**Pytania rekrutacyjne**

**Java**

**1.Co to jest JVM?**

maszyna wirtualna oraz środowisko zdolne do wykonywania [kodu bajtowego Javy](https://pl.wikipedia.org/wiki/Kod_bajtowy_Javy).

Wirtualna maszyna Javy to zestaw aplikacji napisanych na tradycyjne urządzenia i [systemy operacyjne](https://pl.wikipedia.org/wiki/System_operacyjny). Dostarcza środowiska, w którym może się wykonywać program skompilowany do postaci [kodu bajtowego Javy](https://pl.wikipedia.org/wiki/Kod_bajtowy_Javy), zapewniając takie usługi, jak [odśmiecanie pamięci](https://pl.wikipedia.org/wiki/Od%C5%9Bmiecanie_pami%C4%99ci" \o "Odśmiecanie pamięci) czy obsługę [wyjątków](https://pl.wikipedia.org/wiki/Wyj%C4%85tek) oraz [bibliotekę standardową](https://pl.wikipedia.org/wiki/Biblioteka_programistyczna). W zależności od potrzeb i liczby dostępnych narzędzi, wyróżniane są dwie główne dystrybucje: JRE i JDK

Elementy składowe:

W skład maszyny wirtualnej Java wchodzą następujące elementy:

* [**Interpreter**](https://pl.wikipedia.org/wiki/Interpreter_(program_komputerowy)) – wykonuje krok po kroku instrukcje programu zapisane w postaci kodu bajtowego,
* [**kompilator**](https://pl.wikipedia.org/wiki/Kompilator)[**JIT**](https://pl.wikipedia.org/wiki/JIT_(informatyka)) – opcjonalny komponent wchodzący w skład części implementacji, który kompiluje najczęściej wykonywane fragmenty kodu do postaci [kodu maszynowego](https://pl.wikipedia.org/wiki/J%C4%99zyk_maszynowy), dzięki czemu mogą być one wykonywane bezpośrednio przez procesor komputera. Pozwala na zwiększenie wydajności,
* **Zarządca pamięci** – zarządza [stertą](https://pl.wikipedia.org/wiki/Sterta_(informatyka)), na której przechowywane wszystkie [obiekty](https://pl.wikipedia.org/wiki/Obiekt_(programowanie_obiektowe)) wykonywanej aplikacji oraz zapewnia automatyczne zwalnianie nieużywanej pamięci,
* **Weryfikator kodu bajtowego** – kluczowym dla bezpieczeństwa aspektem jest weryfikacja kodu bajtowego przed jego uruchomieniem, której celem jest sprawdzenie poprawności wszystkich odwołań oraz upewnienie się, że wykonanie danego fragmentu nie zaszkodzi stabilności lub bezpieczeństwu systemu, na którym uruchamiana jest maszyna wirtualna. Zajmuje się tym weryfikator kodu bajtowego.
* **Java API** – zestaw bibliotek programistycznych udostępniających takie usługi, jak obsługę plików czy [GUI](https://pl.wikipedia.org/wiki/Graficzny_interfejs_u%C5%BCytkownika), z których korzystają wykonywane aplikacje. Większość biblioteki standardowej napisana jest w języku Java, dlatego maszyny wirtualne nie muszą dostarczać własnej implementacji.

Na JVM działają : Grooove, Scala, Kotlin, JRuby

**2. Wzorce projektowe.**

Sposoby pisania programu, aby był on przejrzysty i bardziej zrozumiały.

Nie są to gotowe rozwiązania. Trzeba je zaadaptować do swoich programów.

**a)Adapter Pattern**

Czyli gdy mamy dwa interfejsy : old i new i chodzi o ty by zapewnić to, by osoby korzystające ze starszego interfejsy mogły nadal to robić w sposób przezroczysty mimo, że już powstał nowy interfejs.

**public class** NoteAdapter **implements** OldInterface {  
  
 **public** List<String> **noteList**;  
 **public** NewNote **newNote**;  
  
  
 **public** NoteAdapter(){  
 **noteList** = **new** ArrayList<>();  
 **newNote** = **new** NewNote();  
 }  
  
 @Override  
 **public void** addNote(String note) {  
 **noteList**.add(note);  
 **newNote**.addNote(note, **null**);  
 }

**public void** addNote(String note, String toWho){  
 **noteList**.add(note+**" "**+toWho);  
 **newNote**.addNote(note, toWho);  
}

**main**

NoteAdapter note = **new** NoteAdapter();  
note.addNote(**"Dziś jest piękny dzień, zajmujemy się wzorcami :)"**);  
note.addNote(**"Odebrać dzieciaka z przedszkola!"**);

**b) Builder Pattern**

To sposób na rozwiązanie zbyt długiego wywołania konstruktora podczas tworzenie obiektu.

