Wyższa Szkoła Bankowa
we Wrocławiu

Programowanie aplikacji w Java

Maciej Gowin

Zjazd 3 - dzień 1

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Język Java: klasa Object

Dotychczas wartości (prymitywne i obiekty typu `String`) porównywaliśmy przy pomocy operatora `==`.

W przypadku obiektów sprawa jest bardziej skomplikowana, gdyż w zmiennej przechowywana jest jedynie referencja do obiektu, a nie jego wartość. Użycie operatora `==` może przynieść nieoczekiwane efekty.

Programowanie: przykład 25

Porównanie obiektów przy pomocy operatora `==`.

```
class Container {
    private String value;

    public Container(String value) {
        this.value = value;
    }
}

public class Main {

    public static void main(String[] args) {
        Container a1 = new Container("test");
        Container a2 = new Container("test");
        Container a3 = new Container("another");
        Container a4 = a1;

        System.out.printf("a1 == a2: %b%n", a1 == a2);
        System.out.printf("a1 == a3: %b%n", a1 == a3);
        System.out.printf("a1 == a4: %b%n", a1 == a4);
    }
}
```

Język Java: klasa Object

Porównanie dwóch na pozór identycznych obiektów przy pomocy operatora `==` powoduje porównanie referencji.

```
Container a1 = new Container("test");
Container a2 = new Container("test");
boolean compare = a1 == a2;
```

Hierarchia dziedziczenia wprowadza klasę `Object`, po której dziedziczą wszystkie inne klasy. Definiuje one metodę `boolean equals(Object)`, która powinna być używana do porównywania obiektów.

Programowanie: przykład 26

Porównanie obiektów przy pomocy metody `equals`.

```
class Container {
    private String value;

    public Container(String value) {
        this.value = value;
    }
}

public class Main {

    public static void main(String[] args) {
        Container a1 = new Container("test");
        Container a2 = new Container("test");
        Container a3 = new Container("another");
        Container a4 = a1;

        System.out.printf("a1 == a2: %b%n", a1 == a2);
        System.out.printf("a1 == a3: %b%n", a1 == a3);
        System.out.printf("a1 == a4: %b%n", a1 == a4);

        System.out.printf("a1 equals a2: %b%n", a1.equals(a2));
        System.out.printf("a1 equals a3: %b%n", a1.equals(a3));
        System.out.printf("a1 equals a4: %b%n", a1.equals(a4));
    }
}
```

Język Java: metoda Object.equals()

Użycie metody `equals` nie przyniosło pożądanych efektów. Jest to spowodowane, domyślną implementacją metody w klasie `Object`. Aby porównanie zadziałało poprawnie, musi ono zostać zdefiniowane explicite, a metoda nadpisana.

```
Container a1 = new Container("test");
Container a2 = new Container("test");
boolean compare = a1.equals(a2);
```

Programowanie: przykład 27

Porównanie obiektów przy pomocy poprawnie nadpisanej metody `equals`.

```
class Container {
    private String value;

    public Container(String value) {
        this.value = value;
    }

    public boolean equals(final Object o) {
        if (o == this)  {
            return true;
        }
        if (!(o instanceof Container)) {
            return false;
        }

        Container other = (Container) o;

        return (this.value == null && other.value == null)
                || (this.value != null && this.value.equals(other.value));
    }
}

public class Main {

    public static void main(String[] args) {
        Container a1 = new Container("test");
        Container a2 = new Container("test");
        Container a3 = new Container("another");
        Container a4 = a1;

        System.out.printf("a1 == a2: %b%n", a1 == a2);
        System.out.printf("a1 == a3: %b%n", a1 == a3);
        System.out.printf("a1 == a4: %b%n", a1 == a4);
        System.out.printf("a1 == null: %b%n", a1 == null);

        System.out.printf("a1 equals a2: %b%n", a1.equals(a2));
        System.out.printf("a1 equals a3: %b%n", a1.equals(a3));
        System.out.printf("a1 equals a4: %b%n", a1.equals(a4));
        System.out.printf("a1 equals null: %b%n", a1.equals(null));
    }
}
```

Język Java: klasa String i equals()

Do tej pory podczas porównywania obiektów typu `String` używaliśmy operatora `==`.

```
String s1 = "test";
String s2 = "test";
boolea isEqual = s1 == 2;
```

Porównania te działały zgodnie z oczekiwaniami, ponieważ do zmiennych przypisywane były literały.

Zmienne wskazywały zatem tę samą przestrzeń w pamięci. Jest to związane ze współdzieloną przestrzenią w pamięci zwaną `String pool`, w której tworzone są niektóre obiekty typu `String`.

Do porównywania ciągów znaków podobnie jak innych obiektów będziemy używać metody `equals` gdyż nie zawsze są one inicjalizowane przy pomocy literałów.

