Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Język Java – wykład 8

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

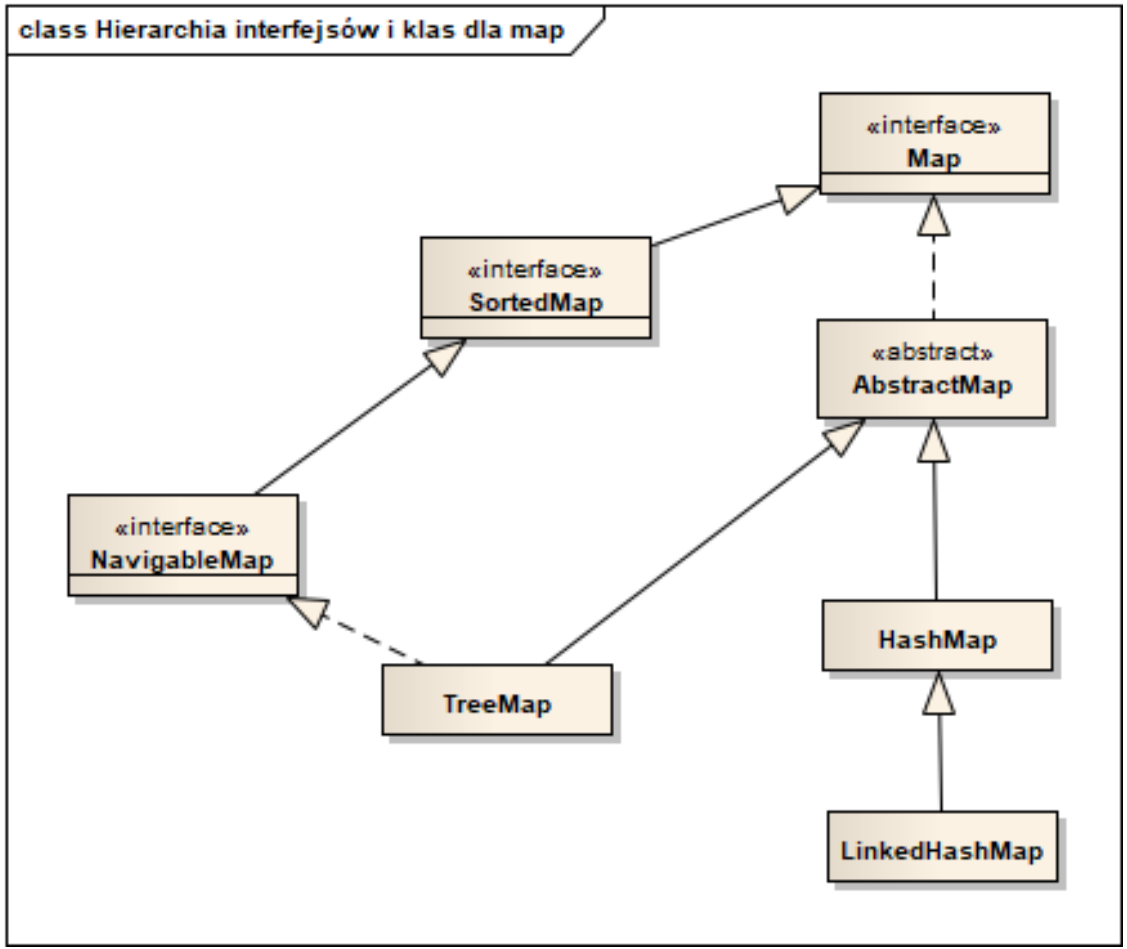
www.lsosnowski.pl

Część 1 – java.util.Collections - Mapy

Mapy – informacje podstawowe

- Mapa to obiekt, który składa się z klucza oraz wartości.
- Wartość można odnaleźć podając jej klucz
- Zarówno klucz jak i wartość stanowią obiekty (nie mogą to być typy proste)
- Klucze muszą być unikalne, wartości mogą zawierać duplikaty
- W zależności od rodzaju mapy, wartość null może być dopuszczona jako klucz lub w niektórych jako wartość
- Mapa nie implementuje interfejsu *Iterable*, nie można pobrać z niej iteratora.
- Nie można zatem bezpośrednio dla mapy zastosować iteracji poprzez pętle „for”.
- Można natomiast to wykonać z użyciem dedykowanych metod widoku kolekcji.