Przykładowo mamy klase Cat i wewnątrz niej klasę statyczną

*// Klasa wewnętrzna statyczna Builder***public static class** Builder{  
 String **name**;  
 String **sex**;  
 **int age**;  
 **int weight**;  
  
 **public** Builder() {  
 }  
  
  
 **public** Builder name(String name) {  
 **this**.**name** = name;  
 **return this**;  
 }  
  
 **public** Builder sex(String sex){  
 **this**.**sex**=sex;  
 **return this**;  
 }  
  
 **public** Builder age(**int** age){  
 **this**.**age**=age;  
 **return this**;  
 }  
  
 **public** Builder weight(**int** weight){  
 **this**.**weight**=weight;  
 **return this**;  
 }  
  
 **public** Cat build(){  
 **return new** Cat(**this**);  
 }  
}  
*// koniec klasy Builder*

main

i uruchamiamy np. tak

Cat cat=Cat.*builder*()  
 .name(**"Kropka"**)  
 .age(15)  
 .sex(**"plec"**)  
 .weight(12)  
 .build();  
System.***out***.println(cat);

**c) Singleton**

Kiedy potrzebujemy tylko jednej instancji obiektu, np. połączenie mysql.

**public class** MySqlConnector {  
 **private static** MySqlConnector *instance*;  
 **private** MySqlConnector(){  
  
 }  
 **public static** MySqlConnector getInstance(){  
 **if** (*instance*==**null**){  
 *instance*=**new** MySqlConnector();  
 }  
 **return** *instance*;  
 }  
}

**3. Garbage Collector**

Komponent JVM odpowiedzialny za czyszczenie pamięci.

**4.Klasa String**

Obiekt typu immutable.

**5.Wątek i proces**

Wątek jest podzbiorem procesu. Jeden proces może zawierać wiele wątków.

**6.Na czym polega debugowanie?**

Usuwanie błędów. Wstrzymane działania programu aby odczytać wartości zmiennych.

**7. Jakie są rodzaje kolekcji?**

List, Set, Queue

**8. Co to jest TDD**

Test Driven Development (metodyki zwinne, kiedyś programowanie ekstremalne)

1. Najpierw programista pisze automatyczny test sprawdzający dodawaną funkcjonalność. Test w tym momencie nie powinien się udać.
2. Później następuje implementacja funkcjonalności. W tym momencie wcześniej napisany test powinien się udać.
3. W ostatnim kroku programista dokonuje [refaktoryzacji](https://pl.wikipedia.org/wiki/Refaktoryzacja" \o "Refaktoryzacja) napisanego kodu, żeby spełniał on oczekiwane standardy.

**9. Final, finally, finalize**

final zmienna finalna

finally wyjątki

finalize metoda

**10.Co to jest czysty kod**

Napisane testy, docs, wzorce projektowe, zgodne z SOLID

**11.Profiler**

Program używany do monitorowania aplikacji.

Szybkość akcji, wykorzystanie pamięci

Np. JProfiler

**Spring**

**1.Co to jest IoC (inversion of control)**

Odwrócenie sterowanie. Jest to zmiana odpowiedzialności pewnej części aplikacji

„Nie dzwoń do nas, my zadzwonimy do Ciebie”

Zewnętrzne biblioteki, frameworki wywołują w odpowiednich momentach kod programisty.

**2. Co to jest WebService**

Usługa zewnętrzna z której mogą korzystać programiści

REST i SOAP

**3.Korzyści używania Springa**

Lekki framework.

Liczy około kilkadziesiąt MB.

Pozwala na korzystanie z IoC.

Pozwala programować aspektowo

**Programowanie aspektowe** (aspect-oriented programming, AOP) to paradygmat tworzenia programów komputerowych wspomagający separację zagadnień i rozdzielenie programu na części w jak największym stopniu niezwiązane funkcjonalnie.

Zarządza cyklem życia aplikacji.

Pozwala na korzystanie z MVC . Struts przeżytek

**4.Co to jest EJB**

Działa po stronie serwera. Technologia umozliwia pisanie cześć backend aplikacji Umożliwia transakcyjność, trwałość (persistent) aplikacji, rozproszenia, bezpieczeństwa.