Programowanie: przykład 28

Porównanie ciągu znaków.

```
public class Main {

    public static void main(String[] args) {
        String s1 = "test";
        String s2 = "test";
        String s3 = "another";
        String s4 = new String("test");
        String s5 = new String("another");
        String s6 = s1;

        System.out.printf("s1 == s2: %b%n", s1 == s2);
        System.out.printf("s1 == s3: %b%n", s1 == s3);
        System.out.printf("s1 == s4: %b%n", s1 == s4);
        System.out.printf("s1 == s5: %b%n", s1 == s5);
        System.out.printf("s1 == s6: %b%n", s1 == s6);

        System.out.printf("s1 equals s2: %b%n", s1.equals(s2));
        System.out.printf("s1 equals s3: %b%n", s1.equals(s3));
        System.out.printf("s1 equals s4: %b%n", s1.equals(s4));
        System.out.printf("s1 equals s5: %b%n", s1.equals(s5));
        System.out.printf("s1 equals s6: %b%n", s1.equals(s6));
    }
}
```

Język Java: kontrakt Object.equals()

Java definiuje kontrakt, który nasza implementacja metody `equals()` powinna spełniać. Większość z założeń to dobre praktyki i nie są one w żaden sposób wymagane przez język.

Metoda `equals()` powinna być:

- zwrotna: obiekt musi być równy samemu sobie
- symetryczna: `x.equals(y)` powinno zwrócić tę samą wartość co `y.equals(x)`
- przechodnia: jeżeli `x.equals(y)` oraz `y.equals(z)` to wtedy również `x.equals(z)`
- spójna: wartość `equals()` powinna ulec zmianie, tylko jeżeli własność użyta w `equals()` uległa zmianie (brak losowości)

Język Java: metoda Object.hashCode()

Klasa obiekt wprowadza jeszcze drugą ważną metodę `hashCode()` powiązaną z metodą `equals()`.

`hashCode()` zwraca wartość całkowitą reprezentującą daną instancję klasy. Wartość ta powinna zostać obliczona na podstawie własności obiektu oraz być spójna z definicją metody `equals()`.

> Jeżeli nadpisujemy metodę `equals()`, nadpisana powinna też zostać metoda `hashCode()`.

Metoda ta jest de facto funkcją skrótu. Przyporządkowuje ona obiektowi krótką wartość o stałym rozmiarze, która jest nieodwracalna.

Język Java: kontrakt Object.hashCode()

Java definiuje kontrakt, który nasza implementacja metody `hashCode()` powinna spełniać. Większość z założeń to dobre praktyki, które powiązane są z metodą `equals()`.

Z metodą `hashCode()` związane są kryteria:

- spójność wewnętrzna: wartość `hashCode()` może ulec zmianie jedynie, jeżeli zmianie ulegnie wartość użyta w metodzie `equals()`
- spójność z `equals()`: obiekty równe sobie muszą zwrócić tę samą wartość dla `hashCode()`
- kolizyjność: nierówne obiekty mogą zwrócić tę samą wartość dla `hashCode()`

Programowanie: przykład 29

Implementacje metody `hashCode()`.

```
public class Main {

    public static void main(String[] args) {
        System.out.println("--- Standard");
        System.out.println("1, foo: " + new Standard(1, "foo").hashCode());
        System.out.println("1, foo: " + new Standard(1, "foo").hashCode());
        System.out.println("1, bar: " + new Standard(1, "bar").hashCode());
        System.out.println("2, foo: " + new Standard(2, "foo").hashCode());
        System.out.println("2, bar: " + new Standard(2, "bar").hashCode());

        System.out.println("--- ObjectsBased");
        System.out.println("1, foo: " + new ObjectsBased(1, "foo").hashCode());
        System.out.println("1, foo: " + new ObjectsBased(1, "foo").hashCode());
        System.out.println("1, bar: " + new ObjectsBased(1, "bar").hashCode());
        System.out.println("2, foo: " + new ObjectsBased(2, "foo").hashCode());
        System.out.println("2, bar: " + new ObjectsBased(2, "bar").hashCode());

        System.out.println("--- IntellijGenerated");
        System.out.println("1, foo: " + new IntellijGenerated(1, "foo").hashCode());
        System.out.println("1, foo: " + new IntellijGenerated(1, "foo").hashCode());
        System.out.println("1, bar: " + new IntellijGenerated(1, "bar").hashCode());
        System.out.println("2, foo: " + new IntellijGenerated(2, "foo").hashCode());
        System.out.println("2, bar: " + new IntellijGenerated(2, "bar").hashCode());
    }
}
```

Język Java: istota equals i hashCode

Metody `equals()` oraz `hashCode()` są wykorzystywane przez wiele bibliotek, które zależą od ich poprawnych implementacji.

Przykładem mogą być wbudowane kolekcje Java:

- Set: metoda `equals()` jest używana do porównania czy dana wartość już istnieje w zbiorze.
- HashMap: metoda `hashCode()` jest używana do definicji kubełka, do którego dodawany jest obiekt.