Hierarchia


class Hierarchia interfejsów i klas dla map
«interface»
Map
«interface»
SortedMap
«abstract»
AbstractMap
«interface»
NavigableMap
HashMap
TreeMap
LinkedHashMap

Interfejs Map

- Określa mapę z unikalnymi kluczami odpowiadającymi wartościom.
- W celu dodania elementu do mapy potrzebny jest zarówno klucz jak i wartość, natomiast w celu pobrania jedynie klucz.
- Interfejs Map jest sparametryzowany w następujący sposób: *interface Map<K,V>*, gdzie K reprezentuje typ obiektów klucza a V typ obiektów wartości.
- Wybrane metody:
  - **clear**() - usuwa wszystkie pary mapy,
  - **containsKey**(key) - wraca *true* jeśli podany klucz istnieje w mapie,
  - **containsValue**(value) – zwraca *true* jeśli mapa zawiera podaną wartość,

Interfejs Map c.d.

- Wybrane metody c.d.:
  - **entrySet**() - zwraca zbiór par mapy w postaci Map.Entry,
  - **get**(key) – zwraca wartość powiązaną z kluczem,
  - **equals**(obj) – zwraca true jeśli jako obj została podana mapa o takich samych parach
  - **forEach**(BiConsumer) – wykonuje akcję przekazaną jako wyrażenie lambda dla każdej pary mapy.
  - **IsEmpty**() - zwraca true jeśli mapa jest pusta.
  - **keySet**() - zwraca zbiór kluczy mapy.
  - **put**(key,value) – wstawia parę klucz – wartość do mapy, nadpisując wartość jeśli taki klucz już istniał.
  - **putAll**(map) – wstawia wszystkie pary przekazanej mapy

Interfejs Map c.d.

- Wybrane metody c.d.:
  - **remove**(key) – usuwa parę dla wskazanego klucza.
  - **replace**(key,value) – zwraca null jeśli przekazany klucz nie istniał wcześniej w mapie, w p.p. wstawia przekazaną parę i zwraca wartość poprzednią skojarzoną z tym kluczem.
  - **size**() - zwraca rozmiar mapy
  - **values**() - zwraca kolekcję zbudowaną z obiektów wartości mapy.
  - **of**() - zwraca niemodyfikowalną mapę złożoną z przekazanych parametrów.
- Mapy pomimo iż znajdują się w pakiecie Collections same nie są kolekcjami, gdyż nie implementują interfejsu Collection.

Interfejs SortedMap

- Zapewnia przechowywanie wpisów w porządku rosnącym na bazie kluczy.
- Rozszerza interfejs Map.
- Interfejs sparametryzowany w postaci: interface SortedMap<K,V> gdzie K określa typ klucza a V typ wartości.
- Metody zadeklarowane:
  - **comparator**() - zwraca komparator używany to porównywania kluczy
  - **firstKey**() - zwraca pierwszy klucz danej mapy
  - **lastKey**() - zwraca ostatni klucz danej mapy
  - **headMap**(end) – zwraca posortowaną mapę, której klucze są mniejsze niż przekazany klucz *end*.

Interfejs SortedMap c.d.

- Metody zadeklarowane c.d.:
  - **subMap**(start,end) – zwraca posortowaną mapę, której klucze są większe lub równe od klucza *start* oraz mniejsze od *end*.
  - **tailMap**(start) – zwraca posortowaną mapę, której klucze są większe lub równe od klucza *start*.
- Wybrane metody mapy mogą generować wyjątki:
  - *ClassCastException* – jeśli przekazany typ argumentu jest niezgodny z zadeklarowanym.
  - *NullPointerException* - gdy przekazana jest wartość null a dany typ mapy nie zezwala na to.
  - IllegalArgrumentException – jeżeli zostanie użyty nieprawidłowy argument.

Interfejs NavigableMap

- Definiuje zachowanie mapy obsługującej wyszukiwanie wpisów na bazie najbliższego dopasowania do danego klucza lub kluczy.
- Rozszerza interfejs SortedMap.
- Interfejs sparametryzowany w postaci: interface NavigableMap<K,V> gdzie K definiuje typ klucza a V typ dla wartości.
- Wybrane metody:
  - **ceilingEntry**(key) – wyszukuje w mapie najmniejszy klucz większy lub równy przekazanemu kluczowi. W przypadku znalezienia metoda zwraca cały wpis (parę), w p.p. zwraca null.
  - **ceilingKey**(key)- wyszukuje w mapie najmniejszy klucz większy lub równy przekazanemu kluczowi. W przypadku znalezienia zwraca klucz w p.p. null.

