Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Język Java – wykład 4

dr inż. Łukasz Sosnowski
lukasz.sosnowski@wit.edu.pl
sosnowsl@ibspan.waw.pl
l.sosnowski@dituel.pl

www.lsosnowski.pl

Część 1 – Interfejsy w JAVA

Podstawowe informacje

- Słowo kluczowe *interface* definiuje element, który reprezentuje koncepcję zupełnego rozdzielenia interfejsu klasy od jej implementacji w języku JAVA.
- Interfejsy określają co ma być zrobione bez definiowania w jaki sposób.
- Koncepcja podobna do klasy abstrakcyjnej w zakresie deklarowania metody bez ciała, lecz dająca więcej możliwości.
- Interfejs implementowany jest przez klasę w celu dostarczenia ciała metod.
- Klasa może implementować wiele interfejsów.
- Ten sam interfejs może być implementowany przez wiele różnych klas przy każda z implementacji może być różna.

Podstawowe informacje c.d.

- Od wersji 8 możliwe jest definiowanie w interfejsie implementacji domyślnej co daje możliwość określania zachowania metody.
- Od wersji 9 można umieszczać metody statyczne oraz metody prywatne.
- Rozszerzenia te stanowią jednak pewne przypadki szczególne. W ogólności należy przyjąć, iż interfejsy będą nam służyły do oddzielenia implementacji.
- Interfejs może zawierać zmienne, lecz nie są to zmienne instancji Niejawnie są zadeklarowane jako *public final static*, a więc pełnią rolę stałych. Dodatkowo wymagają inicjalizacji.
- Deklaracja interfejsu:

```
<public/brak> interface <nazwa>{
     typ metoda(lista parametrów);
     typ zmienna = wartość;
}
```

Implementacja interfejsów

- Do implementacji interfejsu służy klauzula *implements*, która umieszczana jest w definicji klasy oraz wskazanie interfejsów które mają być implementowane (może być wiele).
- Deklaracja klasy implementującej interfejs:

```
class NazwaKlasy implements NazwaInterfejsu, NazwaInterfejsu2{
}
```

- Klasa zwykła implementująca interfejs implementuje wszystkie metody interfejsu.
- Metody interfejsu są metodami abstrakcyjnymi (poza domyślnymi).
- Brak dostarczenia pełnej implementacji przez klasę można obsłużyć przez zadeklarowanie klasy jako abstrakcyjnej.

Przykład

```
public interface IParameter {
	String getName();
	Object getValue();
	String debug();
}

public final class Parameter implements IParameter{
	private String name = null;
	private Object value = null;
	public Parameter(String name,Object value) {
		this.name=name;
		this.value = value;
	}
	public String getName() {
		return name;
	}
	public Object getValue() {
		return value;
	}
	public String myToString() {
		return "name=".concat(getName())
				.concat(", value=").
				concat(getValue().toString());
	}
	public String debug() {
		return myToString();
	}
}
```

Przykład

```
@Test
public void paramTest() {
	Parameter p1 = new Parameter("portalId",2);
	System.out.println(p1.myToString());
	IParameter iP1 = p1;
	//Błąd kompilacji
	//System.out.println(iP1.myToString());
	System.out.println("name2="+iP1.getName()
	+", value2="+iP1.getValue().toString());
}
```

Wynik:

```
name=portalId, value=2
name2=portalId, value2=2
```

Korzystanie z referencji do interfejsu

- Interfejsy dają możliwość deklaracji zmiennej referencyjnej .
- Zmienna może wskazywać na dowolny obiekt klasy implementującej dany interfejs.
- Z użyciem referencji interfejsu uzyskujemy dostęp do metod zadeklarowanych w tym interfejsie.
- Przy odwołaniu do metody zostanie użyta implementacja klasy obiektu, którego referencja została przypisana do zmiennej.
- Przykład: (dla interfejsów i klas z poprzedniego slajdu a Parameter2 ma implementacje identyczną jak Parameter).

```
@Test
public void param2Test() {
      Parameter p1 = new Parameter("x",20);
      Parameter2 p2 = new Parameter2("y",15);
      IParameter params[] = new IParameter[] {p1,p2};
      for(IParameter p:params)
            System.out.println(p.debug());
}
```

```
Wynik:
name=x, value=20
II-name=y, value=15
```

