PODSTAWY JAVY

1. Typy proste, obiekty, referencje na obiekty – analiza porównawcza Javy z C++

Typy danych prymitywnych, to typy proste. Są wstępnie zdefiniowanymi typami danych Java. Określają rozmiar i typ dowolnych standardowych wartości. Java ma 8 prymitywnych typów danych, a mianowicie: bytes, short, int, long, float, double, char, boolean.

Kiedy prymitywny typ danych jest przechowywany, to wtedy ten typ jest stosem, do którego mogą być przypisane wartości. Gdy jakaś zmienna zostanie skopiowana, jest wtedy tworzona inna kopia zmiennej, a zmiany wprowadzone do skopiowanej zmiennej nie będą odzwierciedlać zmian w oryginalnej.

Typy:

Integer: Ta grupa obejmuje byte, short, int, long

byte - jest to jednobajtowy (8-bitowy) typ danych. Zakres wartości jest od –128 do 127, byte b=10

Short – jest dwubajtowy. Zakres od –32768 do 32767. Domyślna wartość 0. Przykład short s=11;

Int – czterobajtowy (32bity). Spory zakres

Long – jest ośmiobajtowy(64bity). Spory zakres

Floating-Point Number: ta grupa obejmuje float, double:

Float jest czterobajtowy (32-bits), a double jest ośmiobajtowy(64-bits)

Characters – ta grupa reprezentuje znaki char, które reprezentują symbole w zestawie znaków; litery, cyfry.

Char – 2 bajtowy (16-bitów) to unsigned unicode znak od 0 do 65.535.

Boolean - używany gdy chcemy przetestować konkretny warunek podczas wykonywania programu. Istnieją dwie wartości, które typ boolowski może przyjąć: prawda lub fałsz. Oba te słowa są słowami kluczowymi. Typ boolowski jest oznaczony słowem Boolean i wykorzystuje tylko 1 bit pamięci.

Object Data Type – Typ danych obiektów
Są one nazywane również typami danych nieprymitywnych lub referencyjnych. Są tak nazywane, ponieważ odnoszą się do konkretnego, dowolnego obiektu. W przeciwieństwie do typów danych prymitywnych, typy nieprymitywne są tworzone przez użytkownikó w Javie. Przykłady te obejmują tablice, ciągi znaków, klasy, interfejsy, itp.

Gdy zmienne referencyjne (odwołania do obiektu) – zostaną zapisane, zmienna zostanie zapisana w stosie, a oryginalny obiekt zostanie zapisany w stercie.

W typie danych obiektu, chociaż zostaną utworzone dwie kopie, obie będą wskazywać na tę samą zmienną w stercie, stąd zmiany wprowadzone do dowolnej zmiennej będą odzwierciedlać zmianę w obu zmiennych.

Referencje na Obiekty
Referencje to zmienne, które wskazują na lokalizację obiektu w pamięci (w stercie).

- Przechowywanie: Przechowywane są na stosie, ale zawierają adresy obiektów przechowywanych w stercie.

- **Zachowanie**: Operacje na referencjach wpływają na obiekty, na które wskazują. Dwie różne referencje mogą wskazywać na ten sam obiekt, co oznacza, że zmiany dokonane przez jedną referencję będą widoczne dla drugiej.
- **Null**: Referencje mogą być null, co oznacza, że nie wskazują na żaden obiekt. Próba użycia referencji o wartości null spowoduje błąd (NullPointerException w języku Java).

Właściwości	Typy pierwotne	Obiekty
Pochodzenie	Predefiniowane typy danych	Typy danych zdefiniowane przez użytkownika
Struktura przechowywania	Przechowywane na stosie	Zmienna referencyjna przechowywana na stosie, a oryginalny obiekt przechowywany w stercie
Podczas kopiowania	Tworzone są dwie różne zmienne z różnymi przypisaniami (tylko wartości są takie same)	Tworzone są dwie zmienne referencyjne, ale obie wskazują na ten sam obiekt w stercie
Kiedy zmiany są wprowadzane w skopiowanej zmiennej	Zmiany nie są odzwierciedlane w oryginalnych zmiennych	Zmiany są odzwierciedlane w oryginalnych zmiennych
Wartość domyślna	Typy pierwotne nie mają wartości null jako wartości domyślnej	Wartością domyślną dla zmiennej referencyjnej jest null
Przykład	byte, short, int, long, float, double, char, boolean	array, klasa String, interfejs itp.

Java a C++

- Język Java oparty jest na C++, eliminuje jego wady, dlatego:
  - nie wykorzystuje wskaźników
  - brak w nim złożonych konwersji
  - brak przeciążania operatorów
  - brak dziedziczenia wielokrotnego
  - inna koncepcja zarządzania tablicami
  - używanie kodowania znaków Unicode zamiast ASCII
  - wielowątkowość
  - automatyczne odzyskiwanie pamięci
  - programowanie tylko obiektowe

2.Czym jest klonowanie i kopiowanie obiektu?

W Javie klonowanie i kopiowanie obiektu odnoszą się do różnych metod tworzenia nowej instancji obiektu na podstawie istniejącego. Oto szczegółowe omówienie tych koncepcji:

Definicje:

- **Klonowanie**: Tworzenie dokładnej kopii obiektu przy użyciu interfejsu Cloneable i metody clone().
- **Kopiowanie**: Ręczne tworzenie nowego obiektu na podstawie istniejącego, może być płytkie lub głębokie.

Klonowanie Obiektu:

Interfejs Cloneable:

- Wskaźnik dla mechanizmu klonowania w Javie.

- Informuje, że metoda clone() może być bezpiecznie użyta na obiektach tej klasy.

Metoda clone():

- Chroniona (protected) w klasie Object.
- Musi być nadpisana i zazwyczaj ustawiona jako publiczna w klasie implementującej Cloneable

Kopiowanie Obiektu:

Płytkie kopiowanie (shallow copy):

- Tworzy nowy obiekt.
- Kopiuje wartości płytkie (referencje) dla pól obiektowych.
- Wewnętrzne obiekty pozostają te same.

Głębokie kopiowanie (deep copy):

- Tworzy nowy obiekt.
- Kopiuje wszystkie wewnętrzne obiekty, tworząc oddzielne instancje.

3. Omów składnie i działanie konstruktora. Przedstaw przykład wywołania w ramach konstruktora, konstruktora tej samej klasy oraz klasy nadrzędnej:

Konstruktor w Javie to specjalna metoda, która jest wywoływana w momencie tworzenia obiektu klasy. Jego głównym zadaniem jest inicjalizacja nowo utworzonego obiektu. Konstruktor ma tę samą nazwę co klasa i nie ma zwracanego typu, nawet void.

```
class ClassName {
    // Konstruktor domyślny (bez argumentów)
    public ClassName() {
        // Kod inicjalizacyjny
    }

    // Konstruktor z argumentami
    public ClassName(Type arg1, Type arg2) {
        // Kod inicjalizacyjny z użyciem arg1 i arg2
    }
}
```

Bicycle ma jeden konstruktor:

public Bicycle(int startCadence, int startSpeed, int startGear) {
gear = startGear;
cadence = startCadence;
speed = startSpeed;
}

Aby utworzyć nowy obiekt Bicycle o nazwie myBike, konstruktor jest wywoływany przez operatora new:

Bicycle myBike = new Bicycle(30, 0, 8);
New Bicycle(30,0,8) tworzy przestrzeń w pamięci dla obiektu i inicjuje jego pola.

Bicycle ma jeden konstruktor ale może mieć ich więcej... w tym konstruktor bezargumentowy :

Utworzenie konstruktora: public Bicycle(){gear=1; cadence = 10; speed = 0; }

Wywołanie: Bicycle yourBike=new Bicycle();

Oba konstruktory mogły zostać zadeklarowane w Bicycle, ponieważ mają różne listy argumentów. Java różnicuje konstruktory na podstawie liczby argumentów na liście i ich typów.

Nie można utworzyć dwóch konstruktorów dla takiej samej liczby oraz typu argumentów dla tej samej klasy. Nie można by było ich wtedy rozróżnić. To byłby błąd w kompilacji.

Kompilator automatycznie podaje bezargumentowy, domyślny konstruktor dla każdej klasy bez konstruktorów.

Można stworzyć konstruktor, który ma niewiadomą ilość argumentów. Używasz varargs, gdy nie wiesz ile argumentów danego typu zostanie przekazanych do konstruktora.

Aby użyć varargs, po typie ostatniego parametru trzeba wpisać wielokropek (trzy kropki ...), a następnie nazwę parametru. Metodę można następnie wywołać z dowolną liczbą tego parametru, w tym bez żadnego. Jest to traktowane jak tablica...

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                  * (corners[1].x - corners[0].x)
                  + (corners[1].y - corners[0].y)
                  * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // więcej kodu metody, który tworzy i zwraca
    // wielokąt łączący punkty
}
```

Najczęściej zobaczysz varargs w metodach drukowania; na przykład metoda printf:

```
public PrintStream printf(String format, Object... args)
```

pozwala wydrukować dowolną liczbę obiektów. Można ją wywołać tak:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

W konstruktorze parametr może mieć taką samą nazwę jak jedno z pól klasy. Jeśli tak, to mówi się, że parametr zasłania pole.

Wywołanie jednego konstruktora z innego (tej samej klasy) (jawne wywołanie konstruktora)

W celu wywołania jednego konstruktora z innego w tej samej klasie używa się słowa kluczowego this. Wywołanie to musi być pierwszą instrukcją w konstruktorze.

This również pozwala odwołać się do pól dla aktualnego obiektu. Najczęstsze użycie: w konstruktorze, gdzie argumenty mogą mieć taką samą nazwę jak pola na obiekcie.

```java
class Example {
    private int x;
    private int y;

    // Konstruktor domyślny
    public Example() {
        this(0, 0); // Wywołanie innego konstruktora z tej samej klasy
    }

    // Konstruktor z argumentami
    public Example(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Wywołanie konstruktora klasy nadrzędnej
Aby wywołać konstruktor klasy nadrzędnej, używa się słowa kluczowego super. Wywołanie to również musi być pierwszą instrukcją w konstruktorze.

```java
class Parent {
    private int a;

    public Parent(int a) {
        this.a = a;
    }
}

class Child extends Parent {
    private int b;

    public Child(int a, int b) {
        super(a); // Wywołanie konstruktora klasy nadrzędnej
        this.b = b;
    }
}
```

Uwaga: Jeśli konstruktor nie wywołuje jawnie konstruktora klasy bazowej, kompilator Java automatycznie wstawia wywołanie konstruktora bez argumentów klasy bazowej.

Jeśli klasa bazowa nie ma konstruktora bez argumentów, otrzymasz błąd kompilacji. Klasa Object ma taki konstruktor, więc jeśli Object jest jedyną klasą bazową, nie ma problemu.

Jeśli konstruktor klasy podrzędnej wywołuje konstruktor swojej klasy bazowej, jawnie lub niejawnie, można by pomyśleć, że będzie wywoływany cały łańcuch konstruktorów, aż do konstruktora Object. W rzeczywistości tak jest. Nazywa się to łańcuchowym wywoływaniem konstruktorów (constructor chaining) i trzeba być tego świadomym, gdy jest długa linia dziedziczenia klas.

4. Wymień i omów wszystkie sposoby użycia słowa kluczowego final

Słowo kluczowe final jest ograniczeniem na rozszerzalność kodu.
final - (nie może być przypisana nowa wartość)

Finalne klasy

Klasy final nie mogą być rozszerzane. Jeśli spojrzymy na kod z podstawowych bibliotek Javy znajdziemy tam wiele finalnych klas. Jednym z przykładów klas final jest klasa String.