Język Java: typy generyczne

Do tej pory tworzyliśmy struktury danych przechowujące obiekty danego typu. Dla przykładu:

- `IntSet` - implementacja zbioru przechowująca typy prymitywne `int`
- `IntegerQueue` - implementacja kolejki przechowująca elementy typu `Integer`

Analogicznie dla implementacji kolejki przechowującej elementy innych typów moglibyśmy stworzyć: `StringQueue` dla typu `String`, `DoubleQueue` dla typu `Double`, `PersonQueue` dla typu `Person`, itd.

Każda z implementacji różniłaby się tylko typem przechowywanych elementów. Operacje wykonywane na kolejce wyglądałyby analogicznie. Takie podejście kłóciłoby się z koncepcją ponownego użycia kodu.

Język Java wprowadza typy generyczne pozwalające zapobiec takim sytuacjom oraz uogólnić operacje na nich wykonywane.

Język Java: typy generyczne

Typ generyczny pozwala na stworzenie pojedynczej klasy, interfejsu lub metody, która może zostać użyty w połączeniu z różnymi typami danych.

> Typy generyczne nie współpracują z typami prymitywnymi.

Język Java: typy generyczne

W definicji klasy `<T>` definiuje parametr typu.

```
class Container<T> {

    private T data;

    public Container(T data) {
        this.data = data;
    }

    public T getData() {
        return this.data;
    }
}
```

Język Java: typy generyczne

Definicję możemy uogólnić do:

```
class ClassName<T1, T2, ..., Tn> {
    /* ... */
}
```

Można zauważyć, że liczba parametrów typu jest nieskończona. Zwyczajowo używamy liter `T, V, U, E, K` dla nazw parametrów typu.

Język Java: typy generyczne

Podczas inicjalizacji obiektu parametr typu zastępujemy konkretnym typem.

```
Container<String> containerOfString = new Container<String>("value");
Container<Integer> containerOfInteger = new Container<Integer>(1);
```

Podczas inicjalizacji z przypisaniem możemy pominąć pomiędzy nawiasami `<>` (ang. diamond operator). Będzie on automatycznie wydedukowany na podstawie typu zmiennej.

```
Container<String> containerOfString = new Container<>("value");
```

Programowanie: przykład 30

Definicja klasy generycznej.

```
class Container<T> {

    private T data;

    public Container(T data) {
        this.data = data;
    }

    public T getData() {
        return this.data;
    }
}

public class Main {

    public static void main(String[] args) {
        Container<String> containerOfString = new Container<String>("value");
        Container<Integer> containerOfInteger = new Container<Integer>(1);

        System.out.println("containerOfString: " + containerOfString.getData());
        System.out.println("containerOfInteger: " + containerOfInteger.getData());
    }
}
```

Język Java: typy generyczne

Typy generyczne możemy zagnieżdżać i tworzyć bardziej skomplikowane konstrukcje.

```
Container<Container<String>> container = new Container<>(new Container<>("value"));
```

Programowanie: zadanie 19

Stwórz kontener przechowujący pojedynczą wartość `Container<T>` oraz parę przechowującą dwie wartości `Pair<T, U>`. Przetestuj działanie, definiując dwie instancje o typach:

- `Pair<Integer, String>` oraz
- `Pair<Container<Integer>, Container<String>>`.

Język Java: problemy typów generycznych

Typy generyczne zostały wprowadzone do Javy w wersji 1.5. Przekazany typ jest usuwany podczas kompilacji, aby zapewnić wsteczną kompatybilność. Ciągle możliwe jest następujące przypisanie:

```
Container container1 = new Container<>(1);
Container container2 = new Container(1);
```

Co więcej, możemy przypisać tak zdefiniowaną generyczną instancję to zmiennej sparametryzowanej. Może to prowadzić do błędów podczas uruchomienia:

```
Container container = new Container(1);
Container<Integer> containerInteger = container;
Container<String> containerString = container;

Integer i = containerInteger.getData();
String s = containerString.getData();
```

Język Java: rozszerzenie typu generycznego

Do tej pory definiowaliśmy typy generyczne, które pozwalały na użycie dowolnych klas. Java wprowadza mechanizm ograniczający typy, które mogą zostać użyte do jako parametry typu generycznego. Ograniczenie to uzyskujemy poprzez rozszerzenie deklaracji o słowo kluczowe `extends`.

```
class Container<T extends Animal> {
    private T data;

    /* ... */
}
```

Dzięki temu nowy obiekt może zostać zainicjalizowany jedynie z typami, które rozszerzają klasę `Animal` lub też implementują interfejs `Animal`.

Główną zaletą takiego zapisu jest późniejsza możliwość użycia metod `Animal` z poziomu klasy generycznej.