Dziedziczenie interfejsów

- Dziedziczenie interfejsów realizowane jest poprzez zastosowanie słowa kluczowego *extends*.
- Interfejs obsługuje wielokrotne dziedziczenie z innych interfejsów. Kolejne interfejsy podawane są po przecinku.
- Przy dziedziczeniu musi być zachowana spójność deklaracji metod z poziomu ich sygnatur.
- Klasa standardowa implementująca interfejs, który dziedziczy z innych interfejsów musi implementować wszystkie metody unikalne pod względem sygnatury.
- Deklaracja:

```
interface INazwaInterfejsu extends IInterfejs1,IInterfejs2 {
}
```

Domyślne implementacje metod w interfejsach

- Metody domyślne tworzone są z użyciem słowa kluczowego *default*. Pojawienie się możliwości definiowania implementacji rozszerza możliwości interfejsu, lecz nie zastępuje klas.
- Metody domyślne funkcjonują również pod pojęciem metody rozszerzeń.
- Funkcjonalność bardzo użyteczna w przypadkach modyfikacji już istniejącego interfejsu, aby zmiana nie powodowała braku kompilacji. Zapewnia szybką możliwość wprowadzenia zmiany.
- Daje możliwość implementacji opcjonalności metody poprzez zapewnienie domyślnej implementacji, z możliwością przesłonięcia.

```
default String testToString() {
      return debug();
}
```

Metody statyczne w interfejsach

- Od wersji 8 JDK w interfejsach można definiować metody statyczne wraz z ich implementacją.
- Wywołanie metody odbywa się analogicznie jak metody statycznej klasy, czyli: INazwaInterfejsu.metoda();
- Stanowi to bardzo duże rozwinięcie możliwości, gdyż do realizacji wykonania metody nie potrzebujemy obiektu klasy implementującej dany interfejs!!!
- UWAGA: statyczne metody interfejsu nie są dziedziczone!!! Dotyczy to zarówno dziedziczenia interfejsów jak również implementacji interfejsu przez klasę.

Metody prywatne w interfejsach

- Od wersji 9 JDK interfejsy zezwalają na deklarowanie i implementację metod prywatnych (sygnatura i ciało metody).
- Metody prywatne mogą jedynie być wykorzystywane w implementacjach domyślnych metod interfejsu.
- Metody te nie są dziedziczone (prywatne!!!) przy dziedziczeniu interfejsów.
- Metody prywatne interfejsu nie są widoczne w klasach implementujących dany interfejs.
- Metody te „klasycznie” używają modyfikatora dostępu *private* oraz definiują ciało metody.

Część 2 – Klasy wyliczeniowe

Klasy wyliczeniowe – podstawowe informacje

- Umożliwiają tworzenie własnego typu danych o ściśle zdefiniowanej liście wartości
- Przykładem takiego typu może być: Nazwy miesięcy, nazwy dni tygodnia, kwartały, zdefiniowane elementy słownikowe, które nie podlegają rozbudowie.
- Stanowią wygodny sposób na grupowanie zmiennych tego samego typu
- Wyliczenia przyjmują wielkie i małe litery.
- Do tworzenia instancji klasy nie używamy operatora *new*.