**SQL**

**1.Co to jest JDBC ?**

– łącze do bazy danych opracowane przez firmę Sun. Umożliwia komunikację klasom Javy z bazą danych przy użyciu języka SQL.

**2. LEFT OUTER JOIN**

Złączenie lewostronne zewnętrzne

**Hibernate**

**1.Co to jest ORM?**

ORM – Object Relational Mapping, mapowanie obiektowo-relacyjne.

Pozwalają na odzwierciedlenie obiektów na tabele w relacyjnej bazie danych. Komunikujemy się obiektowo, a nie przez sql. Najpopularniejszy używany w Javie ORM to Hibernate.

**Zasady**

**1.Co to jest SOLID?**

* **Zasada pojedynczej odpowiedzialności** (ang. ***Single-Responsibility Principle*** *– SRP*),

**Jedna klasa to jedna odpowiedzialność.**

Żadna klasa nie może być modyfikowana z więcej niż jednego powodu”.

* **Zasada otwarte – zamknięte** (ang. *Open/Closed Principle – OCP*),

„Składniki oprogramowania (klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji”.

* 1. **„otwarty na rozbudowę”?** Mianowicie to, że musi istnieć stosunkowo prosty sposób rozbudowy zachowań takiego modułu.
  2. **„Zamknięte dla modyfikacji”** oznacza z kolei, że rozbudowa modułu nie może być przeprowadzona w sposób, który spowoduje zmianę istniejącego kodu źródłowego.\
* **Zasada podstawiania Liskov** (ang. *Liskov Substitution Principle – LSP*),

„Musi istnieć możliwość zastępowania typów bazowych ich podtypami”***.***

Załóżmy, że posiadamy dwie klasy, w których znajdujemy wiele wspólnych składowych. Wspólnymi składowymi mogą być: zbiór właściwości, metod itd. Co w takiej sytuacji należałoby zrobić? Zgodnie z zasadą podstawiania Liskov, wypadałoby wyodrębnić wspólne elementy obu klas do jednej klasy – najlepiej abstrakcyjnej.

* **Zasada segregacji interfejsów** (ang. *Interface Segregation Principle – ISP*),

Zasada segregacji interfejsów ma za zadanie przede wszystkich wyeliminowanie nieporęcznych, niepotrzebnie rozbudowanych interfejsów.

Załóżmy, że posiadamy pewien interfejs oraz dwie klasy, które go implementują. Klasa Aimplementuje nasz przykładowy interfejs w całości – wszystkie funkcje, natomiast klasa B posiada pełną implementację tylko części z nich, pozostałe, które są niepotrzebne pozostają puste lub zwracają domyśle wartości. Jeśli mamy do czynienia z taką sytuacją, powinniśmy rozważyć rozdzielenie interfejsu na mniejsze tak, aby każdy z nich deklarował tylko te funkcje, które rzeczywiście są wywoływane przez danego klienta lub grupę klientów (klasa/grupy klas implementujące dany interfejs).

* **Zasada odwracania zależności** (ang. *Dependency Inversion Principle – DIP*).

1. *Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji.*
2. *Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.*

1)Moduły wysokiego poziomu z natury rzeczy zawierają ważne decyzje strategiczne i modele biznesowe danej aplikacji. Myślę, że wszyscy się ze mną zgodzą, iż te właśnie moduły w największym stopniu odpowiadają za funkcjonowanie aplikacji. Gdyby okazało się, iż zależą one od modułów niskiego poziomu, to zmiany elementów niskiego poziomu mogłyby mieć wpływ na funkcjonowanie modułów wysokopoziomowych, a co za tym idzie wymuszałyby zmiany na wyższych poziomach. Dodatkowo, gdy moduły wysokopoziomowe zależą od niskopoziomowych, ponowne ich wykorzystanie staje się niezwykle trudne. Jeśli jednak odwrócimy tę zależność w drugą stronę, to bardzo łatwo będzie można wielokrotnie je wykorzystywać.

2)Jeśli chodzi o zależność od abstrakcji, można to zamknąć w jednej prostej formule: *pisany przez nas kod nie powinien być uzależniony od konkretnej klasy, zależności takie powinny kończyć się na klasach abstrakcyjnych bądź interfejsach*. Zasadę zależności od abstrakcji (ang. *depend on abstractions)* można również streścić w trzech prostych punktach:

* Żadna zmienna nie powinna zawierać referencji do konkretnej klasy.
* Żadna klasa nie powinna dziedziczyć po konkretnej klasie.
* Żadna metoda nie powinna przykrywać metody zaimplementowanej w którejkolwiek z klas bazowych.