Rozważmy sytuacje, gdzie moglibyśmy rozszerzyć klasę String, nadpisać którąś z jej metod i podmienić wszystkie instancje String na instancje naszej specyficznej podklasy String. Wyniki na obiektach String stałyby się nieprzewidywalne. A biorąc pod uwagę, że klasa String jest używana wszędzie, byłoby to nie do przyjęcia. Dlatego ta klasa jest oznaczona jako final.

Każda próba dziedziczenia po finalnej klasie spowoduje błąd kompilatora.

Dla przykładu:

```
public final class Cat {
    private int weight;

    // standardowy getter i setter
}
```

I spróbujmy ją rozszerzyć:

```
public class BlackCat extends Cat {
}
```

Zobaczymy błąd kompilatora:

Słowo kluczowe final w deklaracji klasy nie oznacza, że obiekty tej klasy są niezmienne. Możemy swobodnie zmieniać pola obiektu klasy final.

```
Cat cat = new Cat();
cat.setWeight(1);

assertEquals(1, cat.getWeight());
```

-----> Nie możemy jej jedynie rozszerzać.

Finalne metody

Metody oznaczone jako final nie mogą być nadpisywane. Czasami nie musimy całkowicie zabraniać rozszerzania klasy, a jedynie zapobiec nadpisywaniu niektórych metod. Dobrym przykładem jest klasa Thread. Legalne jest jej rozszerzenie i utworzenie niestandardowej klasy wątku, ale jej metoda isAlive() jest finalna. Metoda ta sprawdza, czy wątek jest aktywny. Niemożliwe jest poprawne nadpisanie metody isAlive() z wielu powodów, np. Metoda ta jest natywna – zaimplementowana w innym języku programowania, często specyficznym dla systemu operacyjnego i sprzętu, na którym działa.

Jeśli niektóre metody naszej klasy są wywoływane przez inne metody, powinniśmy rozważyć oznaczenie tych metod jako finalne. W przeciwnym razie ich nadpisanie może wpłynąć na działanie wywołujących je metod i spowodować zaskakujące rezultaty.

Jaka jest różnica między oznaczeniem wszystkich metod klasy jako finalne a oznaczeniem klasy jako finalnej? W pierwszym przypadku możemy rozszerzyć klasę i dodać do niej nowe metody.

W drugim przypadku nie możemy tego zrobić.

Finalne zmienne

Zmienne oznaczone jako final nie mogą być ponownie przypisywane. Po zainicjalizowaniu finalnej zmiennej, nie można jej zmieniać.

- prymitywne

Final int i=1;

I=2; -> błąd

-referencyjne

Jeśli mamy finalną zmienną referencyjną, nie możemy jej ponownie przypisać. Ale to nie oznacza, że obiekt na który się odnosi jest niezmienny. Możemy swobodnie zmieniać właściwości tego obiektu.

Przykład:

```
final Cat cat = new Cat();
```

Jeśli spróbujemy ją ponownie przypisać, zobaczymy błąd kompilatora.

Ale możemy zmienić właściwości instancji Cat:

```
cat.setWeight(5);

assertEquals(5, cat.getWeight());
```

Finalne Pola

Mogą być albo stałymi albo polami zapisywanymi tylko raz. Aby je rozróżnić, powinniśmy zadać pytanie: czy uwzględnilibyśmy to pole, gdybyśmy mieli serializować obiekt? Jeśli nie, to nie jest to część obiektu, a stała.

```
static final int MAX_WIDTH = 999;
```

Każde finalne pole musi być zainicjalizowane przed zakończeniem konstruktora.

Dla **statycznych** finalnych pól oznacza to, że możemy je zainicjalizować:
Można je zainicjalizować przy deklaracji, w statycznym bloku inicjalizacyjnym.
Statyczne – dla klasy

Dla **instancyjnych** finalnych pól oznacza to, że możemy je zainicjalizować:
Przy deklaracji, w bloku inicjalizacyjnym instancji, w konstruktorze.
W przeciwnym przypadku kompilator zgłosi błąd.
Instancyjne – dla obiektu

Finalne Parametry

Słowo kluczowe final jest również legalne przed parametrami metod. Finalny parametr nie może być zmieniany wewnątrz metody.

```
public void methodWithFinalArguments(final int x) {
    x=1;
}
```

To spowoduje błąd kompilatora.

Podsumowując: finalna klasa, metoda, zmienne -> referencyjne, prymitywne, pola -> statyczne, instancyjne, parametry

PROGRAMOWANIE OBIEKTOWE

5. Omów porównawczo dziedziczenie klas i implementację interfejsów.

Dziedziczenie jest kluczowym elementem programowania obiektowego. W Java istnieją dwa główne sposoby dziedziczenia: poprzez klasy i interfejsy. Kluczowe słowa, które umożliwiają te mechanizmy to **extends** oraz **implements**.

Rozszerzanie klas - extends

Słowo kluczowe extends jest używane do dziedziczenia cech (pól i metod) jednej klasy przez inną klasę. Umożliwia to rozszerzenie funkcjonalności klasy bazowej w podklasie.

W Java jedna klasa może dziedziczyć tylko po jednej klasie. Używamy tego, gdy chcemy utworzyć specjalistyczną wersję klasy bazowej, zachowując i rozszerzające jej funkcjonalność.

```
class One {
    public void methodOne() {
        // Some Functionality
    }
}

class Two extends One {
    public static void main(String args[]) {
        Two t = new Two();
        t.methodOne(); // Calls the method from One
    }
}
```

Implementacja interfejsów (implements)

Implements jest używane do implementowania interfejsów. Interfejs to specjalny typ klasy, który definiuje zbiór abstrakcyjnych metod, które muszą być zaimplementowane przez klasę.
Klasa może implementować dowolną liczbę interfejsów. Klasa implementująca interfejs musi zaimplementować wszystkie metody tego interfejsu.
Używamy **implements**, gdy chcemy zagwarantować, że klasa dostarcza określoną funkcjonalność, definiowaną przez interfejs.

```
// Defining an interface
interface One {
    public void methodOne();
}

// Defining another interface
interface Two {
    public void methodTwo();
}

// Implementing both interfaces
class Three implements One, Two {
    public void methodOne() {
        // Implementation of the method
    }

    public void methodTwo() {
        // Implementation of the method
    }
}
```

RÓŻNICE: (chat)

Dziedziczenie klas, a implementacja interfejsów:

-**Extends** umożliwia klasie dziedziczenie po innej klasie, lub interfejsowi dziedziczenie po innych interfejsach

-**Implements** umożliwia klasie implementowanie interfejsu, co oznacza dostarczanie konkretnej implementacji metod zdefiniowanych w interfejsie

Obowiązkowość implementacji metod:
-Przy użyciu extends nie jest konieczne nadpisywanie wszystkich metod z klasy nadrzędnej,
-przy użyciu implements klasa musi zaimplementować wszystkie metody interfejsu
class abc implements interfejs1, interfejs2, interfejs3

Dziedziczenie pojedynczej klasy vs implementacja wielu interfejsów:
-Klasa może dziedziczyć tylko po jednej klasie, co ogranicza wielokrotne dziedziczenie
-Klasa może implementować dowolną liczbę interfejsów, co pozwala na wielokrotne dziedziczenie interfejsów

Dziedziczenie interfejsów:
-Interfejs może dziedziczyć po wielu interfejsach używając extends,
-Interfejs nie może implementować innego interfejsu

6. Omów porównawczo klasy abstrakcyjne i interfejsy.

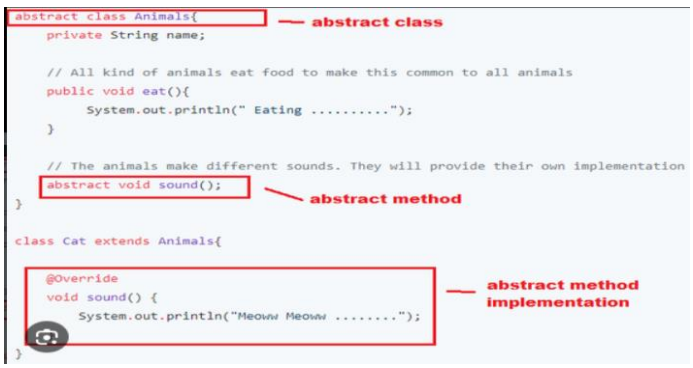
Klasa vs Interfejs

Klasa:
-Definiowana przez użytkownika struktura służąca jako szablon do tworzenia obiektów
-Może mieć pola i metody, które reprezentują stan i zachowania obiektu

Interfejs:
-Definiowana przez użytkownika struktura podobna do klasy,
-Może zawierać kolekcje stałych pól i sygnatur, metod, które będą nadpisywane przez klasy implementujące interfejs,
-Metody statyczne i domyślne w interfejsach, dodawanie prywatnych metod

Interfejs vs Klasa abstrakcyjna

Klasa abstrakcyjna:
-deklarowana za pomocą słowa kluczowego abstract
-może deklarować sygnatury metod za pomocą słowa kluczowego abstract, zmuszając podklasy do implementacji tych metod.
-może zawierać pola i metody o dowolnych modyfikatorach dostępu,
-może zawierać bloki inicjalizacyjne instancji i statyczne oraz konstruktory, które są wykonywane podczas instancjowania obiektu



Interfejs:

-Może zawierać stałe pola i sygnatury metod, które muszą być zaimplementowane przez klasy implementujące interfejs.
-Wszystkie metody są domyślnie publiczne i abstrakcyjne z wyjątkiem metod statycznych i domyślnych.
-Możliwe jest dodawanie metod domyślnych i statycznych, prywatnych.

Kiedy używać interfejsu?

-Używane, gdy chcemy zdefiniować kontrakt dla niepowiązanych klas.
-Wszystkie metody są domyślnie publiczne i abstrakcyjne.
-Od Java 8 mogą zawierać metody domyślne i statyczne, a od Java 9 prywatne.

Kiedy używać klasy abstrakcyjnej?

Gdy chcemy użyć koncepcji dziedziczenia, dostarczając wspólne metody bazowe, które podklasy nadpisują. Jeśli mamy określone wymagania i tylko częściowe szczegóły implementacji, Gdy klasy rozszerzające klasę abstrakcyjną mają kilka wspólnych pól lub metod (które wymagają niepublicznych modyfikatorów), Kiedy chcemy mieć metody, które nie są finalne lub statyczne do modyfikowania stanu obiektu

W skrócie:
-Używane, gdy chcemy dzielić kod między powiązane klasy.
-Mogą mieć pola i metody o dowolnych modyfikatorach.
-Umożliwiają częściową implementację metod, które podklasy muszą nadpisać.

7. Omów polimorfizm

Jest jednym z czterech podstawowych filarów programowania obiektowego, obok abstrakcji, enkapsulacji i dziedziczenia. Polimorfizm pozwala obiektom różnych klas reagować na te same metody w sposób specyficzny dla swojej klasy. Tutaj omówimy dwa rodzaje: statyczny (kompilacyjny) oraz dynamiczny (runtime)

Polimorfizm Statyczny(kompilacyjny) - Jest rozwiązywany na etapie kompilacji. Oznacza to, że kompilator określa, która metoda powinna zostać wywołana na podstawie sygnatury metody.

```
public class TextFile extends GenericFile {
    //...

    public String read() {
        return this.getContent().toString();
    }

    public String read(int limit) {
        return this.getContent().toString().substring(0, limit);
    }

    public String read(int start, int stop) {
        return this.getContent().toString().substring(start, stop);
    }
}
```

W powyższym przykładzie, klasa TextFile posiada trzy przeciążone metody read(). Kompilator podczas kompilacji sprawdza, która z tych metod powinna zostać wywołana na podstawie podanych argumentów.