Klasy wyliczeniowe

- Wyliczenia są klasami, które dziedziczą po klasie Enum.
- Wyliczenia nie mogą dziedziczyć z innych klas wskazanych przez programistę.
- Wyliczenia nie mogą być klasą bazową dla innej klasy.
- Wyliczenie jest listą stałych posiadających nazwę.
- Elementy wyliczenia nazywamy stałymi wyliczeniowymi. Każda z nich jest zadeklarowana jako *public static.*
- Wyliczenie definiowane jest jako:

```
enum NazwaTypuWyliczeniowego{
stała1, stała2,…
}
```

- Dozwolone jest tworzenie zmiennych typów wyliczeniowych:

```
TypWyliczeniowy nazwaZmiennej;
```

Klasy wyliczeniowe c.d.

- Stałe wyliczeniowe można porównywać przy użyciu operatora ==.
- Stałe wyliczeniowe można używać w instrukcji *switch.*
- Stałe wyliczeniowe przy konwersji na String są reprezentowane przez ich nazwę.
- Wyliczenia posiadają wbudowane metody values() i valueOf(String). Pierwsza zwraca tablicę typu wyliczeniowego dla którego jest wywołana. Druga zwraca stałą wyliczeniową, której nazwa odpowiada wartości przekazanego argumentu.
- Stałe wyliczeniowe jest obiektem typu wyliczeniowego
- Klasy wyliczeniowe mogą definiować konstruktory prywatne i własne metody.
- Wbudowana metoda *ordinal()* zwracająca pozycję na liście.

Przykład

```
public enum EnProfessions {
	SECRETARY, PROGRAMMER, TESTER, ANALYST, SALESMANAGER
}

@Test
public void compareTest() {
	EnProffessions pr1 = EnProfessions.PROGRAMMER;
	EnProffessions pr2 = EnProfessions.PROGRAMMER;
	assertEquals(EnProfessions.PROGRAMMER, pr2);
	assertEquals(pr1,pr2);
	assertSame(pr1,pr2);
	if (pr2 == EnProfessions.SALESMANAGER) {
		fail("Błędny zawód");
	}
	for (EnProfessions pr : EnProfessions.values()) {
		switch (pr) {
		case SECRETARY: {
			log.trace(EnProfessions.SECRETARY);
			break;
		}
		case PROGRAMMER: {
			log.trace(EnProfessions.PROGRAMMER);
			break;
		}
		case TESTER: {
			log.trace(EnProfessions.TESTER);
			break;
		}
		default:
			log.trace("default:" + pr.toString());
			break;
		}
	}
}
```

```
0 [main] TRACE pl.wit.lab4.EnumTest  - SECRETARY
1 [main] TRACE pl.wit.lab4.EnumTest  - PROGRAMMER
1 [main] TRACE pl.wit.lab4.EnumTest  - TESTER
1 [main] TRACE pl.wit.lab4.EnumTest  - default:ANALYST
 1 [main] TRACE pl.wit.lab4.EnumTest  - default:SALESMANAGER
```

Przykład

```
public class EnumProfessionNotPresentException extends EnumConstantNotPresentException {

	private static final long serialVersionUID = -5839018676212618866L;

	public EnumProfessionNotPresentException(Class<? extends Enum> enumType, String constantName) {
		super(enumType, constantName);
	}

	public EnumProfessionNotPresentException(Class<? extends Enum> enumType, int id) {
		super(enumType, "id="+id);
	}
}

public enum EnProfessions2 {
	CEO(12), CTO(32), SECRETARY(43), PROGRAMMER(56), TESTER(65), ANALYST(77), SALESMANAGER(79);

	private int id;

	EnProfessions2(int id) {
		this.id = id;
	}

	public int getId() {
		return id;
	}

	public static EnProfessions2 getById(int id){
		for(EnProfessions2 ep:EnProfessions2.values()){
			if(ep.getId()==id)
				return ep;
		 }
		throw new EnumProfessionNotPresentException(EnProfessions2.class,id);
	}
}
```