Język Java: metody generyczne

Analogicznie do klas generycznych możemy zdefiniować metody generyczne (niestatyczne oraz statyczne).

```
class WithGenericMethods {

    public <T> void nonStaticMethod(T value) {
        System.out.println("nonStaticMethod: " + value);
    }

    public static <T> void staticMethod(T value) {
        System.out.println("staticMethod: " + value);
    }
}
```

Język Java: typ generyczny jako parametr

Podczas wywołania przekazujemy typ. Może on jednak zostać pomięty. Zostanie automatycznie wydedukowany przez kompilator na podstawie przekazanego parametru.

```
WithGenericMethods.<String>staticMethod("test");
WithGenericMethods.staticMethod("test");
```

Programowanie: przykład 31

```
class WithGenericMethods {

    public <T> void nonStaticMethod(T value) {
        System.out.printf("nonStaticMethod: %s: %s%n", value, value.getClass().getName());
    }

    public static <T> void staticMethod(T value) {
        System.out.printf("staticMethod: %s: %s%n", value, value.getClass().getName());
    }
}

public class Main {

    public static void main(String[] args) {
        WithGenericMethods withGenericMethods = new WithGenericMethods();
        withGenericMethods.<String>nonStaticMethod("test");
        withGenericMethods.nonStaticMethod("test");
        withGenericMethods.<Integer>nonStaticMethod(1);
        withGenericMethods.nonStaticMethod(1);

        WithGenericMethods.<String>staticMethod("test");
        WithGenericMethods.staticMethod("test");
        WithGenericMethods.<Integer>staticMethod(1);
        WithGenericMethods.staticMethod(1);
    }
}
```

Język Java: typ generyczny jako parametr

Podczas definicji metod ze sparametryzowanymi typami generycznymi definiowaliśmy typ:

```
public static <T> void consumeOfT(Container<T> value) { /* ... */ }
```

Jeżeli typ T jest nieistotny możemy użyć `<?>` (tzw. wildcard), który opisuje dowolny typ. Dzięki temu możemy pominąć definicję parametry typu.

```
public static void consumeOfWildcard(Container<?> value) { /* ... */ }
```

Język Java: typ generyczny jako parametr

Kolejną elementem języka jest ograniczanie typów sparametryzowanych przekazanych w parametrach metody.

Ograniczenie z góry (ang. upper bounds)

Pozwala na przekazanie klas rozszerzających dany typ.

```
public static void consumeOfExtendsAnimal(Container<? extends Animal> value) {
```

Ograniczenie z dołu (ang. lower bounds)

Pozwala na przekazanie klas będących rozszerzeniem danego typu.

```
public static void consumeOfSuperAnimal(Container<? super Animal> value) {
```

Programowanie: przykład 32

Użycie `upper bounds` oraz `lower bounds`.

```
public class Main {

    public static void main(String[] args) {
        Container<Cat> catContainer = new Container<>(new Cat());
        Container<Dog> dogContainer = new Container<>(new Dog());
    }

    public static <T> void consume1(Container<T> container) {
        // container.setData(new Dog());
        // container.setData(new Cat());
    }

    public static void consume2(Container<?> container) {
        // container.setData(new Dog());
        // container.setData(new Cat());
    }

    public static <T> void consume3(Container<? extends Animal> container) {
        // container.setData(new Dog());
        // container.setData(new Cat());
    }

    public static <T> void consume4(Container<? super Animal> container) {
        container.setData(new Dog());
        container.setData(new Cat());
    }

    public static void consume5(Container<Animal> container) {
        container.setData(new Dog());
        container.setData(new Cat());
    }

    public static void consume6(Container<Object> container) {
        container.setData(new Dog());
        container.setData(new Cat());
    }
}
```

Język Java: typy generyczne a dziedziczenie

Załóżmy, że `Dog extends Animal`. Niestety w tak zdefiniowanej hierarchii dziedziczenia nie zachodzi relacja `Container<Dog> extends Container<Animal>`.

Jest to związane z usuwaniem typu podczas kompilacji oraz kompatybilnością wsteczną.

Język Java: metod generyczne a przeciążanie

Przeciążanie metod nie zachodzi, dla typów generycznych. Poniższa deklaracja spowoduje błąd kompilacji.

```
public static void consume(Container<Dog> value) { }
public static void consume(Container<Cat> value) { }
```

Jest to związane z usuwaniem typu podczas kompilacji oraz kompatybilnością wsteczną.

Język Java: kolekcje

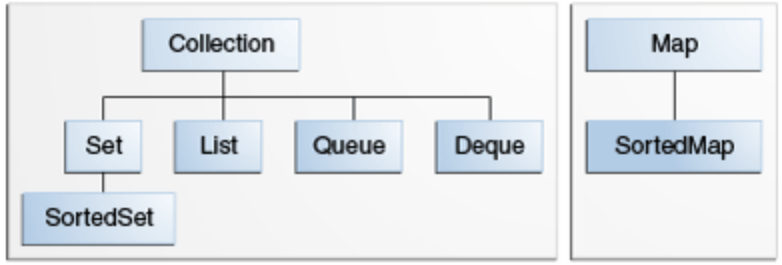
Java dostarcza szereg klas i interfejsów, które implementują struktury danych i algorytmy, o których była już mowa.

Przykładem może być klasa `HashSet` implementująca interfejs `Set`, który jest odzwierciedleniem struktury danych zbioru.