Interfejs NavigableMap c.d.

- Wybrane metody c.d:
  - **descendingKeySet**() - zwraca zbiór typu NavigableSet zawierający klucze danej mapy posortowane malejąco. Zawiera referencje do tych samych obiektów co mapa.
  - **descendingMap**() - zwraca mapę typu NavigableMap zawierającą elementy posortowane po kluczach malejąco. Zawiera referencje do tych samych obiektów.
  - **firstEntry**() - zwraca pierwszy wpis (parę) mapy (z najmniejszym kluczem).
  - **floorEntry**(key) - wyszukuje w mapie największy klucz mniejszy lub równy przekazanemu w argumencie. W przypadku odnalezienia zwraca wpis (parę) dla tego klucza, w p.p. zwraca null.

Interfejs NavigableMap c.d.

- Wybrane metody c.d:
  - **floorKey**(key) - wyszukuje w mapie największy klucz mniejszy lub równy przekazanemu w argumencie i zwraca go lub null jeśli nie istnieje.
  - **higherEntry**(key) -wyszukuje największy klucz większy niż przekazany w argumencie. W przypadku znalezienia, zwraca dla niego wpis (parę), w p.p. wartość null.
  - **lastEntry**() - zwraca ostatni wpis (parę) mapy.
  - **lowerEntry**(key) – wyszukuje w mapie największy klucz mniejszy od przekazanego w argumencie i zwraca dla niego wpis lub null jeśli klucz nie został odnaleziony.
  - **lowerKey**() - wyszukuje w mapie największy klucz mniejszy od przekazanego w argumencie i zwraca go lub null.

Interfejs NavigableMap c.d.

- Wybrane metody c.d:
  - **navigableKeySet**() - zwraca zbiór typu NavigableSet dla kluczy danej mapy. Zawiera referencje do tych samych obiektów.
  - **pollFirstEntry**() - zwraca i usuwa pierwszy wpis (parę) w danej mapie.
  - **pollLastEntry**() - zwraca i usuwa ostatni wpis (parę) w danej mapie.

Interfejs Map.Entry

- Reprezentuje pojedynczy wpis przechowywany w mapie (parę klucz wartość).
- Interfejs sparametryzowany w postaci: interface Map.Entry<K,V>, gdzie K odpowiada za typ klucza a V za typ wartości.
- Wybrane metody:
  - **equals**(obj) – zwraca true jeśli przekazany w argumencie wpis Map.Entry posiada identyczny klucz i wartość.
  - **getKey**()- zwraca klucz wpisu mapy.
  - **getValue**() - zwraca wartość wpisu mapy.
  - **setValue**(value) – ustawia wartość wpisu mapy.

Klasa HashMap

- Klasa implementująca interfejs Map i jego metody. Używa tablicy mieszającej do przechowywania wpisów co gwarantuje stały czas wykonania metody pobierającej i wstawiającej wpisy nawet dla dużych zbiorów danych lecz nie zapewnia zachowania kolejności wpisów w mapie.
- Stanowi klasę pochodną klasy AbstractMap.
- Klasa sparametryzowana w postaci: class HashMap<K,V> gdzie K stanowi typ kluczy a V typ wartości.
- Posiada 4 konstruktory: HashMap(), HashMap(map), HashMap(size), HashMap(size, fillRatio), gdzie odpowiednio: pierwszy tworzy pustą mapę, drugi inicjalizuje przekazaną mapą w argumencie, trzeci inicjalizuje pojemność a czwarty dodatkowo współczynnik wypełnienia. Wartości domyślne tych 2 parametrów to 16 i 0.75.