Narzędzia

**1.Różnice SVN i git**

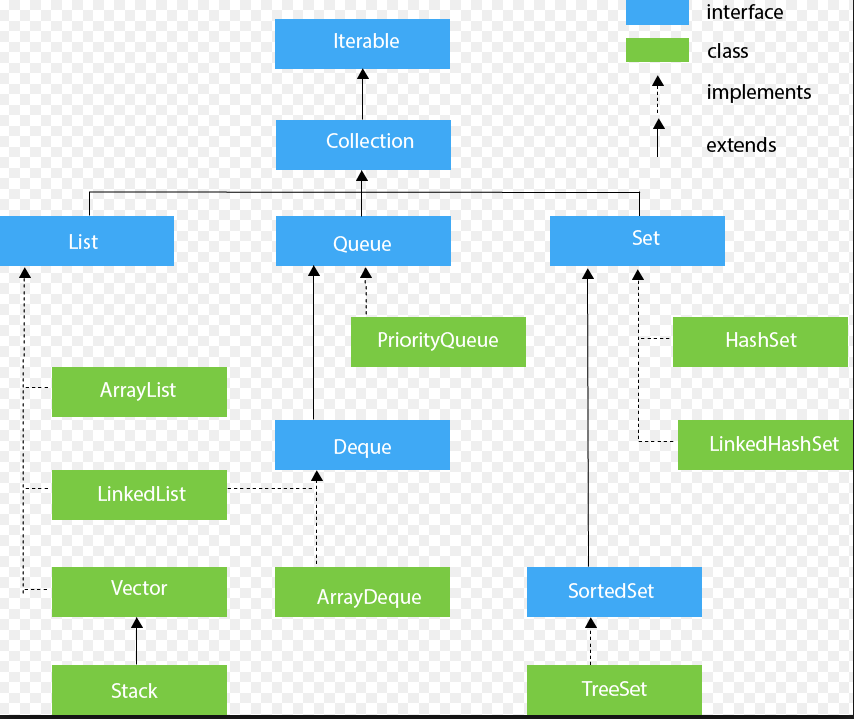
Kontrola wersji

A różnice

**Git:** szybszy, repo są mniejsze, bo trzymają wskaźniki do blobów a nie kopiowanie, system rozproszony

SVN – istnieje jedno centralne repo

**Kolekcje**



**Wyrażenia regularne**

Pattern.*matches*(**".{3}[0-9]{3}"**,**"abc345"**)); *//Czy pierwsze trzy znaki są dowolne, zaś następne 3 znaki tylko cyframi?*Pattern.*matches*(**"[0-9]+"**,**"545"**)); *//Czy string zawiera tylko cyfry?*Pattern.*matches*(**"[a-zA-z]+"**,**"a"**)); *//Czy string zawiera tylko litery?*Pattern.*matches*(**"\\w+"**,**"3443dfdf"**));*//Czy string zawiera tylko cyfry i litery?*Pattern.*matches*(**"[0-9]{5}"**,**"12345"**));*//Czy string zawiera tylko cyfry i ma dokładnie 5 znaków?*Pattern.*matches*(**"J.\*"**,**"J"**));*//Czy string rozpoczyna się od "J"?*Pattern.*matches*(**"J.\*U"**,**"J5UUU"**));*//Czy string rozpoczyna się od "J" i kończy na "U"?*Pattern.*matches*(**"[A-Z].\*"**,**"Ala"**));*//Czy string rozpoczyna się od wielkiej litery?*Pattern.*matches*(**".\*\\p{Space}.\*"**,**"g g"**));*//Czy string zawiera spację? (biały znak)?*Pattern.*matches*(**"[7-9][0-9]{8}"**,**"777777777"**));*//Czy string jest numerem telefonu zaczynającym się od 7 lub 8 lub 9?*Pattern.*matches*(**"[a-zA-Z]{2}([0-9]{4}){4}"**,**"ab1234123412341234"**));*//Czy string jest numerem bankowym? (Zakładając, że zaczyna się od dwóch liter, następnie 4 grupy po 4 cyfry)*Pattern.*matches*(**"[0-9]{2}-[0-9]{3}"**,**"20-533"**));*//Czy string jest kodem pocztowym?*