Do tej pory implementowaliśmy struktury danych. Pozwoliło to poznać ich zachowanie i możliwe implementacje. Od tego momentu będziemy się starali używać wbudowanych struktur.

Język Java: kolekcje

Zbiór dostępnych struktur danych możemy zaprezentować poprzez hierarchię interfejsów.



Źródło: https://docs.oracle.com/javase/tutorial/collections/interfaces/

Język Java: Collection

`Collection` - interfejs definiujący wspólne operacje na kolekcjach.

Do głównych operacji należą:

- `add()` - dodanie elementu do kolekcji.
- `addAll()` - dodanie wielu elementów do kolekcji.
- `remove()` - usunięcie elementu z kolekcji.
- `removeAll()` - usunięcie wielu elementów z kolekcji.
- `clear()` - usunięcie wszystkich elementów z kolekcji.
- `size()` - pobranie rozmiaru kolekcji.
- `iterator()` - pobranie iteratora pozwalającego na przejście po wszystkich elementach kolekcji.
- `contains()` - sprawdzenie, czy kolekcja posiada dany element.
- `containsAll()` - sprawdzenie, czy kolekcja posiada dane elementy.

Język Java: List

`List` - interfejs definiujący operacje na uszeregowanych kolekcjach, do których możemy odnosić się w sposób znany z tablic.

- `get()` - pobranie elementu pod danych indeksem.
- `set()` - ustawienie elementu pod danym indeksem.
- `remove()` - usunięcie elementu pod danym indeksem.

Język Java: Set

`Set` - interfejs definiujący operacje na zbiorach elementów, gdzie elementy są unikatowe w kolekcji. Nie rozszerza interfejsu `Collection` o istotne metody. Do głównych różnic zaliczamy:

- zmianę podejścia do przechowywanych elementów,
- brak uszeregowania elementów.

Język Java: Queue

`Queue` - interfejs definiujący operacje na kolejce, w której elementy zarządzane są w porządku FIFO.

- `offer()` - dodanie elementu do kolejki.
- `element()` - pobranie pierwszego elementu z kolejki lub błąd w przypadku braku elementu.
- `peek()` - pobranie pierwszego elementu z kolejki lub `null` w przypadku braku elementu.
- `remove()` - pobranie oraz usunięcie pierwszego elementu z kolejki lub błąd w przypadku braku elementu.
- `poll()` - pobranie oraz usunięcie pierwszego elementu z kolejki lub `null` w przypadku braku elementu.

Język Java: Deque

`Deque` - interfejs definiujący kolejkę dwustronną będącą rozszerzeniem klasycznej kolejki (ang. double ended queue). W tym przypadku elementy mogą zostać dodane i usunięte zarówno z początku, jak i z końca kolejki.

Do podstawowych operacji zaliczamy:
`addFirst()` - dodanie elementu na początku kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.
`addLast()` - dodanie elementu na końcu kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.
`offerFirst()` - dodanie elementu na początku kolejki lub zwrócenie `false` w przypadku pełnej kolejki.
`offerLast()` - dodanie elementu na końcu kolejki lub zwrócenie `false` w przypadku pełnej kolejki.

Język Java: Deque

Do podstawowych operacji zaliczamy:

`getFirst()` - pobranie elementu z początku kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.

`getLast()` - pobranie elementu z końca kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.

`peekFirst()` - pobranie elementu z początku kolejki lub zwrócenie `null` w przypadku pełnej kolejki

`peekLast()` - pobranie elementu z końca kolejki lub zwrócenie `null` w przypadku pełnej kolejki

Język Java: Deque

Do podstawowych operacji zaliczamy:
`removeFirst()` - pobranie oraz usunięcie elementu z początku kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.
`removeLast()` - pobranie oraz usunięcie elementu z końca kolejki lub zgłoszenie wyjątku w przypadku pełnej kolejki.
`pollFirst()` - pobranie oraz usunięcie elementu z początku kolejki lub zwrócenie `null` w przypadku pełnej kolejki.
`pollLast()` - pobranie oraz usunięcie elementu z końca kolejki lub zwrócenie `null` w przypadku pełnej kolejki.

Język Java: Deque

Dzięki dwukierunkowości kolejki interfejsy ten definiuje też metody znane ze stosu. Są to:
`push()` - dodanie elementu na początku stosu/kolejki dwukierunkowej.
`pop()` - pobranie oraz usunięcie elementu z początku stosu/kolejki dwukierunkowej.
`peek()` - obranie elementu z początku stosu/kolejki dwukierunkowej.

> Java definiuje również klasę `Stack` realizujący stos, ale nie będziemy się na niej koncentrować. Jest ona głównie przewidziana do środowisk wielowątkowych, gdyż jej metody są synchronizowane.

Język Java: Map

`Map` - interfejs definiujący operacje na elementach przechowywanych w parach klucz-wartość. Klucze są unikatowe. Dostęp do elementu odbywa się poprzez klucz.