Polimorfizm Dynamiczny - Jest rozwiązywany w czasie wykonywania programu przez Java Virtual Machine (JVM). Umożliwia on obiektom podklas wywoływanie odpowiednich metod, które mogą być nadpisane w

klasach potomnych.

```
public class GenericFile {
    private String name;

    public String getFileInfo() {
        return "Generic File Impl";
    }
}

public class ImageFile extends GenericFile {
    private int height;
    private int width;

    @Override
    public String getFileInfo() {
        return "Image File Impl";
    }
}

public static void main(String[] args) {
    GenericFile genericFile = new ImageFile("SampleImageFile", 200, 100,
      new BufferedImage(100, 200, BufferedImage.TYPE_INT_RGB).toString().getBytes(),
    System.out.println("File Info: \n" + genericFile.getFileInfo());
}
```

W powyższym przykładzie, genericFile.getFileInfo() wywołuje metodę getFileInfo() z klasy ImageFile, mimo że typem obiektu jest GenericFile. Dzieje się tak dlatego, że JVM w czasie wykonywania utrzymuje odniesienie do rzeczywistego typu obiektu ImageFile.

Inne charakterystyki polimorficzne w Java:

Koercja – odnosi się do niejawnej konwersji typów wykonywanej przez kompilator, aby zapobiec błędom typów. Np.:
String str = “string” + 2;

Przeciążenie operatorów

Polega na użyciu tego samego symbolu lub operatora w różnych kontekstach. Np. Symbol (+) może być użyty do dodawania liczb lub konkatenacji łańcuchów znaków

```
String str = "2" + 2;
int sum = 2 + 2;
System.out.printf("str = %s\nsum = %d\n", str, sum);
```

Polimorficzne Parametry

Parametryczna polimorfia pozwala nazwie parametru lub metody w klasie być powiązana z różnymi typami. Np.

```
public class TextFile extends GenericFile {
    private String content;

    public String setContentDelimiter() {
        int content = 100;
        this.content = this.content + content;
    }
}
```

Polimorficzne podtypy

Polimorficzne podtypy pozwalają na przypisanie wielu podtypów do jednego typu i wywoływanie metod dostępnych w tych podtypach:

```
GenericFile [] files = {new ImageFile("SampleImageFile", 200, 100,
  new BufferedImage(100, 200, BufferedImage.TYPE_INT_RGB).toString()
  .getBytes(), "v1.0.0"), new TextFile("SampleTextFile",
  "This is a sample text content", "v1.0.0")};

for (int i = 0; i < files.length; i++) {
    files[i].getInfo();
}
```

PROBLEMY Z POLIMORFIZMEM

-Identyfikacja typów podczas rzutowania

Rzutowanie w górę i w dół mogą prowadzić do błędów w czasie wykonywania, jeśli typ nie jest prawidłowy.

```
GenericFile file = new GenericFile();
ImageFile imageFile = (ImageFile) file;
System.out.println(imageFile.getHeight());
```

Powyższy kod spowoduje ClassCastException, ponieważ GenericFile nie jest instancją ImageFile.

-Problem delikatnej klasy bazowej

Zmiany w klasie bazowej mogą powodować problemy w klasach pochodnych.

8. Omów enkapsulacje. Przedstaw modyfikatory dostępu w kontekście enkapsulacji. Uwzględnij zagnieżdżenie klas.

Enkapsulacja polega na ukrywaniu wewnętrznych szczegółów implementacji obiektów i udostępnianiu tylko tych elementów, które są niezbędne do korzystania z tych obiektów. Jest realizowana w Javie głównie za pomocą modyfikatorów dostępu, które określają, które części kodu mogą uzyskać dostęp do pól i metod klasy.

Modyfikatory dostępu w Javie:

-public - człon publiczny jest dostępny z dowolnego miejsca. Inne klasy zarówno w tym samym pakiecie, jak i w innych pakietach, mogą uzyskiwać dostęp do pól i metod oznaczonych jako publiczne.

-protected - człon chroniony jest dostępny w obrębie tego samego pakietu oraz w podklasach, nawet jeśli znajdują się one w różnych pakietach. Często jest używany w dziedziczeniu, aby umożliwić dostęp do członów klasy bazowej przez klasy pochodne.

-default (package-private): gdy nie zostanie określony żaden modyfikator dostępu, domyślnym poziomem dostępu jest dostęp pakietowy. Człony z dostępem pakietowym są dostępne tylko dla innych klas w tym samym pakiecie. Nie są one dostępne z innych pakietów.

-private: Człon prywatny jest dostępny tylko w obrębie klasy, w której został zadeklarowany. Żadna inna klasa, nawet ta w tym samym pakiecie, nie może uzyskać dostępu do pól lub metod oznaczonych jako prywatne.

Przykład zagnieżdżenia klas:

Zagnieżdżenie odnosi się do sytuacji, gdy jedna klasa jest zdefiniowana wewnątrz innej klasy.. Modyfikatory dostępu również mają zastosowanie do zagnieżdżonych klas.

9. Przedstaw rodzaje asocjacji między klasami. Zilustruj przykładami i schematami UML.

Asocjacje między klasami w programowaniu obiektowym opisują relacje i powiązania między obiektami. W Javie wyróżniamy trzy główne typy asocjacji (asocjacja, agregacja, kompozycja)

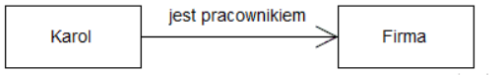
1) Asocjacja – to ogólne powiązanie między dwoma oddzielnymi klasami, które mogą komunikować się ze sobą. Może być jednostronna lub dwustronna.

Przykład:

```
class Teacher {
    private String name;
    // Konstruktor, gettery i settery
}

class Student {
    private String name;
    // Konstruktor, gettery i settery
}

class School {
    private List<Teacher> teachers;
    private List<Student> students;
    // Konstruktor, metody do dodawania nauczycieli i uczniów
}
```

Schemat UML:

```
lua

+---------------+    +---------------+
|    Teacher    |    |    Student    |
+---------------+    +---------------+
        |                    |
        |                    |
        +----------+---------+
                   |
                   |
             +----------+
             |  School  |
             +----------+
```

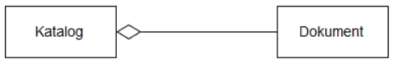


2) Agregacja – to szczególny rodzaj asocjacji, który wskazuje, że jedna klasa jest częścią innej klasy, ale może istnieć niezależnie od niej. Obiekty składowe nie są zarządzane przez obiekt nadrzędny.

```
class Department {
    private String name;
    private List<Teacher> teachers;
    // Konstruktor, gettery i settery
}
```

Schemat UML:

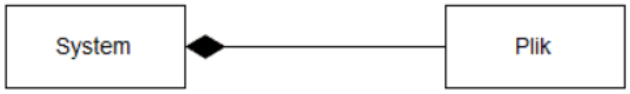
```
lua

+---------------+    +---------------+
|    Teacher    |    |  Department   |
+---------------+    +---------------+
        |                    |
        +----------+---------+
                   |
                   <>  // Linia z diamentem oznaczająca agregację
```



3) Kompozycja – silniejszy rodzaj agregacji, który wskazuje, że jedna klasa jest częścią innej klasy i nie może istnieć bez niej. Obiekty składowe są zarządzane przez obiekt nadrzędny i są niszczone razem z nim.

```
class Library {
    private List<Book> books;

    public Library() {
        books = new ArrayList<>();
    }

    public void addBook(String title) {
        books.add(new Book(title));
    }

    // Inne metody
}

class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    // Gettery i settery
}
```

```
+---------------+    +---------------+
|     Book      |    |    Library    |
+---------------+    +---------------+
        |                    |
        +----------+---------+
                   |
                   ++  // Linia z pełnym diamentem oznaczająca kompozycję
```



Podsumowując:

-Asocjacja to luźne powiązanie, gdzie obiekty mogą istnieć niezależnie

-Agregacja to relacja “część-całość”, gdzie obiekty częściowe mogą istnieć niezależnie od obiektu całościowego

-Kompozycja to silna relacja “część-całość”, gdzie obiekty częściowe są integralną częścią obiektu całościowego i nie mogą istnieć bez niego.

WZORCE PROJEKTOWE

10. Singleton – konstrukcja, zastosowanie, ograniczenia i problemy

Singleton zapewnia istnienie tylko jednej instancji danej klasy i globalny dostęp do niej.

Konstrukcja:

2. Prywatny konstruktor, aby zapobiec tworzeniu obiektów z zewnątrz.
3. Prywatne, statyczne pole przechowujące instancję klasy.
4. Statyczna metoda `getInstance()`, która zwraca instancję:
    - Jeśli pole jest `null`, tworzy nową instancję i ją zwraca.
    - Jeśli pole nie jest `null`, zwraca istniejącą instancję.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Singleton oparty na klasie:
-prywatny konstruktor
-statyczne pole przechowujące instancję
-statyczna metoda getInstance()

Singleton za pomocą Enum

```
public enum EnumSingleton {

    INSTANCE("Initial class info");

    private String info;

    private EnumSingleton(String info) {
        this.info = info;
    }

    public EnumSingleton getInstance() {
        return INSTANCE;
    }

    // gettery i settery
}
```

Podejście to gwarantuje serializacje i bezpieczeństwo wątków przez samą implementację wyliczenia, co zapewnia wewnętrznie, że dostępna jest tylko jedna instancja, rozwiązując problemy wskazane w implementacji opartej na klasach.

Singleton jest używany w:
-Logowaniu, Ustawieniach konfiguracyjnych, Buforowaniu, Pulach połączeń

Typowe pułapki:

Wyróżniamy dwa rodzaje problemów z Singletonami:
-**egzystencjalne** – czy naprawdę potrzebujemy singletona?
-**implementacyjne** – czy poprawnie go zaimplementowaliśmy?

Problemy egzystencjalne:

- Singleton to globalna zmienna, której zmienny stan jest trudny do zarządzania.
- Trudność w przełączaniu konfiguracji dla różnych środowisk, np. testowego i produkcyjnego

Problemy implementacyjne:

- **Brak synchronizacji:** W środowisku wielowątkowym może prowadzić do stworzenia wielu instancji.

Problemy JVM:

- Singleton może być zniszczony przez garbage collectora i stworzony na nowo, co skutkuje różnymi instancjami.
- W systemach rozproszonych, każda JVM może mieć swoją instancję Singletona.

- Różne klasy loaderów mogą tworzyć własne instancje Singletona.

