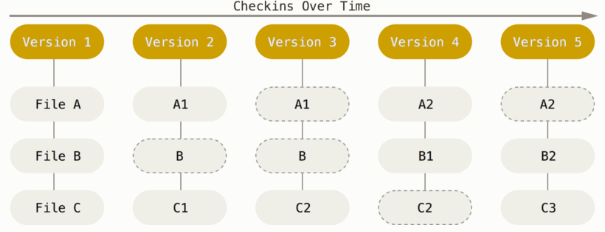
W rozproszonym systemie kontroli wersji (DVCS - Distributed Version Control System) takim jak Git, gdy jeden z serwerów, używanych przez te systemy do współpracy, ulegnie awarii, repozytorium każdego klienta może zostać po prostu skopiowane na ten serwer w celu przywrócenia go do pracy.

## Git jako „migawka” projektu

Systemy VCS głównie przechowuję informację jako listę zmian na plikach, traktują przechowywane informacje jako zbiór plików i zmian dokonywanych w danym okresie czasu.

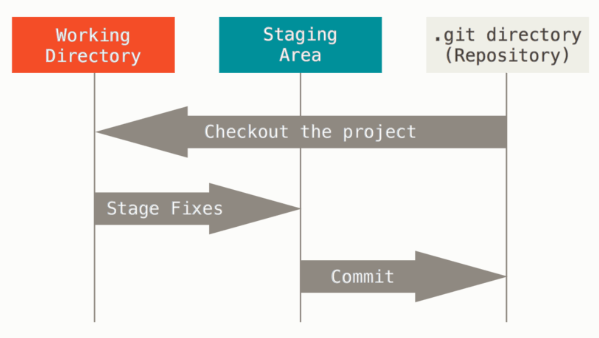


Git traktuje dane jako zestaw migawek (ang. Snapshots) małego systemu plików. Za każdym razem gdy zapisywany jest stan projektu, Git tworzy obraz przedstawiający to jak wyglądają wszystkie pliki w danym momencie i przechowuje referencje do tejże migawki.



Podstawowy sposób pracy z Git (wraz z grafiką):

* Modyfikacja plików w katalogu roboczym (ang. Working directory)
* Oznaczenie zmienionych plików jako plików śledzonych, dodając ich bieżący stan (migawkę) do przechowalni (ang. Staging area)
* Zatwierdzenie (COMMIT), podczas którego zawartość plików z przechowalni zapisywania jest jako migawka projektu w katalogu git



## Branch, merge, cherry-pick

Branch jak sama nazwa wskazuje jest to odgałęzienie z reguły od głównego projektu, na której użytkownik może dokonywać modyfikacji bez wprowadzania bałagany do gałęzi głównej. Gałąź w Gicie jest po prostu lekkim, przesuwalnym wskaźnikiem na któryś z owych zestawów zmian. Domyślna nazwa gałęzi Gita to master.

Merge jest operacją łączenia branchy. Za pomocą komendy ***git merge*** można łączyć poszczególne gałęzie projektu w jedno. W zależności od zmian wykonanych w plikach w trakcie komendy ***git merge*** może pojawić się konflikt o czym poinformuje komunikat w terminalu. Konflikt oznacza to, że git nie może połączyć dwóch wersji kodu. Po usunięciu wszystkich konfliktów i przebudowaniu projektu należy utworzyć commiti wypchnąć zmiany.

Podczas pracy na kilku branchach, gdy użytkownik chce skopiować tylko jeden commit z jednego do drugiego brancha, należy uzyć polecenia ***cherry-pick.*** Aby to zrobić należy:

* Upewnić się, czy oba branche dostępne są lokalnie
* Zrobić checkout na branch, z którego chcemy pobrać commit
* Za pomocą funkcji ***git log*** lub ***git tree*** skopiować hash commita, który ma być skopiowany
* Zrobić checkout na docelowy branch
* Zrobić cherry-pick za pomocą polecenia  ***git cherry-pick hash\_commita***
* Sprawdzić za pomocą ***git log*** lub ***git tree*** czy zmiany zostały wprowadzone