Do podstawowych operacji zdefiniowanych przez interfejs zaliczamy:

- `put(K, V)` - dodanie elementu V pod kluczem K. Jeżeli istnieje element pod kluczem K, powinien on zostać zastąpiony nowym elementem.
- `putAll()` - dodanie wszystkich elementów.
- `putIfAbsent(K, V)` - dodanie elementu V pod kluczem K tylko w przypadku, gdy pod kluczem K nie istnieje żaden element.
- `get(K)` - pobranie elementu pod kluczem K lub `null` jeżeli nie istnieje taki element.
- `getOrDefault(K, V def)` - pobranie elementu pod kluczem K lub `def` jeżeli element nie istnieje.

Język Java: Map

Do podstawowych operacji zdefiniowanych przez interfejs zaliczamy:

- `containsKey(K)` - sprawdzenie, czy pod danym kluczem istnieje jakikolwiek element.
- `containsValue(V)` - sprawdzenie, czy dany element istnieje.
- `replace(K, V)` - zamiana elementu pod kluczem K, jeżeli pod kluczem istnieje element.
- `replace(K, V old, V new)` - zamiana elementu pod kluczem K, jeżeli pod kluczem K istnieje element oraz jego wartość jest równa `old`.
- `remove(K)` - usuwanie elementu pod kluczem K.
- `remove(K, V)` - usuwanie elementu pod kluczem K, jeżeli jego wartość to V.

Język Java: Map

Oraz operacje na wewnętrznych kolekcjach:

`keySet()` - pobranie wszystkich kluczy w mapie.

`values()` - pobranie wszystkich wartości w mapie.

`entrySet()` - pobranie wszystkich par klucz-wartość w mapie.

Język Java: Iterator

`Iterator` - interfejs dostarczający operacje umożliwiające przejście po wszystkich elementach danej struktury.

Wszystkie kolekcje dostarczają metodę `iterator()` zwracającą iterator.

Do podstawowych operacji zaliczamy:

- `hasNext()` - sprawdzenie, czy istnieje kolejny element w kolekcji.
- `next()` - pobranie następnego elementu z kolekcji.
- `remove()` - usunięcie ostatniego elementu z kolekcji.
- `forEachRemaining()` - wykonanie konkretnej akcji na wszystkich pozostałych elementach kolekcji.

Język Java: kolekcje

Do tej pory poznaliśmy interfejsy, które definiują zbiór operacji dostępnych dla danej strukturach danych. Istnieje szereg klas je implementujących. Skupimy się na najczęściej używanych.

- ArrayList
- LinkedList
- HashSet
- HashMap

Język Java: ArrayList

Kolekcja `ArrayList` implementuje interfejs `List`. Wewnętrznie używa tablicy do przechowywania elementów. Tablica ta jest rozszerzana wraz z dodawaniem nowych elementów do listy.

```
List<String> list = new ArrayList<String>();
```

Programowanie: przykład 33

Wykorzystanie listy oraz iteratora.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Main {

    public static void main(String[] args) {
        List<String> items = new ArrayList<>();
        items.add("item1");
        items.add("item2");
        items.add("item3");

        for (int i = 0; i < items.size(); i++) {
            System.out.println("for: " + items.get(i));
        }

        for (String item: items) {
            System.out.println("foreach: " + item);
        }

        Iterator<String> iterator = items.iterator();
        while (iterator.hasNext()) {
            System.out.println("iterator: " + iterator.next());
        }
    }
}
```

Język Java: LinkedList

Kolekcja `LinkedList` implementuje interfejsy `List` oraz `Deque`. Wewnętrznie używa dwukierunkowej listy elementów do przechowywania.

```
List<String> list = new LinkedList<String>();
Deque<String> deque = new LinkedList<String>();
```

Język Java: ArrayList a LinkedList

ArrayList

Dostęp do elementu pod wskazanym indeksem o złożoności O(1).
W teorii dodanie nowego elementu odbywa się w stałym czasie.
Niektóre operacje dodawania mogą powodować problemy wydajnościowe w związku z koniecznością kopiowania elementów.

LinkedList

Nie dostarcza szybkiego dostępu do elementów o wskazanym indeksie. Dotarcie do konkretnego elementu wymaga przejścia po liście.
Optymalna w sytuacjach, w których większość operacji to operacje dodawania elementów.

Język Java: HashSet

Kolekcja `HashSet` implementuje interfejs `Set`.

```
Set<String> set = new HashSet<String>();
```

Programowanie: przykład 34

Użycie `HashSet` wraz z ciągiem znaków oraz własną klasą.