Część 3 – Automatyczne opakowanie, słowo kluczowe *static*

Automatyczne opakowywanie

- Automatyczne opakowanie i automatyczne wypakowanie to mechanizmy dostępne od JDK5 pozwalające zamieniać wartości typów prostych na typy obiektowe i odwrotnie.
- Mechanizm ten jest bezpośrednio związany z typami opakowującymi.
- Typy proste nie są obiektowe. Nie dziedziczą więc z klasy *Object*. Nie można ich przekazywać przez referencję. Dlatego też często istnieje potrzeba użycia reprezentacji obiektowej danej zmiennej typu prostego.
- Typy opakowujące to: *Double*, *Float*, *Long*, *Integer*, *Short*, *Byte*, *Character*, *Boolean*. Wszystkie pochodzą z pakietu *java.lang*
- Klasy opakowujące definiują odpowiednie konstruktory umożliwiające stworzyć obiektu na podstawie wartości typu prostego (do JDK9)

Automatyczne opakowywanie

- Od JDK9 konstruktory te zostały uznane za przestarzałe (ang. deprecated). Aktualnie zalecane jest używanie metod statycznych *valueOf*(), które w poszczególnych klasach są przeciążone również dla argumentu klasy *String*.
- Wszystkie typy opakowujące przesłaniają metodę toString() zwracającą czytelną wartość przechowywaną w obiekcie.
- Proces hermetyzacji wartości typu prostego za pomocą obiektu nazywamy **opakowaniem**.
- Proces pobierania wartości opakowanej z obiektu nazywamy **wypakowaniem**.
- Manualne opakowanie i wypakowanie wygląda tak:
  Integer iObject = Integer.valueOf(5); int i = iObject.intValue();
- Typy opakowujące tworzą nowy obiekt przy modyfikacjach wartości typu prostego.

Automatyczne opakowywanie

- Automatyczne opakowanie polega na automatycznej hermetyzacji, bez jawnego tworzenia obiektu. Analogicznie automatyczne wypakowanie.
- Mechanizm uruchamia się wszędzie tam, gdzie wymagany jest typ obiektowy a podany typ prosty i odwrotnie.
- Przykład: *Integer iObject = 5; //Opakowanie*
  int i = iObject; //Wypakowanie
- Należy zwrócić uwagę iż w pozornie prostych operacjach opakowywanie i wypakowywanie może być automatycznie uruchamiane wielokrotnie, np.. *Integer iObject =5; iObject++;*
- Automatyczne opakowywanie i wypakowanie stanowi niezerowy koszt. Dlatego też trzeba konstruować kod unikając wywoływania tego mechanizmu, tam gdzie nie jest to niezbędne.

Przykład

```
@Test
public void boxingUnBoxingTest() {
      long start = System.currentTimeMillis();
      Double x,y,z;
      Double mean = null;
      x=11.3;
      y=12.9;
      z=10.1;
      for(int i=0;i<100000;i++) {
            mean = (x+y+z)/3;
      }
      long end = System.currentTimeMillis();
      assertNotNull(mean);
      System.out.println("Czas z opakowywaniem="+(end-start)+" ms");
}
```

→

```
Czas z opakowywaniem=6 ms
```

```
@Test
public void withoutBoxingUnBoxingTest() {
      long start = System.currentTimeMillis();
      double x,y,z;
      double mean=0.0d;
      x=11.3;
      y=12.9;
      z=10.1;

      for(int i=0;i<100000;i++) {
            mean = (x+y+z)/3;
      }
      long end = System.currentTimeMillis();
      assertNotEquals(0.0,mean,0.0);
      System.out.println("Czas bez opakowywania="+(end-start)+" ms");
}
```

```
Czas bez opakowywania=1 ms
```

Słowo kluczowe *static*

- Umożliwia tworzenie zmiennych składowych klasy nie związanych z obiektem danej klasy. Do takich zmiennych odwołujemy się poprzez nazwę klasy, np. *Person.typeId = 50;*
- Zmienne poprzedzone słowem kluczowym *static* są zmiennymi globalnymi. Wszystkie obiekty danej klasy współdzielą tę samą wartość zmiennej;
- Jako statyczne mogą również być deklarowane metody. Wtedy wywołanie następuje poprzez nazwę klasy, np. *Person.process();*
- Istnieje kilka ograniczeń dla metod static:
  - mogą wywoływać tylko inne metody statyczne
  - mogą używać bezpośrednio tylko zmiennych statycznych
  - nie można dla nich używać referencji *this*
- Możliwe są bloki instrukcji *static*. Blok jest wywołany podczas ładowania klasy.