Rozwiązania:

```
public static synchronized Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

Synchronizacja - użycie słowa kluczowego synchronized w metodzie getInstance() zapewnia, że tylko jeden wątek może utworzyć instancję.
Enum: Zapewnia bezpieczeństwo wątków i serializację.

Podsumowanie: Singleton to wzorzec projektowy gwarantujący istnienie jednej instancji klasy z globalnym punktem dostępu, przydatny w kontekstach takich jak logowanie i konfiguracje. Problemy mogą wynikać z niewłaściwej synchronizacji oraz specyficznych zachowań JVM w środowiskach wielowątkowych i rozproszonych.

11. Budowniczy – konstrukcja i zastosowanie (przykład z domem)

Budowniczy umożliwia tworzenie złożonych obiektów krok po kroku, pozwalając na produkcję różnych typów i reprezentacji obiektu przy użyciu tego samego kodu konstrukcyjnego.

Konstrukcja: Budowniczy oddziela kod konstrukcyjny obiektu od jego klasy i umieszcza go w osobnych obiektach zwanych budowniczymi. Proces budowy jest podzielony na etapy (np. buildWalls(), buildDoors()), które mogą być wywoływane w zależności od potrzeb.

Kierownik: Aby zarządzać kolejnością wywołań budowniczych, wprowadza się klasę kierownika, która:
Przechowuje kolejkę wywołań budowniczych.
Określa kolejność etapów konstrukcji.
Ukrywa szczegóły konstrukcji przed kodem klienckim.

Klient łączy budowniczego z kierownikiem, wywołuje proces budowy i odbiera wynik od budowniczego.

Struktura:
Interfejs budowniczego - Definiuje etapy konstrukcji.
Konkretni budowniczowie - Implementują etapy, tworząc różne reprezentacje obiektów.
Kierownik - Zarządza porządkiem konstruowania obiektu.

Przykład:
Budowa samochodów:
Klasa budowniczego samochodu posiada metody do konfiguracji różnych części auta.
Klasa kierownika zna sekwencje konstrukcji popularnych modeli.

Klient może bezpośrednio używać budowniczego do tworzenia niestandardowych modeli lub skorzystać z kierownika dla standardowych modeli.
Wynikowa metoda zwracająca nowo utworzony obiekt znajduje się w budowniczym.

Zastosowanie:
Unikanie "teleskopowych konstruktorów":
Przy wielu parametrach konstruktora, wzorzec budowniczy pozwala na tworzenie obiektów krok po kroku.

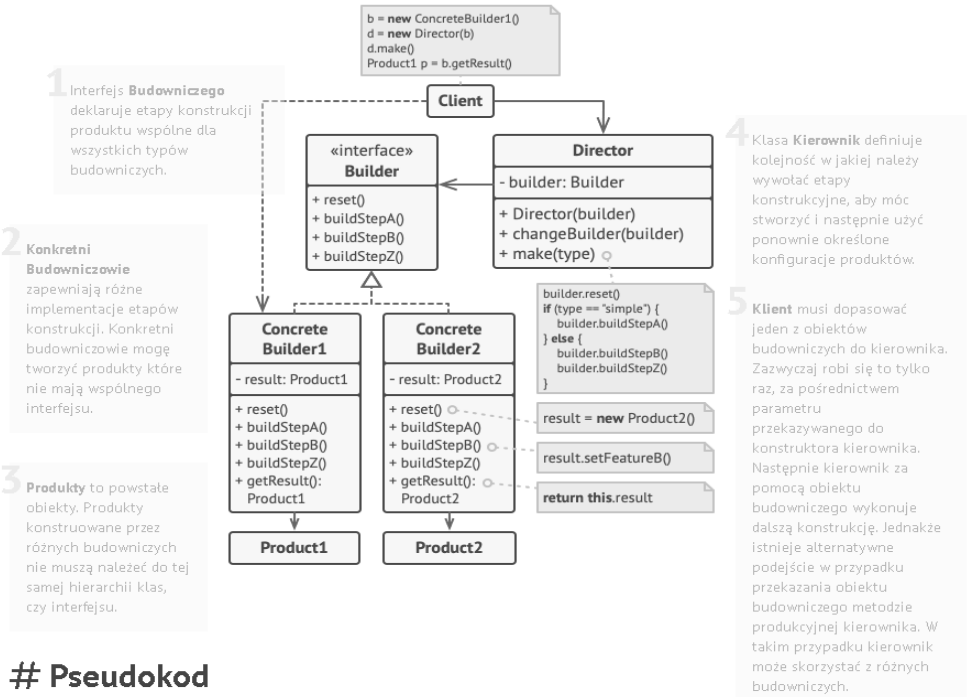
Różne reprezentacje produktów:

Budowanie np. domów z kamienia i drewna, gdzie etapy konstrukcji są podobne, ale różnią się szczegółami.

Złożone obiekty:
Budowa drzew kompozytowych i innych złożonych struktur.

Zalety:
Możliwość etapowego konstruowania obiektów.
Odkładanie lub rekursywne wykonywanie niektórych etapów.
Izolacja skomplikowanego kodu konstrukcyjnego od logiki biznesowej.
Ponowne wykorzystanie tego samego kodu do budowy różnych reprezentacji produktów.
Wady:
Większa złożoność kodu ze względu na dodatkowe klasy.

STRUKTURA - obrazek ze strony:



- # Pseudokod

Bazowy interfejs budowniczego definiuje wszelkie możliwe etapy konstrukcji, a konkretni budowniczy implementują te kroki by móc tworzyć poszczególne reprezentacje obiektów. Natomiast klasa kierownik pilnuje właściwego porządku konstruowania.

12. Dekorator – konstrukcja i zastosowanie

Dekorator pozwala dodawać nowe funkcje obiektom poprzez umieszczanie ich w specjalnych obiektach opakowujących, które implementują te same metody. Dzięki temu obiekty dekorowane zachowują się jak oryginalne, ale mogą mieć dodatkowe funkcje.

Przykład zastosowania(dla lepszego zrozumienia):
W bibliotece używającej klasy Powiadamiacz do wysyłania e-maili użytkownicy zaczęli wymagać powiadomień SMS, przez Facebooka, oraz Slacka. Rozszerzanie klasy poprzez dziedziczenie szybko prowadziło do złożonego i trudnego do zarządzania kodu.

Dekorator umożliwia dynamiczne dodawanie funkcji powiadamiania bez zmiany kodu bazowego. Klient może opakować obiekt Powiadamiacz wieloma dekoratorami, tworząc stos, w którym każdy dekorator dodaje swoją funkcję.

Struktura dekoratora:

1. Komponent - Deklaruje wspólny interfejs dla obiektów i dekoratorów.
5. Konkretny Komponent - Definiuje podstawowe zachowanie.
6. Bazowy Dekorator - Posiada referencję do opakowywanego obiektu i deleguje mu działania.
7. Konkretni Dekoratorzy - Nadpisują metody bazowego dekoratora, dodając nowe funkcje przed lub po delegacji do opakowywanego obiektu.
8. Klient może opakowywać komponenty w wiele warstw dekoratorów, o ile działa na wszystkich obiektach poprzez interfejs komponentu.

Przykład techniczny:

Dekoratory mogą kompresować i szyfrować dane przed ich zapisaniem oraz dekompresować i odszyfrować po odczytaniu. Dzięki temu dane są bezpieczne bez zmiany kodu źródłowego, który je przetwarza.

Zastosowanie dekoratora:

- Gdy chcesz dynamicznie przypisywać dodatkowe funkcje obiektom w czasie działania programu.
- Gdy rozszerzanie funkcji obiektu przez dziedziczenie jest niepraktyczne.

Dekorator pozwala elastycznie dodawać funkcje obiektom bez zmiany ich oryginalnej struktury, ułatwiając zarządzanie i rozbudowę kodu.

KOLEKCJE

13. Omów interfejs Collection

Interfejs Collection<E> Jest korzeniem hierarchii kolekcji w Javie. Reprezentuje grupę obiektów, znanych jako jej elementy. Niektóre kolekcje pozwalają na duplikaty elementów, inne nie. Niektóre są uporządkowane, inne nie. JDK nie dostarcza bezpośrednich implementacji tego interfejsu: dostarcza implementacje bardziej specyficznych podinterfejsów, takich jak Set i List. Ten interfejs jest zazwyczaj używany do przekazywania kolekcji i manipulowania nimi, gdzie wymagane jest maksymalne uogólnienie.

Multizbiory (kolekcje nieuporządkowane, które mogą zawierać duplikaty elementów) powinny bezpośrednio implementować ten interfejs.

Wszystkie klasy implementujące Collection ogólnego przeznaczenia powinny dostarczyć dwa “standardowe” konstruktory: pusty, który tworzy pustą kolekcję oraz konstruktor z pojedynczym argumentem typu Collection, który tworzy nową kolekcję z tymi samymi elementami co argument.

W efekcie ten drugi konstruktor pozwala użytkownikowi skopiować dowolną kolekcję, produkując równoważną kolekcję o pożądanym typie implementacji. Nie ma sposobu na wymuszenie tej konwencji, ale wszystkie implementacje kolekcji ogólnego przeznaczenia w bibliotekach platformy Java jej przestrzegają.

“Destrukcyjne” metody zawarte w tym interfejsie, czyli metody modyfikujące kolekcje na której operują, są określone, aby rzucać UnsupportedOperationException jeśli kolekcja nie obsługuje operacji. W takim przypadku te metody mogą ale nie muszą rzucać tego wyjątku, jeśli wywołanie nie miałoby wpływu na kolekcję. Np. Wywołanie metody addAll(Collection) na niezmiennej kolekcji może, ale nie musi, rzucać wyjątku, jeśli kolekcja do dodania jest pusta.

Niektóre implementacje kolekcji mają ograniczenia co do elementów, które mogą zawierać. Np. Niektóre zabraniają elementów null, a inne mają ograniczenia co do typów swoich elementów. Próba dodania nieuprawnionego elementu rzuca niekontrolowany wyjątek, zazwyczaj NullPointerException lub ClassCastException.

Każda kolekcja musi sama określić swoją politykę synchronizacji. W przypadku braku silniejszej gwarancji przez implementację, wywołanie jakiejkolwiek metody na kolekcji modyfikowanej przez inny wątek może prowadzić do niezdefiniowanego zachowania

Niektóre operacje kolekcji, które wykonują rekursywne przeszukiwanie kolekcji, mogą kończyć się wyjątkiem dla instancji samoodwołujących się, gdzie kolekcja bezpośrednio lub pośrednio zwraca samą siebie. Obejmuje to metody: clone(), equals(), hashCode() i toString(). Implementacje mogą opcjonalnie obsłużyć scenariusz samoodwołujący się, jednak większość obecnych implementacji tego nie robi.

Ten interfejs jest członkiem Java Collections Framework.

Domyślne implementacje metod nie stosują żadnego protokołu Wymagania implementacyjne: Jeśli implementacja Collection ma specyficzny protokół synchronizacji, to musi nadpisać domyślne implementacje, aby zastosować ten protokół.

14. Porównaj klasy LinkedList i ArrayList

Java oferuje wiele opcji kolekcji w swojej standardowej bibliotece, w tym dwa popularne typy list: ArrayList, LinkedList.

1) ArrayList

Używa tablicy do implementacji interfejsu List. Ponieważ tablice w Javie mają stały rozmiar, ArrayList tworzy tablicę z początkową pojemnością. Jeśli trzeba przechowywać więcej elementów niż ta domyślna pojemność, ArrayListy zamienia tablicę na nową, większą

Dodawanie - Podczas tworzenia pustej ArrayList inicjuje ona swoją tablicę z domyślną pojemnością (obecnie 10). Dodanie nowego elementu jest proste, gdy tablica nie jest jeszcze pełna:

```
BackingArray[size] = newItem;
size++;
```

W najlepszym i średnim przypadku, złożoność czasowa operacji dodawania wynosi O(1). Gdy tablica jest pełna, konieczne jest utworzenie nowej tablicy z większą pojemnością i skopiowanie wszystkich istniejących elementów do nowej tablicy, co ma złożoność O(n) w najgorszym przypadku.

Dostęp po indeksie – jest szybki, aby pobrać element pod indeksem i, wystarczy zwrócić element znajdujący się pod tym indeksem, co ma złożoność O(1).

Usuwanie po indeksie – wymaga przesunięcia wszystkich elementów za nim o jeden indeks wstecz. Złożoność czasowa O(1) w najlepszym przypadku, w najgorszym O(n)

Zastosowania i ograniczenia - ArrayList jest często wybierana, gdy liczba operacji odczytu jest znacznie większa niż liczba operacji zapisu. Jeśli można oszacować maksymalną liczbę elementów, sensowne jest inicjalizowanie ArrayList z określoną pojemnością, aby uniknąć niepotrzebnych kopii i alokacji tablic.

ArrayList nie może przechowywać więcej niż w^31-1 elementów z powodu ograniczeń int w indeksowaniu tablic w Javie.

2) LinkedList

LinkedList używa zbioru połączonych węzłów do przechowywania i pobierania elementów. Każdy węzeł zawiera dwa wskaźniki: jeden wskazujący na następny element i drugi na poprzedni. LinkedList jest implementacją listy dwukierunkowej.

Dodawanie – dodanie nowego węzła wymaga połączenia obecnego ostatniego węzła z nowym węzłem i zaktualizowania wskaźnika ostatniego węzła Złożoność tej operacji to zawsze O(1).

Dostęp po indeksie – LinkedList nie wspiera szybkiego dostępu losowego. Aby znaleźć element po indeksie, trzeba ręcznie przeszukać część listy. W najlepszym przypadku złożoność czasowa to O(1), ale w średnim i najgorszym, może być to O(n).

Usuwanie po indeksie – aby usunąć element, należy go najpierw znaleźć, a następnie odłączyć od listy. Złożoność czasowa zależy od czasu dostępu, czyli O(1) w najlepszym przypadku, a w innych to O(n).

Zastosowanie - jest bardziej odpowiednia, gdy liczba operacji dodawania jest znacznie większa niż liczba operacji odczytu. Jest także użyteczna w sytuacjach, gdy często potrzebujemy dostępu do pierwszego lub ostatniego elementu. LinkedList implementuje także interfejs Deque, wspierając efektywny dostęp do obu końców kolekcji.

15. W C++ jest typ std::multimap, jak zrekonstruować jego działanie w Javie?

std::multimap w C++ to kontener asocjacyjny, który przechowuje posortowaną listę par klucz-wartość, umożliwiając wielokrotne wstawianie par z tym samym kluczem. Java nie posiada bezpośredniego odpowiednika std::multimap, ale możemy uzyskać podobną funkcjonalność za pomocą klasy TreeMap oraz kolekcji List do przechowywania wartości.

TreeMap do przechowywania kluczy. TreeMap utrzymuje klucze w posortowanej kolejności, zapewnia wydajne operacje wyszukiwania, wstawiania i usuwania
Listy do przechowywania wartości. Każdy klucz w TreeMap jest powiązany z listą wartości. Pozwala to na przechowywanie wielu wartości dla jednego klucza, co jest główną cechą ‘multimap’

Działanie:
-Dodanie nowej pary klucz-wartość sprawdza, czy klucz już istnieje w TreeMap. Jeśli tak, nowa wartość jest dodawana do listy wartości, jeśli nie, to jest tworzona nowa lista dla tego klucza.
-Pobranie wszystkich wartości dla danego klucza zwraca listę wartości
-Usunięcie wartości dla klucza usuwa ją z odpowiedniej list wartości.

Metody takie jak put, get, remove, containsKey, containsValue, size oraz iterowanie po elementach mapy są zaimplementowane, aby spełniać wymagania podobne do tych w std::multimap w C++.

Zastosowanie:
-Przydatne, gdy wymagane jest przechowywanie wielu wartości dla jednego klucza oraz utrzymanie kluczy w posortowanej kolejności
-Typowe w aplikacjach wymagających mapowania z wieloma wartościami, jak np. Indeksy dokumentów, katalogi produktów, itp.

PLIKI

16. Readery - podział, różnice, zastosowanie(Tu chyba bez tych szczegółowych o inputStreamReader i BufferedReader)

Java Reader to abstrakcyjna klasa służąca do odczytu strumieni znaków. Klasa ta definiuje podstawowe metody do odczytu i zamykania strumieni, które muszą być zaimplementowane przez podklasy. Kluczowym zadaniem Reader jest umożliwienie odczytu danych tekstowych z różnych źródeł, tj. Pliki, tablice znaków czy strumienie wejściowe.

Podział:

BufferedReader – zapewnia buforowane odczytywanie znaków, tablic i linii zwiększając wydajność
CharArrayReader – czytnik, który odczytuje dane z tablicy znaków
FilterReader – abstrakcyjna klasa dla strumieni filtrujących
InputStreamReader - przekształca strumień bajtów na strumień znaków
PipedReader - strumień, który można połączyć z PipedWriter, tworząc połączenie między dwoma wątkami
StringReader – czytnik, który odczytuje dane z ciągu znaków

Różnice między Readerami:

BufferedReader vs CharArrayReader:

BufferedReader buforuje dane dla wydajności, podczas gdy CharArrayReader odczytuje dane bezpośrednio z tablicy znaków.

InputStreamReader vs PipedReader:

InputStreamReader konwertuje strumień bajtów na znaki, natomiast PipedReader odczytuje dane z PipedWriter, co umożliwia komunikacje między wątkami.

Zastosowanie:

BufferedReader - używany do odczytu dużych ilości tekstu, gdzie wydajność jest kluczowa
CharArrayReader – przydatny do odczytu danych z tablic znaków
FilterReader –podstawowa klasa do tworzenia filtrów przetwarzających strumień znaków
InputStreamReader - używany do odczytu danych tekstowych z wejścia bajtowego, takiego jak pliki lub sieć
PipedReader – idealny do komunikacji między wątkami
StringReader – przydatny do odczytu danych z ciągów znaków, np. Podczas testowania

17. Serializacja obiektów. Zapis i odczyt obiektów z pliku

Jakieś 10 min warto obejrzeć V

Serializacja to proces konwersji stanu obiektu na strumień bajtów, który można następnie zapisać do pliku, bazy danych lub przesłać przez sieć. Deserializacja to proces odwrotny, polegający na odtworzeniu obiektu z zapisanego strumienia bajtów. Serializacja umożliwia trwałe przechowywanie obiektów oraz ich wymianę między różnymi platformami.
Serializacja w Javie jest niezależna od instancji, co oznacza, że obiekty mogą być serializowane na jednej platformie, a deserializowane na innej. Klasy, które mogą być serializowane, muszą implementować specjalny interfejs Serializable.

Do zapisu i odczytu obiektów służą klasy ObjectOutputStream i ObjectInputStream oraz metody:
writeObject() - konwertuje obiekt na strumień bajtów,
readObject() - odczytuje strumień bajtów i konwertuje go na obiekt Javy.

```
public class Person implements Serializable {
    private int age;
    private String name;
    private Address country; // Address musi być również serializowalny
}
```

Jeśli klasa implementuje Serializable, to jej wszystkie podklasy również są serializowane. Jeśli obiekt zawiera referencję do innego obiektu, ten drugi obiekt także musi implementować Serializable, w przeciwnym razie zostanie rzucony wyjątek NotSerializableException.

Java umożliwia nadpisanie domyślnego sposobu serializacji obiektów za pomocą metod writeObject i readObject, jest to przydatne, gdy obiekt zawiera atrybuty, które nie są serializowane

WYJĄTKI

18. Omów składnie i zastosowanie wyjątków

Wyjątki są nieoczekiwanymi zdarzeniami, które występują podczas wykonywania programu i zakłócają jego normalny przebieg. Można je przechwycić i obsłużyć, aby utrzymać ciągłość działania programu.

Główne powody występowania wyjątków:

-nieprawidłowe dane wejściowe od użytkownika, awaria urządzenia, utrata połączenia sieciowego, ograniczenie sprzętowe, błędy w kodzie, próba otwarcia niedostępnego pliku

Różnica między błędami, a wyjątkami:

-Błędy - poważne problemy, których aplikacja nie powinna próbować przechwytywać np. Wyciek pamięci
-Wyjątki - problemy, które aplikacja może próbować obsłużyć, np. IOException

Hierarchia wyjątków

-Wszystkie typy wyjątków i błędów dziedziczą po klasie "Throwable",
-Exception – klasa używana do wyjątkowych sytuacji, które można obsłużyć
-Error – klasa używana przez JVM do zgłaszania błędów środowiska wykonawczego

Rodzaje wyjątków:

1) Wbudowane wyjątki:

-Checked Exceptions – sprawdzane w czasie kompilacji, np. IOException
-Unchecked exceptions – nie są sprawdzane w czasie kompilacji, np. NullPointerException

Wyjątki definiowane przez użytkownika:

Można definiować własne wyjątki, gdy wbudowane nie opisują odpowiednio sytuacji

Zalety obsługi wyjątków w Javie:

-Umożliwia dokończenie wykonywania programu
-Łatwe oddzielenie kodu programu od kodu obsługi błędów,
-Propagacja błędów
-Znaczące raportowanie błędów
-Identyfikacja typów błędów

Kluczowe metody obsługi wyjątków:

-printStackTrace() - drukuje informacje o wyjątku, w tym nazwę, opis i ślad stosu
-toString() - drukuje nazwę i opis wyjątku
-getMessage() - drukuje tylko opis wyjątku

Jak JVM obsługuje wyjątki?

1. Tworzy obiekt wyjątku i przekazuje go do systemu wykonawczego
2. JVM przeszukuje stos wywołań, aby znaleźć odpowiedni blok 'catch',
3. Jeśli nie znajdzie, program kończy się niepowodzeniem

Jak programista obsługuje wyjątki?

-try – blok kodu, który może zgłaszać wyjątków
-catch – blok kodu, który obsługuje zgłoszone wyjątki
-throw - służy do ręcznego zgłaszania wyjątku
-throws – deklaruje, że metoda może zgłaszać wyjątki
-finally – blok kodu, który jest zawsze wykonywany po bloku 'try' (niezależnie czy wyjątek wystąpił)(ten blok jest

```
try {
    // kod, który może zgłaszać wyjątki
} catch (TypWyjątku1 ex) {
    // obsługa wyjątku TypWyjątku1
} catch (TypWyjątku2 ex) {
    // obsługa wyjątku TypWyjątku2
} finally {
    // kod, który zawsze zostanie wykonany
}
```

opcjonalny)

19. Wyjątek Exception, a RuntimeException, wyjaśnij różnicę i podaj przykład

Exception

To główna klasa reprezentująca wyjątki w Javie. Wszystkie klasy wyjątków dziedziczą bezpośrednio lub pośrednio z Exception.

Sprawdzane wyjątki - wyjątki które są bezpośrednimi podklasami Exception i nie są podklasami RuntimeException, nazywane są sprawdzanymi wyjątkami. Muszą być zadeklarowane w sygnaturze metody za pomocą klauzuli throws lub być obsłużone w bloku try-catch.

Przykłady: IOException, SQLException, ClassNotFoundException

Charakterystyka: muszą być deklarowane lub obsługiwane, są używane do reprezentowania sytuacji, które mogą wystąpić podczas normalnej pracy programu ale są spodziewane (np. Problemy z dostępem do plików, błędy sieciowe). Są sprawdzane przez kompilator, który wymusza ich obsługę lub deklarację.

RuntimeException

Ta klasa jest podklasą Exception i reprezentuje wyjątki, które mogą wystąpić podczas normalnej pracy maszyny wirtualnej Javy, ale nie są oczekiwane.

Niesprawdzane wyjątki - wyjątki, które są podklasami RuntimeException, nie muszą być deklarowane w sygnaturze metody za pomocą klauzuli throws ani być obsługiwane w bloku try-catch.

Przykłady: NullPointerException, ArrayIndexOutOfBoundsException,ArithmeticException

Charakterystyka: Nie muszą być deklarowane ani obsługiwane, reprezentują błędy programistyczne, tj. Dostęp do nullowej referencji lub nieprawidłowy indeks tablicy. Nie są sprawdzane przez kompilator pod kątem deklaracji ani obsługi.

20. Wyrażenie try-with-resources, składnia, przypadki użycia

To specjalny rodzaj bloku try, który służy do automatycznego zamykania zasobów po zakończeniu ich użycia. Zasobem może być każdy obiekt, który implementuje interfejs java.lang.AutoCloseable lub jego podklasę java.io.Closeable

Składnia

W nawiasach po słowie kluczowym try deklarowane są zasoby, które mają być automatycznie zamknięte po zakończeniu bloku try.