```
public class Main {

    public static void main(String[] args) {
        Set<String> setOfString = new HashSet<>();
        setOfString.add("aaaaa");
        setOfString.add("bbbbb");
        setOfString.add("aaaaa");

        System.out.println(setOfString);

        Set<ClassNoEquals> setOfClassNoEquals = new HashSet<>();
        setOfClassNoEquals.add(new ClassNoEquals("aaaaa"));
        setOfClassNoEquals.add(new ClassNoEquals("bbbbb"));
        setOfClassNoEquals.add(new ClassNoEquals("aaaaa"));

        System.out.println(setOfClassNoEquals);

        Set<ClassWithEquals> setOfClassWithEquals = new HashSet<>();
        setOfClassWithEquals.add(new ClassWithEquals("aaaaa"));
        setOfClassWithEquals.add(new ClassWithEquals("bbbbb"));
        setOfClassWithEquals.add(new ClassWithEquals("aaaaa"));

        System.out.println(setOfClassWithEquals);
    }
}
```

Język Java: HashMap

Mapa `HashMap` implementuje interfejs `Map`.

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

Programowanie: zadanie 20

Zaimplementuj program pobierający słowa zadane przez użytkownika oraz zliczający ilość wystąpień danego słowa.

Do pobierania słów użyj metody `Scanner.next()`.
Do zliczania ilości wystąpień słów użyj klasy `HashMap`.

Przykładowe użycie programu.

Język Java: algorytmy na kolekcjach

`Collections.sort()` - sortowanie elementów.
`Collections.shuffle()` - przeorganizowanie elementów w pseudolosowej kolejności.
`Collections.reverse()` - odwrócenie kolejności elementów.
`Collections.fill()` - ustawienie każdego z elementów listy wartością zadaną.
`Collections.copy()` - przekopiowanie elementów z jednej kolekcji do drugiej przy zachowaniu indeksów.
`Collections.swap()` - podmienienie elementów na zadanych indeksach.
`Collections.addAll()` - dodanie wszystkich elementów jednej kolekcji do drugiej.
`Collections.frequency()` - sprawdzenie ilości wystąpień danej wartości w kolekcji.
`Collections.disjoint()` - sprawdzenie, czy dwie listy posiadają wspólne elementy.
`Collections.min()` - wyszukanie najmniejszego elementu.
`Collections.max()` - wyszukanie największego elementu.

Język Java: typ wyliczeniowy

W Javie typ wyliczeniowy (ang. enumeration) jest typem o stałych wartościach, które są inicjalizowane podczas implementacji. Typ ten definiujemy wraz z listą wartości, które może przyjmować.

Dla przykładu:

```
enum Direction {
   NORTH, SOUTH, EAST, WEST
}
```

Stworzyliśmy typ wyliczeniowy `Direction` z 4 możliwymi wartościami. Wartości te możemy przypisać do zmiennej:

```
Direction direction = Direction.NORTH;
```

Język Java: typ wyliczeniowy

Typ wyliczeniowym jest specjalnym typem klasy. Uogólniając:

```
enum Name {
   VAL1, VAL2, ..., VALN
}
```

> Stałe wartości typu wyliczeniowego są reprezentowane przez słowa pisane z wielkich liter.

Typy wyliczeniowe zostały wprowadzone, aby usprawnić definicje stałych.

```
class Direction {
   public final static int NORTH = 1;
   public final static int SOUTH = 2;
   public final static int EAST = 3;
   public final static int WEST = 4;
}
```

Język Java: typ wyliczeniowy

Jak każda z klas typ wyliczeniowy może posiadać: konstruktor, pola, oraz metody.

Co istotne konstruktor jest automatycznie prywatny i nie może zostać wywołany spoza klasy.

```
public enum Direction {
    NORTH("N"), SOUTH("S"), EAST("E"), WEST("W");

    private String shortVersion;

    Direction(String shortVersion) {
        this.shortVersion = shortVersion;
    }

    public String getShortVersion() {
        return shortVersion;
    }
}
```

Język Java: typ wyliczeniowy

Do wbudowanych metod dla każdego typu wyliczeniowego należą:

- `ordinal()` - pobranie pozycji danej wartości w postaci liczby całkowitej
- `compareTo()` - porównanie dwóch wartości na podstawie `ordinal`.
- `toString()` - pobranie reprezentacji tekstowej danej wartości.
- `name()` - pobranie nazwy wartości w formacie tekstowym. Metoda ta nie może zostać nadpisana.
- `valueOf()` - pobranie wartości na podstawie reprezentacji tekstowej. Jeżeli nie istnieje, zostanie zgłoszony wyjątek.
- `values()` - pobranie tablicy z wszystkimi wartościami dla danego typu wyliczeniowego.

Programowanie: zadanie 21

Dla typu wyliczeniowego:

```
enum Direction {
   NORTH, SOUTH, EAST, WEST
}
```

Zaimplementuj metodę `Direction safeValueOf(String value)` zwracającą wartość typu wyliczeniowego na podstawie reprezentacji tekstowej lub `null` jeżeli nie istnieje.

Język Java: niestatyczne klasy zagnieżdżone

Niestatyczne klasy zagnieżdżone (lub klasy wewnętrzne, ang. non-static nested classes, inner classes) są to klasy, które są zdefiniowanie w ciele innej klasy, mające one dostęp to pól i metod klasy zewnętrznej.