Słowo kluczowe *static c.d*

- Słowo kluczowe *static* może występować również przy deklarowaniu importu klasy lub metod. Umożliwia to używanie metod wewnątrz klasy bez poprzedzania ich nazwą klasy.
- Przykład:

```
public class Quadrat {
      public static double area(double a, double b) {
            return a*b;
      }
}

import static pl.wit.lab4.Quadrat.area;
```

lub

```
import static pl.wit.lab4.Quadrat.*;


@Test
public void test() {
      assertEquals(6d,area(2d, 3d),0.0);
}
```

Część 4 – typowe kolekcje w języku JAVA
cz.1 – interfejsy kolekcji

Wprowadzenie do kolekcji

- Szkielet Collections pojawił się od JAVA 1.2
- Szkielet spełnia kilka ważnych funkcji:
  - dostarcza niezwykle wydajnych implementacji podstawowych kolekcji (tablic dynamicznych, list, drzew, tablic mieszających),
  - ujednolica sposób korzystania z różnych rodzajów kolekcji,
  - umożliwia rozszerzanie i dostosowywanie kolekcji,
  - bazuje na jednym zbiorze interfejsów, które są implementowane w klasach dostarczonych w szkielecie, ale mogą być również implementowane we własnych klasach,
  - zapewnia mechanizmy integrujące szkielet z tablicami.
- Dodatkowo szkielet zapewnia algorytmy dla kolekcji, dostarczone w postaci metod statycznych (omawiane na laboratorium – prezentacje studentów).
- Mapy jako rozwinięcie.

Interfejs Collection

- Podstawowy interfejs szkieletu. Implementują go wszystkie klasy definiujące kolekcje.
- Interfejs sparametryzowany postaci: interface Collection<T>, gdzie T określa typ przechowywanego obiektu.
- Interfejs ten rozszerza interfejs Iterable, co powoduje, że przez każdą kolekcję można dokonać iteracji z użyciem pętli typu for-each.
- Interfejs określa szereg metod: add() i addAll() -dodające elementy, remove() i removeAll – usuwające elementy, clear() - usuwająca wszystkie elementy kolekcji, contains() i containsAll() sprawdzające czy w kolekcji znajduje się obiekt lub obiekty wskazane w argumencie. Dodatkowo iterator(), size(), retainAll(), czy też metoda stream() tworząca strumień obiektów.

Interfejs List

- Rozszerza interfejs Collections, dodając zachowanie związane z przechowywaniem sekwencji elementów. Elementy mogą być wstawiane lub pobierane na podstawie ich położenia.
- Lista indeksuje położenie od zera.
- Lista może zawierać duplikaty elementów
- Lista jest sparamentryzowanym interfejsem: List<T>, gdzie T określa typ przechowywanych obiektów.
- Interfejs listy definiuje własne metody: add() i addAll() używające indeksu jako wskazania miejsca wstawienia elementu/ów, get() – pobierająca obiekt ze wskazanego indeksu, indexOf i lastIndexOf zwracająca pozycję indeksu na której znajduje się wystąpienie zaadanego obiektu (pierwsze lub ostatnie), remove() - usuwająca element ze wskazanego indeksu, i inne: replaceAll(), set(), subList(), etc.