```
try (ResourceType resource = new ResourceType()) {
```

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExample {
    public static String readFirstLineFromFile(String path) throws IOException {
        try (FileReader fr = new FileReader(path);
             BufferedReader br = new BufferedReader(fr)) {
            return br.readLine();
        }
    }
}
```

Przypadki użycia:

Przykład z FileReader
W przykładzie po lewej zasoby FileReader i BufferedReader są automatycznie zamykane po zakończeniu bloku try, niezależnie od tego, czy kod zakończył się pomyślnie, czy rzucił wyjątek.

Z ZipFile i BufferedWriter - w tym przykładzie zarówno ZipFile, jak i BufferedWriter są automatycznie zamykane po zakończeniu bloku try

Zalety try-with-resources:
Automatyczne zamykanie zasobów, lepsza czytelność kodu, bezpieczeństwo w przypadku wyjątków

Wymagania:
Zasoby muszą implementować interfejs **AutoCloseable** lub **Closeable**. Interfejs AutoCloseable wprowadza metodę **close**(), która jest automatycznie wywoływana po zakończeniu bloku try.

STRUMIENIE

21. Opisz trzy metody pośrednie w strumieniu.

-**filter**(predicate) - zwraca nowy strumień, który zawiera elementy spełniające określony warunek (predicate). Predicate to funkcja, która przyjmuje element i zwraca wartość typu Boolean, określając, czy dany element powinien być uwzględniony w nowym strumieniu.

-**sorted ()**– metoda zwraca nowy strumień, w którym elementy są posortowane według naturalnego porządku. Jeśli elementy strumienia nie są porównywalne, zostanie rzucony wyjątek ClassCastException podczas wykonywania operacji końcowe

-**distinct**() - Metoda distinct zwraca nowy strumień, który zawiera unikalne elementy strumienia wejściowego. Dla uporządkowanych strumieni, wybór unikalnych elementów zachowuje kolejność pojawiania się(element pojawiający się jako pierwszy zostaje zachowany)

22. Omów kolektory

Kolektory to narzędzia używane są na końcowym etapie przetwarzania strumieni. Służą do gromadzenia, przekształcania, manipulacji danymi ze strumieni w różne struktury danych, tj. Listy, zbiory, mapy.

Metoda stream.collect() umożliwia wykonanie operacji modyfikujących na elementach strumienia. Implementacja strategii dla tej operacji dostarczana jest poprzez interfejs Collector.

Przykłady kolektorów: givenList.stream().collect(toList());
Collectors.toList() - zbiera wszystkie elementy strumienia do instancji List
Collectors.toSet() - Zbiera wszystkie elementy strumienia do instancji Set

Collectors.toCollection() - pozwala na użycie niestandardowej implementacji kolekcji (toCollection(LinkedList::new))
Collectors.toMap() - zbiera elementy strumienia do instancji Map.
Collectors.collectingAndThen() - umożliwia wykonanie dodatkowej akcji na wyniku po zakończeniu zbierania elementów
Collectors.joining() - łączy elementy Stream<String>(strumienia) w jeden łańcuch znaków
Collectors.counting() - liczy wszystkie elementy strumienia

Własne kolektory:
Możemy również implementować własne kolektory, implementując interfejs Collector.

TESTOWANIE

23. Przedstaw motywacje do stosowania testów, omów Test Driven Development.

-Testy pomagają upewnić się, że kod działa bez zarzutu oraz spełnia wymagania
-Mogą pełnić rolę dokumentacji pokazując jego poprzednie wersje i jakie są ostateczne wyniki
-Posiadając dobrze przetestowane oprogramowanie, programiści mogą być pewniejsi przy wprowadzaniu aplikacji na rynek oraz mogą szybciej reagować na jego potrzeby

Test Driven Development
Jest to podejście, w którym testy są pisane przed napisaniem faktycznego kodu danej aplikacji.
Obejmuje etapy:
-Write falling test: Na tym etapie piszemy test dla funkcjonalności, która jeszcze nie została zaimplementowana. Uruchamiamy test, aby upewnić się, że nie przechodzi (jest czerwony). Jest to konieczne, aby uniknąć fałszywie pozytywnych wyników.
-**Write minimum code to pass the test** – piszemy minimalną ilość kodu, aby test przeszedł. Skupiamy się na spełnieniu warunków testu, a nie jakości kodu. Test przechodzący jest sygnalizowany na zielono.
-**Refactor** – Ostatnim etapem jest refaktoryzacja, podczas której poprawiamy jakość kodu. Możemy uruchamiać testy w trakcie refaktoryzacji, aby upewnić się, że zmiany nie wprowadziły regresji.

24. Parametryzacja testów
Parametryzacja testów pozwala na uruchomienie testu z różnymi danymi wejściowymi. Jest to pomocne, gdyż pozwala na przetestowanie różnych przypadków zmniejszając możliwość wystąpienia potencjalnych błędów

Aby używać testów sparametryowanych musimy dodać artefakt junit-jupiter-params do naszego projektu.
Aby test był sparametryzowany, musi mieć @ParametrizedTest

Źródła argumentów - testy mogą korzystać z różnych źródeł argumentów:
4.1. **proste wartości string** -> uruchamiamy test dwa razy, dla dwóch stringów

```
@ParameterizedTest
@ValueSource(strings = {"", "  "})
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

@ValueSorce(strings = {“”, “ “})

4.2 Wartości null i puste - możemy użyć @NullSource i @EmptySource do przekazywania wartości null i pustych

```
@ParameterizedTest
@NullSource
void isBlank_ShouldReturnTrueForNullInputs(String input) {
    assertTrue(Strings.isBlank(input));
}

@ParameterizedTest
@EmptySource
void isBlank_ShouldReturnTrueForEmptyStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

4.3 Enum - użycie @EnumSource do testowania wartości z wyliczeń

4.4 CSV Literals -> @CsvSource do przekazywania wielu argumentów

4.5 Pliki CSV -> @CsvFileSource

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1)
void toUpperCase_ShouldGenerateTheExpectedUppercaseValueCSVFile(String input, String ex
    String actualValue = input.toUpperCase();
    assertEquals(expected, actualValue);
}
```

4.6 Metoda - użycie metody do dostarczania argumentów za pomocą @MethodSource

```
@ParameterizedTest
@MethodSource("provideStringsForIsBlank")
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input, boolean expected) {
    assertEquals(expected, Strings.isBlank(input));
}

private static Stream<Arguments> provideStringsForIsBlank() {
    return Stream.of(
      Arguments.of(null, true),
      Arguments.of("", true),
      Arguments.of("  ", true),
      Arguments.of("not blank", false)
    );
}
```

4.7 Pole - użycie pola jako źródła argumentów za pomocą @FieldSource

```
static List<String> cities = Arrays.asList("Madrid", "Rome", "Paris", "London");

@ParameterizedTest
@FieldSource("cities")
void isBlank_ShouldReturnFalseWhenTheArgHasAtLEastOneCharacter(String arg) {
    assertFalse(Strings.isBlank(arg));
}
```

4.8 Własny dostawca argumentów - za pomocą interfejsu ArgumentsProvider

```
class BlankStringsArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of(
          Arguments.of((String) null),
          Arguments.of(""),
          Arguments.of("   ")
        );
    }
}

@ParameterizedTest
@ArgumentsSource(BlankStringsArgumentsProvider.class)
void isBlank_ShouldReturnTrueForNullOrBlankStringsArgProvider(String input) {
    assertTrue(Strings.isBlank(input));
}
```

Powtarzalne Adnotacje źródeł argumentów: Można wielokrotnie używać tych samych adnotacji źródeł argumentów:

```
@ParameterizedTest
@MethodSource("asia")
@MethodSource("europe")
void whenStringIsLargerThanThreeCharacters_thenReturnTrue(String country) {
    assertTrue(country.length() > 3);
}
```

Można tutaj też personalizować nazwy wyświetlane, za pomocą @ParametrizedTest

```
@ParameterizedTest(name = "{index} {0} is 30 days long")
@EnumSource(value = Month.class, names = {"APRIL
```

Napisać coś więcej o tym? Chyba olać..

PAKIETY

25. Zasady tworzenia i zastosowanie pakietów

Tworzenie:

Aby utworzyć pakiet należy wybrać nazwę dla pakietu, umieścić instrukcję ”package nazwa” na początku każdego pliku źródłowego, który ma być częścią tego pakietu. Każdy plik źródłowy może zawierać tylko jedną instrukcję package i powinna być ona pierwszą linią w pliku źródłowym. Jeśli plik zawiera kilka typów, tylko jeden może być publiczny i jego nazwa musi być taka sama jak nazwa pliku. (?)

Nazewnictwo pakietów - muszą być pisane małymi literami

Używanie członków pakietów - aby użyć publicznego członka pakietu spoza jego pakietu można odwołać się do niego za pomocą jego w pełni kwalifikowanej nazwy, zaimportować członka pakietu, zaimportować cały pakiet

Odwoływanie się do członka pakietu za pomocą wpełni kwalifikowanej nazwy:

Jeśli pakiet graphics zawiera klasę Rectangle, można do niej odwołać się w następujący sposób:

```
graphics.Rectangle myRect = new graphics.Rectangle();
```

Importowanie członka pakietu:

```
import graphics.Rectangle;
Rectangle myRectangle = new Rectangle();
```

Importowanie całego pakietu:

```
import graphics.*;
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

Zastosowanie:

-Pakiety pomagają organizować kod w logiczne grupy,

-Dzięki pakietom mogą istnieć klasy o tej samej nazwie, dzięki nim nie ma konfliktów nazw,

-Pakiety umożliwiają kontrolowanie widoczności klas

26. Czym są pliki JAR? Omów ich tworzenie i wykorzystanie. Omów manifest.

Czym są pliki JAR? (Java ARchive) - to format plików archiwum używany do agregacji wielu plików klasy Java i zasobów (takich jak obrazy, dźwięki, metadane) w jednym pliku dla łatwiejszej dystrybucji i implementacji. Pliki JAR są używane do kompresji i organizacji bibliotek, aplikacji i apletów Java.

Najprościej: Pliki JAR są plikami archiwum, które zawierają pliki klas i bibliotek skompresowanych w jeden plik

Tworzenie:

Na samym początku trzeba skompilować pliki źródłowe .java do plików .class za pomocą kompilatora javac
javac MyClass1.java MycClass2.java

Utworzenie pliku JAR:

-Tworzy się za pomocą polecenia “jar cf nazwa_pliku.jar MyClass1.class MyClass2.class” gdzie “c” oznacza

create, a “f” file ->określ nazwę pliku archiwum.

Dodanie katalogu: Możesz również dodać cały katalog do pliku JAR:
jar cf myarchieve.jar -C path/to/classes .

Tutaj -C zmienia katalog na path/to/classes, a . Wskazuje, że wszystkie pliki i podkatalogi w bieżącym katalogu mają zostać dodane do pliku JAR.

Wykorzystanie plików JAR:

-Biblioteki – pakietowanie bibliotek, które mogą być używane przez inne aplikacje,
-Aplety – dystrybucja aplikacji za pomocą technologii
-Dystrybucja kompletnych aplikacji, często zawierających manifest określający główną klasę, którą JVM ma uruchomić
-Java Web Start: Dystrybucja aplikacji za pomocą technologii Java Web Start.

Manifest – to specjalny plik metadanych, który może być zawarty w pliku JAR, aby dostarczyć dodatkowe informacje o zawartości archiwum. Domyślny manifest jest umieszczony w pliku META-INF/MANIFEST.MF

Jego zawartość: Manifest jest tekstowym plikiem, który może zawierać różne atrybuty.
Przykładowo:
Main-Class - określa główną klasę, która zawiera metodę main do uruchomienia aplikacji,
Class-Path - określa dodatkowe pliki JAR, które są potrzebne do uruchomienia aplikacji

Tworzenie manifestu

Można utworzyć niestandardowy plik manifestu i dodać go do pliku JAR.
1)Stwórz plik manifest.txt zawierające żądane atrybuty:
Main-Class: com.example.MyMainClass
2)Użyj opcji m z poleceniem jar, aby dodać manifest do pliku JAR
jar cfm myarchive.jar manifest.txt -C path/to/classes .