Przykład

```
@Test
    public void hashMapTest() {
        //Tworzy pustą mapę
        Map<String,Double> mapSalaries = new HashMap<String,Double>();
        //Wstwia pary do mapy
        mapSalaries.put("Kowalski Jan", 1000.00);
        mapSalaries.put("Nowak Jan", 1100.00);
        mapSalaries.put("Nowakowska Janina", 1200.00);
        //Pobranie zbioru par
        Set<Map.Entry<String,Double>> setEntries = mapSalaries.entrySet();
        for(Map.Entry<String,Double> entry:setEntries) {
            System.out.println(entry.getKey()+":"+entry.getValue());
        }
        assertTrue(mapSalaries.containsKey("Nowak Jan"));
        assertTrue(mapSalaries.containsValue(1000.00));
        //Usuwa parę o podanym kluczu
        mapSalaries.remove("Kowalski Jan");
        assertFalse(mapSalaries.containsKey("Kowalski Jan"));
        //Nadpisuje parę
        mapSalaries.put("Nowakowska Janina", 2000.00);
        assertEquals(2000,mapSalaries.get("Nowakowska Janina"),0.0);
        //Czyści całą mapę
        mapSalaries.clear();
        //Sprwdza czy mapa jest pusta
        assertTrue(mapSalaries.isEmpty());
    }
```

Klasa TreeMap

- Mapa zapewniająca porządek rosnący dla przechowywania kluczy.
- Używa struktury drzewiastej do przechowywania wpisów, co gwarantuje wydajną metodę odczytywania wartości.
- Klasa sparametryzowana w postaci: class TreeMap<K,V>
- Implementuje interfejs NavigableMap
- Stanowi klasę pochodną dla AbstractMap.
- Dostarcza 4 konstruktorów: TreeMap(), TreeMap(comp), TreeMap(map), TreeMap(sortedMap).
- Umożliwia sterowanie porządkiem poprzez przyjmowanie komparatora.
- Jeśli brak przekazanego komparatora wykorzystywany jest porządek naturalny.

Przykład

```
@Test
public void treeMapTest() {
	// Tworzy pustą mapę
	Map<String, Double> mapSalaries = new TreeMap<String, Double>();
	// Wstwia pary do mapy
	mapSalaries.put("Nowakowska Janina", 1200.00);
	mapSalaries.put("Kowalski Jan", 1000.00);
	mapSalaries.put("Testowy Jan", 1100.00);
	// Pobranie zbioru par
	Set<Map.Entry<String, Double>> setEntries = mapSalaries.entrySet();
	//Wpisy posortowane w porzadku naturalnym kluczy
	for (Map.Entry<String, Double> entry : setEntries) {
		System.out.println(entry.getKey() + ":" + entry.getValue()); }
	//Utworzenie 2 mapy z porządkiem odwrotnym
	Map<String, Double> mapSalaries2 = new TreeMap<String, Double>(Comparator.reverseOrder());
	mapSalaries2.putAll(mapSalaries);
	StringBuilder sb = new StringBuilder();
	mapSalaries2.forEach((n,m)-> sb.append(n).append(":").append(m).append("\n"));
	System.out.println("-----------------");
	System.out.println(sb.toString());
	//Utworzenie komparatora działającego tylko na wycinku klucza: imieniu
	Comparator<String> nameDesc = (a,b) ->{
		String[] arrA = a.split(" "); String[] arrB = b.split(" ");
		return arrA[arrA.length-1].compareTo(arrB[arrB.length-1]); };
	//utworzenie 3 mapy
	Map<String, Double> mapSalaries3 = new TreeMap<String, Double>(nameDesc);
	//Zasilenie 3 elemenatmi z 1 mapy ale UWAGA, dodadzą się tylko 2 elementy
	//i dodatowo ze zmienioną wartością ze względu na komparator!!!
	mapSalaries3.putAll(mapSalaries);
	StringBuilder sb2 = new StringBuilder();
	mapSalaries3.forEach((n,m)-> sb2.append(n).append(":").append(m).append("\n"));
	System.out.println("-----------------");
	System.out.println(sb2.toString());
}
```

Konsola

```
Kowalski Jan:1000.0
Nowakowska Janina:1200.0
Testowy Jan:1100.0
-----------------
Testowy Jan:1100.0
Nowakowska Janina:1200.0
Kowalski Jan:1000.0

-----------------
Kowalski Jan:1100.0
Nowakowska Janina:1200.0
```

Klasa LinkedHashMap

- Klasa pochodna dla *HashMap*, rozszerzająca funkcjonalność o listę do przechowywania kolejności przekazywanych elementów.
- Mapa daje możliwość iterowania zgodnie z kolejnością wstawiania wpisów (par) poprzez użycie widoku kolekcji tej klasy.
- Możliwe jest również uzyskanie dostępu do wpisów zgodnie z kolejnością ostatniego dostępu do nich.
- Klasa sparametryzowana w postaci: class LinkedHashMap<K,V>
- Dostarcza 5 konstruktorów gdzie 4 są analogicznie jak dla HashMapy z 5 (*LinkedHashMap(size,fillRatio, order)*) umożliwia przekazanie flagi odnośnie kolejności elementów zgodnie z ostatnim dostępem.
- Definiuje jedną własną metodę removeEldestEntry(entry) wykorzystywaną przez *put()* i *putAll()*.