Interfejs Set

- Definiuje matematyczne pojęcie zbioru rozszerzając jednocześnie interfejs Collection.
- Cechą charakterystyczną zbioru jest unikalność elementów (brak duplikatów).
- Metoda add() interfejsu zwraca wartość false, jeśli dodawany element już istniał w zbiorze.
- Interfejs jest sparametryzowany, w postaci: Set<T>, gdzie T oznacza typ obiektu przechowywanego w zbiorze.
- Interfejs definiuje jedynie 2 własne metody: of() i copyOf(). Pierwsza ma wiele przeciążonych wersji. Służy do tworzenia zbioru na podstawie podanych obiektów. Druga tworzy zbiór niemodyfikowalny na podstawie przekazanej kolekcji. W obu przypadkach wartości null nie są dopuszczalne jako elementy do utworzenia zbioru.

Interfejs SortedSet

- Rozszerza interfejs Set i deklaruje zachowanie dla zbioru posortowanego w porządku rosnącym.
- Interfejs jest sparametryzowany w postaci: SortedSet<T>, gdzie T jest typem obiektu przechowywanego w zbiorze.
- Interfejs deklaruje następujące własne metody: **comparator**() zwracającą komparator użyty do posortowania zbioru lub null w przypadku wykorzystania porządku naturalnego, **first**() zwracającą pierwszy element zbioru, **headSet**(el) zwraca SortedSet zawierające elementy mniejsze niż przekazany w argumencie, ale znajdujące się w zbiorze, **last**() zwraca ostatni element zbioru, **subSet**(start, end) – zwraca SortedSet zawierając elementy pomiędzy start (włącznie) i end (bez), **tailSet**(start) – zwraca SortedSet z elementami >= od start.

Interfejs NavigableSet

- Rozszerza interfejs SortedSet i deklaruje zachowanie kolekcji odnajdującej elementy o najbliższym dopasowaniu do jednej lub wielu wartości.
- Interfejs jest sparametryzowany w postaci: NavigableSet<T>, gdzie T jest typem przechowywanych obiektów.
- Interfejs dodaje m.in. metody: **ceilling**() - wyszukuje w zbiorze najmniejszy element typu T większy lub równy przekazanemu w argumencie, **descendingIterator** – zwraca iterator odwrotny, **descendingSet**() - zwraca egzemplarz interfejsu zawierający odwrócony zbiór (z tymi samymi referencjami do elementów), **floor**() - wyszukuje największy element w zbiorze mniejszy lub równy przekazanemu w argumencie, **higher**(), **lower**() - zwracające odpowiednio najmniejszy/największy element większy od przekazanego.