Opcje: m- dodaj manifest do pliku JAR

27. Wymień i omów trzy zastosowania Mavena

1)Automatyzacja budowy oprogramowania:

- **Pobieranie zależności:** Automatyczne pobieranie bibliotek potrzebnych do projektu.
- **Kompilacja kodu:** Konwersja kodu źródłowego na kod binarny.
- **Testowanie:** Automatyczne uruchamianie testów, by sprawdzić poprawność kodu.
- **Pakowanie:** Tworzenie plików JAR, WAR lub ZIP do wdrożenia.
- **Wdrażanie:** Umieszczanie gotowego oprogramowania na serwerach.

Dzięki temu proces budowy jest szybki, spójny i mniej podatny na błędy.

2)Zarządzanie zależnościami:

- **Automatyczne pobieranie:** Maven sam pobiera potrzebne biblioteki i ich aktualizacje.
- **Transitive dependencies:** Maven automatycznie pobiera zależności zależności.
- **Izolacja zależności:** Oddziela zależności projektów i wtyczek, zmniejszając ryzyko konfliktów.

To sprawia, że zarządzanie bibliotekami jest łatwiejsze i bardziej efektywne.

3)Zarządzanie konfiguracją projektu poprzez POM (Project Object Model):

- **Identyfikatory projektu:** groupId, artifactId, version – unikalne identyfikatory projektu.
- **Zarządzanie zależnościami:** Deklarowanie i zarządzanie bibliotekami.
- **Konfiguracja wtyczek:** Określanie używanych wtyczek i ich ustawień.
- **Definicja profili:** Tworzenie różnych ustawień dla różnych środowisk (np. produkcyjnych, testowych).

POM pomaga utrzymać porządek w projekcie i ułatwia zarządzanie konfiguracją.

28. Elementy skłądowe pliku pom.xml

Pom.xml to plik XML, który zawiera informacje o projekcie oraz szczegóły konfiguracyjne używane przez Mavena do budowania projektu

Minimalny POM:

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

-project - korzeń dokumentu
-modelVersion – powinno być ustawione na 4.00
-groupId – identyfikator grupy projektu
-artifactId – identyfikator artefaktu
-version – wersja artefaktu w ramach określonej grupy

PROGRAMOWANIE GENERYCZNE

29. Programowanie generyczne, a typ Object

Typy generyczne w Javie pozwalają na tworzenie klas, interfejsów i metod, które mogą operować na różnych typach danych, zdefiniowanych w momencie tworzenia obiektu lub wywołania metody. W ten sposób kod staje się bardziej elastyczny i bezpieczny, ponieważ typy są sprawdzane na etapie kompilacji.

Przykład niegenerycznej klasy Box:

```java
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Ponieważ metody przyjmują lub zwracają Object, można przekazać dowolny obiekt, co może prowadzić do błędów w czasie wykonywania programu, jeśli jeden kod oczekuje Integer, a inny przekazuje String.

```java
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Wersja generyczna klasy Box:

Wszystkie wystąpienia Object zostały zastąpione przez T, co oznacza typ, który zostanie określony podczas tworzenia obiektu Box. Dzięki temu typ jest znany i sprawdzany na etapie kompilacji, co zapobiega błędom.

Typ Object:

W Javie Object jest klasą bazową dla wszystkich innych klas. Oznacza to, że każda klasa dziedziczy bezpośrednio lub pośrednio po Object.
Ponieważ Object jest nadrzędnym typem dla wszystkich innych typów, można przechowywać dowolny obiekt w zmiennej typu Object. Używanie Object nie zapewnia typowania w czasie kompilacji, co może prowadzić do błędów w czasie wykonywania programu. Przykładowo, rzutowanie obiektu na niewłaściwy typ może spowodować wyjątek ClassCastException.

Porównianie typów generycznych i Object:

1) Bezpieczeństwo typów:

-Object – przechowywanie wartości w zmiennej typu Object wymaga rzutowania przy odczycie, co może prowadzić do błędów w czasie wykonywania programu,
-Typy generyczne - używanie typów generycznych eliminuje konieczność rzutowania, ponieważ typy są jasno określone i jawne

2) Czytelność kodu:

-Object – kod może być mniej czytelny i trudniejszy do zrozumienia, ponieważ typ obiektu nie jest jawnie określony
-Typy generyczne – kod z typami generycznymi jest bardziej czytelny, ponieważ są jasno określone i jawne

3) Elastyczność:

-Object - umożliwia przechowywanie dowolnych obiektów, ale kosztem bezpieczeństwa typów,
-Typy generyczne - zapewniają elastyczność przy jednoczesnym zachowaniu bezpieczeństwa typów

30. Ograniczenia typów generycznych

-dziedziczenie -> Typy generyczne nie dziedziczą automatycznie po sobie, np. List<Object> nie jest supertypem List<String>, trzeba o tym pamiętać, aby to obejść Java wprowadza tzw. Dzikie karty (wildcards)

Wildcards(dzikie karty) - pozwalają na określenie, że parametr typu może być dowolnym typem, lub że jest ograniczony do pewnych typów, np. Collection<?> lub List<? extends Shape> lub List<? Super Shape>

-Nie można tworzyć statycznych członków klas z parametrami typów
-Rzutowanie generycznych typów może być problematyczne i powoduje błędy
-Brak instancji typów generycznych, np. Nie można napisać new T(), bo T nie jest znane w czasie generacji

31. Interfejsy funkcyjne: Supplier, Predicate, Consumer,Function

Interfejsy funkcyjne są kluczowe dla korzystania z wyrażeń lambda i referencji do metod, umożliwiając bardziej zwięzłe i czytelne kodowanie.

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Supplier – dostarcza wynik bez żadnych parametrów wejściowych. Jest to typowy “dostawca” wartości.

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Predicate<T> - jest to interfejs reprezentujący funkcję, która przyjmuje jeden argument i zwraca wartość boolean. Jest to typowy interfejs używany do testowania warunków

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

Consumer<T> - reprezentuje operację, która przyjmuje jeden argument i nie zwraca żadnego wyniku Jest to typowy interfejs używany do operacji, które wykonują akcje ale nie zwracają wyników.

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Function<T,R> - reprezentuje funkcję, która przyjmuje jeden argument i zwraca wynik. Jest to typowy interfejs używany do mapowania i przekształcania wartości.

32. Interfejsy funkcyjne, a funkcje lambda, implementacja własnego interfejsu

Interfejsy funkcyjne a funkcje Lambda:

Interfejsy funkcyjne – to interfejsy, które zawierają tylko jedną metodę abstrakcyjną. Mogą jednak zawierać też metody statyczne i domyślne. Interfejsy funkcyjne są kluczowe w kontekście wyrażeń lambda, ponieważ pozwalają na przekazywanie funkcjonalności jako argumentu metody lub traktowanie kodu jako danych

Przykładowy interfejs funkcyjny:

```java
@FunctionalInterface
interface CheckPerson {
    boolean test(Person p);
}
```

```java
List<Person> roster = ... // lista osób
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
                  && p.getAge() >= 18
                  && p.getAge() <= 25
);
```

Funkcje lambda - są bardziej zwięzłym sposobem wyrażania instancji interfejsów funkcyjnych. Pozwalają na tworzenie krótkich bloków kodu, które można przekazać jako argumenty, co upraszcza kod i poprawia jego czytelność.

Implementacja własnego interfejsu funkcyjnego

Możemy stworzyć własny interfejs funkcyjny i użyć go w połaczeniu z wyrażeniami lamvda. Oto przykład:

Aby stworzyć własny interfejs, musimy dodać adnotację @FunctionalInterface, po czym stworzyć metodę abstrakcyjną.

33. Omów szablon Optional

Klasa “Optional” wproadzona została w Java 8. Umożliwia reprezentowanie opcjonalnych wartości zamiast “null”.

Tworzenie obiektów Optional:

Optional<string> empty = Optional.empty();

Tworzenie Optional z wartością:
Optional<String> opt=Optional.of(“wartosc”);

Tworzenie Optional z możliwością null:
Optional <String> opt = Optional.ofNullable(możliwaNull);

Sprawdzanie obecności wartości za pomocą dwóch metod:
isPresent(), isEmpty()

ifPresent() ->Jeżeli wartość jest obecna, to wykonuje akcje
Wartość domyślna z orElse() - zwraca wartość, jeśli jest obecna, jeśli nie, to wartość domyślną

```
String wynik = Optional.ofNullable(null).orElse("domyślna");
```

String result = optional.orElse("default")

Różnica między orElse() i orElseGet()
orElse() - dostarczyciel wartości jest zawsze wykonywany,
orElseGet() - dostarczyciel wartości jest wykonywany tylko, gdy wartość jest nieobecna

```
String wynik = Optional.ofNullable(null).orElseGet(() -> "domyślna");
```

Wyjątki z orElseThrow() - rzuca wyjątek, jeśli wartość jest nieobecna
optional.get() - zwraca wartość, w przeciwnym razie rzuca NoSuchException

Filtriowanie z filter() -> filter() zwraca Optional, jeśli wartość spełnia predykat
Transformacja wartości z map() -> map() przekształca wartość wewnątrz Optional
Zagnieżdżone Optional z flatMap() -> flatMap() przekształca wartość wewnątrz zagnieżdżonego Optional

```
Optional<String> result = Stream.of(opt1, opt2, opt3)
                              .filter(Optional::isPresent)
                              .map(Optional::get)
                              .findFirst();
```

Łączenie Optionali -> użyj stream do łączenia kilku Optional i pobierania pierwszej niepustej wartości

Optional jest przeznaczony do użycia jako typ zwracany, a nie jako parametr metody, pamiętaj. Trzeba również unikać używania Optional jako pola w serializowanej klasie.

Optional pomaga w unikaniu NullPointerException

WĄTKI

34. Tworzenie wątku, charakterystyka i porównanie Thread i Runnable

Tworzenie wątku
W Javie istnieją dwa główne podejścia do tworzenie wątków:

-przez dziedziczenie klasy **Thread**(rozszerzamy tą klasę, przysłaniamy metodę run(), która zawiera kod, który ma być wykonywany w nowym wątku)
-przez implementację interfejsu **Runnable**(tworzymy klasę implementującą Runnable, przekazujemy tę instancję do konstruktora klasy Thread. Kod do wykonania umieszczamy w metodzie run() interfejsu Runnable)

Charakterystyka:
1) Thread – klasa w Javie która reprezentuje wątek, dziedziczy po klasie Object, może być rozszerzona przez inne klasy, każdy obiekt Thread ma unikalne ID, wywołanie start() tworzy nowy wątek i uruchamia metodę run()

2) Runnable – To interfejs funkcyjny z jedną metodą abstrakcyjną run(). Może być implementowany przez dowolną klasę, pozwala na oddzielenie kodu do wykonania od mechanizmu zarządzania wątkami, może być używany w wielu wątkach, dzieląc tę samą instancję

Porównanie:
Thread – prostszy w użyciu, nie wymaga osobnej klasy implementującej interfejs, może być używany bezpośrednio, bez konieczności tworzenia dodatkowej instancji, ogranicza możliwość dziedziczenia innych klas, tworzy nowy obiekt Thread dla każdego wątku

Runnable - Elastyczność i oddzielenie kodu od mechanizmu wątków, może być używany w wielu wątkach, dzieląc tę samą instancję, wymaga osobnej klasy implementującej interfej

35. Omów synchronizację wątków

Pozwala na kontrolowanie dostępu do współdzielonych zasobów, tj. Zmienne, obiekty, metody. Mechanizmy synchronizacji wątków:

synchronized - może być używane na poziomie metody/bloku kodu, chroni sekcje krytyczne, umożliwiając tylko jednemu wątkowi dostęp w danym momencie

locki – Klasa ReentrantLock umożliwia bardziej zaawansowaną synchronizację, pozwala na zarządzanie blokadami. Możemy użyć lock () i unlock() w odpowiednich miejscach.

semafory – klasa Semaphore pozwala na kontrolowanie dostępu do zasobów. Określamy maksymalną liczbę wątków, które mogą korzystać z danego zasobu jednocześnie.