```
class Car {

    private String model;

    class Engine {

        private Integer power;
    }
}
```

Język Java: niestatyczne klasy zagnieżdżone

W związku z tym, że klasy wewnętrzne mają dostęp do klasy zewnętrznej, w pierwszej kolejności konieczne jest zainicjalizowanie klasy zewnętrznej. Można powiedzieć, że klasy wewnętrzne istnieją tylko w kontekście klas zewnętrznych.

```
Car car = new Car();
Car.Engine engine = car.new Engine();
```

Język Java: niestatyczne klasy zagnieżdżone

Dostęp do klasy zewnętrznej z ciała klasy wewnętrznej odbywa się poprzez nazwę klasy.

```
class Car {

    private String brand;

    class Engine {

        private String brand;

        public String getOverallBrand() {
            return String.format("%s %s", Car.this.brand, this.brand);
        }
    }
}
```

Język Java: statyczne klasy zagnieżdżone

Statyczne klasy zagnieżdżone (lub klasy wewnętrzne, ang. static nested classes) są to statyczne klasy, które są zdefiniowanie w ciele innej klasy. W związku z tym nie mają one dostępu to pól i metod klasy zewnętrznej.

```
class Car {

    private String model;

    static class Engine {

        private Integer power;
    }
}
```

Język Java: statyczne klasy zagnieżdżone

W związku z tym, że statyczne klasy zagnieżdżone nie mają dostępu do klasy zewnętrznej, do ich inicjalizacji nie używamy kontekstu klasy zewnętrznej.

```
Car.Engine engine = new Car.Engine();
```

Język Java: klasy anonimowe

Klasy anonimowe to klasy, które zdefiniowane są zaraz podczas inicjalizacji. Nie posiadają one nazwy oraz mogą byc zdefiniowane na poziomie klasy zewnętrznej lub w metodzie.

Dla przykładu:

```
public class Main {

    public static void main(String[] args) {
        ParentType anonymous = new ParentType() {
            public String toString() {
                return "Anonymous of ParentType";
            }
        };
    }
}
```

Gdzie `ParentType` to wcześniej zdefiniowany interfejs, który chcemy zaimplementować lub klasa, którą chcemy rozszerzyć.

Język Java: klasy anonimowe

Głównym miejscem użycia klas anonimowych są przypadki, w których dana definicja klasy jest użyta tylko raz w miejscu inicjalizacji.

```
interface Message {
    String getValue();
}

public class Main {

    public static void main(String[] args) {
        System.out.println(getDefaultMessage().getValue());
    }

    public static Message getDefaultMessage() {
        return new Message() {
            public String getValue() {
                return "Anonymous of ParentType";
            }
        };
    }
}
```

Język Java: słowo kluczowe final

Słowo kluczowe `final` definiuje stałą, która jest niezmienna. Może ono byc użyte w połączeniu ze zmiennymi, klasami oraz metodami.

Zmienna `final`

Zmienna `final` nie może zainicjalizowana drugi raz.

```
final int value = 43;
```

Język Java: słowo kluczowe final

Metoda `final`

Metoda `final` nie może być przesłonięta.

```
public final int count() {
    /* ... */
}
```

Klasa `final`

Klasa `final` nie może być rozszerzona.

```
public final class StringWrapper {
    /* ... */
}
```

Język Java: rzutowanie typów

Przy okazji typów prymitywnych mówiliśmy o automatycznej konwersji z typu prymitywnego do odpowiedniego typu obiektowego (tzw. autoboxing). Podobnie dla typów prymitywnych konwertowaliśmy typ `double` do typu `int`.

W przypadku konwersji typów mamy do czynienia z ich rzutowaniem.

Dla przykładu:

```
double d = 7.5;
int i = (int) d;
```

Język Java: rzutowanie typów

Rzutowanie możemy też wykorzystać przy okazji obiektów oraz hierarchii dziedziczenia. Należy jednak uważać na potencjalne błędy związane z użyciem złych typów.

```
interface Animal {}
class Dog implements Animal {}
class Cat implements Animal {}

public class Main {

    public static void main(String[] args) {
        Animal animal = new Dog();

        Dog dog = (Dog) animal;
        // Cat cat = (Cat) animal; /* ClassCastException */
    }
}
```

> Aby zabezpieczyć się przed błędami rzutowania, warto używać operatora instanceof przed samą operacją rzutowania.

Język Java: rzutowanie typów

Istnieją dwa typy rzutowania:

Rzutowanie rozszerzające (Widening Type Casting)

Konwersja zachodzi automatycznie, ponieważ typ o mniejszym rozmiarze zostaje zastąpiony typem o rozmiarze większym.

Przykładowo: konwersja `int` do `double`.

```
int i = 1;
double d = i;
```

Język Java: rzutowanie typów

Rzutowanie zawężające (Narrowing Type Casting)

Konwersja musi być zdefiniowana explicite, ponieważ typ o większym rozmiarze zostaje zastąpiony typem o rozmiarze mniejszym.

Przykładowo: konwersja `double` do `int`.

```
double d = 1.0;
int i = (int) d;
```