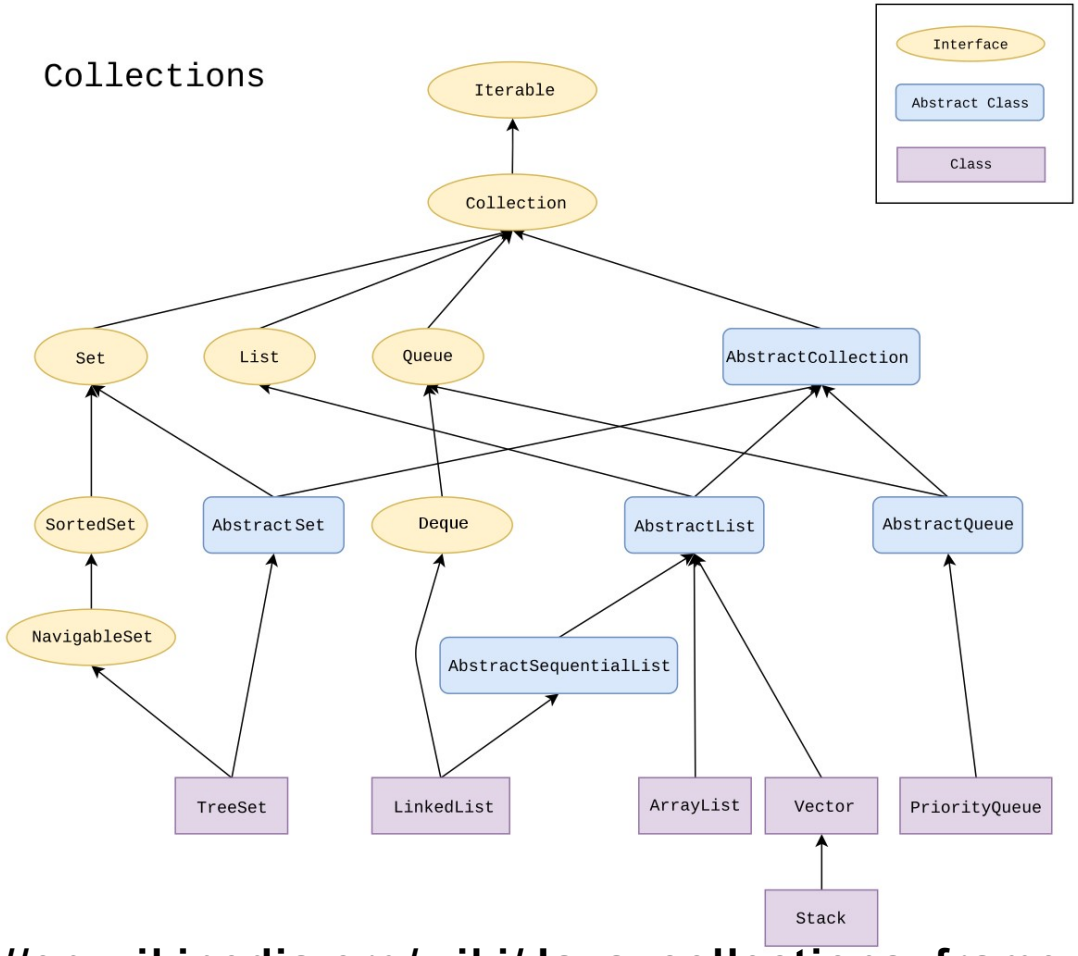
Interfejs Queue

- Rozszerza interfejs Collections i definiuje zachowanie kolejki, (najczęściej FIFO).
- Queue jest interfejsem sparametryzowanym w postaci: Queue<T> gdzie T okresla typ obiektu przetwarzany przez kolejkę.
- Interfejs definiuje własne metody: **element**() - zwraca pierwszy element kolejki, **offer**(T obj) – dodaje element do kolejki zwracając true jeśli się to uda lub false w p.p., **peek**() - zwraca pierwszy element kolejki lub null jeśli kolejka jest pusta, **poll**() zwraca pierwszy element kolejki i go usuwa. Dla pustej kolejki zwraca null, **remove**() - usuwa i zwraca pierwszy element kolejki, lecz jeśli kolejka jest pusta generuje wyjątek **NoSuchElementException.**

Interfejs Deque

- Rozszerza interfejs Queue i deklaruje zachowanie kolejki dwukierunkowej (działającej zarówno jako FIFO jak i LIFO).
- Interfejs jest sparametryzowany w postaci: Deque<T> gdzie T oznacza typ przechowywanego obiektu.
- Interfejs definiuje sporą liczbę własnych metod. Warto zwrócić uwagę na metody **push**() i **pop**(), które umożliwiają pracę w trybie stosu.
- Inne metody: **addFirst**(), **addLast**(), **getFirst**(), **getLast**(), **offertFirst**(), **offerLast**(), **peekFirst**(), **peekLast**(), **pollFirst**(), **pollLast**() - odpowiednio realizują dodawanie, pobieranie, pobieranie z usuwaniem. Stanowią rozwinięcie lecz przez analogię metod interfejsu Queue.

Hierarchia



*** - źródło w https://en.wikipedia.org/wiki/Java_collections_framework**

Podsumowanie

- Interfejsy
- Dziedzicznie interfejsów
- Typy wyliczeniowe
- Automatyczne opakowywanie i wypakowywanie
- Zmienne i metody statyczne statyczne
- Interfejsy kolekcji - omówienie

Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Dziękuję za uwagę!