Latche – klasa CountDownLatch pozwala na oczekiwanie na zakończenie określonej liczby wątków
Barriery –klasa CyclicBarrier pozwala na grupowanie wątków, wszystkie wątki czekają na siebie nawzajem, a potem wykonują określoną operację

36. Omów cykl życia wątku

1. Nowość
Na tym etapie wątek jest utworzony ale jeszcze nie został uruchomiony, tworzymy go przy pomocy klasy Thred, pozostaje w tym stanie oczekując na uruchomienie. Nazywany jest również “utworzonym wątkiem”

2. Możliwość uruchomienia (Runnable)
Wątek przechodzi w ten stan po wywołaniu metody start(), kontrola nad wątkiem jest przekazywana programowi planującemu, który decyduje kiedy wątek zostanie wykonany. W tym stanie wątek może wykonywać kod

3. Bieganie (Running)
Wątek znajduje się w tym stanie, gdy jest aktywnie wykonywany na procesorze, wykonuje kod zawarty w metodzie run ()

4. Czekanie (Waiting)
Wątek przechodzi w ten stan gdy oczekuje na pewne zdarzenie lub zasób, może to być oczekiwanie na zakończenie innego wątku, na sygnał od innego wątku lub dostęp do zasobu, nie zużywa w tym czasie zasobów procesora.

5. Nie żyje (Terminated):
Wątek kończy swoje działanie, gdy wykonanie metody run() zostaje zakończone. Może to być wynik normalnego zakończenia lub wystąpienia wyjątku.

JAVAFX

37. Porównanie frameworków do tworzenia interfejsu użytkownika

Porównywanie do tworzenia interfejsów użytkownika w Javie obejmuje Java AWT, Java Swing i JavaFX. Każdy z tych frameworków oferuje różne unikalne funkcje, komponenty i mozliwości

Java AWT (Abstract Window Toolkit)
-Komponenty: Przycisk(Button), Pole tekstowe(TextField),Pole wyboru (CheckBox)
-Umożliwia użytkownikom klikanie, wprowadzanie tekstu, wybieranie opcji
-Obsługa złożonych elementów -> Menu, okna dialogowe, okna

Java Swing
-Komponenty: Przycisk (JButton), Pole tekstowe (JTextField), Lista (JFlist), Tabela (JTable)
-Komponenty są w pełni konfigurowalne, oferując spójny wygląd na różnych platformach
-Obsługa przeciągania i upuszczania, cofania/ponawiania, stylizacji
-Oddziela model danych, interfejs użytkownika i logikę sterowania

JavaFX
-Tabele,wykresy,drzewa
-Płynne animacje, efekty wizualne,
-Wideo, dźwięk, grafika 3D,
-Obsługa na komputerach, w przeglądarkach i urządzeniach mobilnych
-Stylizacja za pomocą CSS, definicja układów z FXML

Porównanie wydajności:
-Java AWT – lekki i odpowiedni do prostych aplikacji ale różnice w wydajności na różnych platformach,
-Java Swing – optymalizowany pod kątem wydajności, odpowiedni dla aplikacji o średniej i dużej złożoności
-JavaFX – wykorzystuje akceleracje sprzętową, doskonała wydajność dla nowoczesnych interfejsów użytkownika

Bogactwo komponentów interfejsu użytkownika:
-Java AWT – podstawowy zestaw komponentów, ograniczone możliwości
-Java Swing – bogata biblioteka komponentów, szeroka gama interaktywnych elementów,
-JavaFX – nowoczesny i rozbudowany zestaw komponentów, zaawansowane możliwości multimedialne

Łatwość obsługi i krzywa uczenia się:

-AWT - łatwy i prosty do nauki, odpowiedni dla początkujących
-Swing – bogaty zestaw funkcji, wymaga bardziej stromej krzywej nauki
-JavaFX – nowoczesne funkcje mogą być bardziej złożone ale oferują bardziej elastyczne środowisko programistyczne

Zgodność z nowoczesnymi standardami Java:
-AWT – kompatybilny z nowoczesnymi standardami, ale mniej bogaty w funkcje,
Swing – zaktualizowany do pracy z nowymi wersjami Java, nadal szeroko stosowany
-JavaFX – nowoczesny zestaw narzędzi, dobrze zintegrowany z nowoczesnymi standardami Java

Zgodność platformy
Java AWT - głównie aplikacje komputerowe, ograniczona obsługa internetowa i mobilna,
Java Swing - głównie aplikacje komputerowe, ograniczone wsparcie dla aplikacji internetowych i mobilnych
JavaFX – wieloplatformowy, obsługuje komputery, internet i urządzenia mobilne

Tabela porównawcza: Java AWT, Java Swing i JavaFX

Kryterium	Java AWT	Java Swing	JavaFX
Wydajność	Lekka, różnice między platformami	Zoptymalizowana, średnia/duża złożoność	Akceleracja sprzętowa, doskonała wydajność
Komponenty UI	Podstawowe	Bogata biblioteka	Nowoczesne, rozbudowane
Łatwość obsługi	Prosty, łatwy	Bogaty zestaw funkcji, bardziej złożony	Nowoczesny, elastyczny
Zgodność z nowoczesnymi standardami	Kompatybilny, mniej funkcji	Kompatybilny, szeroko stosowany	Nowoczesny, dobrze zintegrowany
Wsparcie społeczności	Duża społeczność, mniej rozwijany	Silna społeczność, aktywny rozwój	Rosnąca społeczność, aktywnie utrzymywany
Zgodność platformy	Aplikacje komputerowe, ograniczone wsparcie internetowe/mobilne	Aplikacje komputerowe, ograniczone wsparcie internetowe/mobilne	Wieloplatformowy, PC/internet/mobilne
Fragmenty kodu	Prosty kod	Bogaty zestaw komponentów	Nowoczesne komponenty, FXML

38. Czym są wydarzenia i jak je obsługiwać w JavaFX?

Wydarzenie to akcja, która zachodzi, gdy użytkownik wchodzi w interakcję z aplikacją, np. Kliknięcie przycisku, przesunięcie myszki, wpisanie tekstu na klawiaturze czy przewinięcie strony. Możemy te wydarzenia podzielić na dwa typy:
-Pierwszoplanowe - wymagają bezpośredniej interakcji użytkownika, np. Kliknięcie przycisku, ruch myszką, naciśnięcie klawisza
-drugoplanowe – nie wymagają interakcji użytkownika, np. Przerwanie systemu operacyjnego, awarie sprzętu lub oprogramowania, wygaśnięcie timera, zakończenie operacji

Obsługa wydarzeń w JavaFX:
JavaFX zapewnia wsparcie dla szerokiej gamy wdarzeń. Klasa Event z pakietu javafx.event jest bazową

klasą dla wszystkich wydarzeń w JavaFX. Przykłady to:
wydarzenia myszy -> MouseEvent, wydarzenia klawiatury -> KeyEvent, wydarzenia przeciągania -> DragEvent, wydarzenia okna -> WindowEvent

W JavaFX każde wydarzenie ma:
->**Target** - węzeł na którym zdarzenie zaszło (np. Przycisk),
->**Source** - źródło, z którego wydarzenie zostało wygenerowane (Myszka)
->**type** – typ wydarzenia (np. Kliknięcie myszką, naciśnięcie klawisza)

Fazy obsługi wydarzeń w JavaFX:
1) Budowanie ścieżki wydarzenia: Po wygenerowaniu wydarzenia, tworzona jest ścieżka dystrybucji wydarzenia od etapu do źródłowego węzła.
2)**Faza przechwytywania wydarzenia** – po utworzeniu ścieżki dystrybucji wydarzenia, jest ono dystrybuowane od węzła korzenia do węzła docelowego (z góry na dół). Jeśli którykolwiek z węzłów na tej ścieżce ma zarejestrowany filtr dla danego wydarzenia, zostanie on wykonany.
3) **Faza bąblowania wydarzenia** – w tej fazie wydarzenie jest dystrybuowane od węzła docelowego do węzła korzenia(z dołu do góry). Jeśli którykolwiek z węzłów na tej ścieżce ma zarejestrowaną obsługę wydarzenia, zostanie ona wykonana.
Obsługi i filtry wydarzeń zawierają logikę aplikacji do przetwarzania wydarzeń. Węzeł może zarejestrować więcej niż jedną obsługę/filtr. W przypadku węzłów rodzic-dziecko można dostarczyć wspólny filtr/obsługę dla rodzica, który będzie domyślnie przetwarzał wydarzenia dla wszystkich węzłów potomnych.
Wszystkie obsługi i filtry implementują interfejs EventHandler z pakietu javafx.event

39. Omów zależność między typami Scene i Stage w JavaFX

Są to kluczowe elementy do tworzenia aplikacji graficznych.

Klasa Scene reprezentuje zawartość wizualną aplikacji i jest kontenerem dla wszystkich elementów w grafie sceny. Oto kilka głównych cech klasy Scene:
-Scene przechowuje węzły aplikacji, które tworzą graf sceny
-Aplikacja musi ustawić główny węzeł grafu sceny, może to być grupa lub węzeł resizable
-Rozmiar sceny może być ustawiony podczas jej tworzenia. Jeśli scena jest resizable, jej rozmiar dostosowuje się do rozmiaru głównego węzła.
-Scena może wspierać bufor głębokości i wygładzanie krawędzi, co jest istotne przy renderowaniu obiektów 3D,
-Obiekty klasy Scene muszą być tworzone i modyfikowane na wątku aplikacji JavaFX.

Klasa Stage – jest najwyższym kontenerem w JavaFX, który reprezentuje okno aplikacji. Każda aplikacja JavaFX ma przynajmniej jeden główny stage, który jest tworzony przez platformę. Można tworzyć dodatkowe stage'y.
-Stage może mieć różne style, jak DECORATED, UNDECORATED, TRANSPARENT i UTILITY. Styl musi być ustawiony przez pokazaniem okna.
-Stage może mieć opcjonalnego właściciela (owner), który jest innym oknem. Kiedy właściciel jest zamknięty, wszystkie potomne okna rónież są zamknięte.
-Stage może być modalny, co oznacza, że blokuje interakcje z innymi oknami.
-Obiekty klasy Stage muszą być tworzone i modyfikowane na wątku aplikacji JavaFX.

Relacja pomiędzy Scene i Stage

Przypisanie sceny do stage: Scene jest przypisywana do Stage za pomocą metody setScene. Stage jest kontenerem okna, a Scene jego zawartością wizualną.
Rozmiar i layout – rozmiar Stage wpływa na rozmiar Scene. Jeśli Stage zmienia rozmiar, Scene również może się zmienić, co wpływa na layout grafu sceny.
Renderowanie: Stage renderuje zawartość Scene na ekranie
Interakcje użytkownika: Wydarzenia użytkownika, tj. Kliknięcie myszą czy naciskanie klawiszy są przetwarzane przez Scene, a następnie propagowane do odpowiednich węzłów.

Podsumowując:

Stage jest oknem aplikacji, a Scene jest jej zawartością. Scene zawiera wszystkie węzły aplikacji, które są renderowane i interakcje użytkownika są przetwarzane. Stage zarządza wyświetlaniem Scene i interakcjami z systemem operacyjnym.