Przykład

```
@Test
public void linkedHashMapTest() {
      Consumer<String> console = (n) -> System.out.println(n);
      // Tworzy pustą mapę
      Map<String, Double> mapSalaries = new LinkedHashMap<String, Double>();
      // Wstwia pary do mapy
      mapSalaries.put("Nowakowska Janina", 1200.00);
      mapSalaries.put("Kowalski Jan", 1000.00);
      mapSalaries.put("Testowy Jan", 1100.00);
      // Pobranie zbioru par
      Set<Map.Entry<String, Double>> setEntries = mapSalaries.entrySet();
      // Wpisy z zachowaniem kolejności wstawiania
      for (Map.Entry<String, Double> entry : setEntries) {
            System.out.println(entry.getKey() + ":" + entry.getValue());
      }
      Set<String> keys = mapSalaries.keySet();
     assertEquals("[Nowakowska Janina, Kowalski Jan, Testowy Jan]", keys.toString());
      Map<String, Double> mapSalaries2 = new LinkedHashMap<String, Double>(100,0.75f,true);
      mapSalaries2.putAll(mapSalaries);
      Double sallary = mapSalaries2.get("Kowalski Jan");
      console.accept("----------------");
      console.accept("sallary="+sallary);
      console.accept("----------------");
      Set<Map.Entry<String, Double>> setEntries2 = mapSalaries2.entrySet();
      // Kolejność odwrotna do ostaniego dostępu
      for (Map.Entry<String, Double> entry : setEntries2) {
            console.accept(entry.getKey() + ":" + entry.getValue());
      }
      Set<String> keys2 = mapSalaries2.keySet();
    assertEquals("[Nowakowska Janina, Testowy Jan, Kowalski Jan]", keys2.toString());
}
```

Konsola

```
Nowakowska Janina:1200.0
Kowalski Jan:1000.0
Testowy Jan:1100.0
----------------
sallary=1000.0
----------------
Nowakowska Janina:1200.0
Testowy Jan:1100.0
Kowalski Jan:1000.0
```

Klasa IdentityHashMap

- Klasa implementująca mapę analogiczną jak *HashMap* lecz z tą różnicą, że do porównywania używane są bezpośrednio referencje co powoduje inne jej zachowanie.
- Rozszerza klasę *AbstractMap* i implementuje interfejs *Map*.
- Ten typ mapy wykorzystywany jest sporadycznie w wyjątkowych sytuacjach wymagających tej specyficznej cechy.
- Klasa sparametryzowana w postaci: *class IdentityHashMap<K,V>*

Przykład

```
@Test
public void identityHashMapTest() {
      Consumer<String> console = (n) -> System.out.println(n);
      // Tworzy pustą mapę
      Map<String, Double> mapSalaries = new IdentityHashMap<String, Double>();
      // Wstwia pary do mapy na bazie referecnji!!!
      mapSalaries.put(new String("Nowakowska Janina"), 1200.00);
      mapSalaries.put(new String("Kowalski Jan"), 1000.00);
      mapSalaries.put(new String("Testowy Jan"), 1100.00);
      mapSalaries.put(new String("Testowy Jan"), 1150.00);
      Set<Map.Entry<String, Double>> setEntries = mapSalaries.entrySet();
      for (Map.Entry<String, Double> entry : setEntries) {
            console.accept(entry.getKey() + ":" + entry.getValue());
      }
      assertEquals(4,mapSalaries.keySet().size());
      console.accept("-----------------");
      //Utworzenie zwykłej hashmapy juz po wartościahc kluczy eliminuje duplikaty wartości!!
      Map<String, Double> mapSalaries2 = new HashMap<String, Double>(mapSalaries);
      for (Map.Entry<String, Double> entry : mapSalaries2.entrySet()) {
            console.accept(entry.getKey() + ":" + entry.getValue());
      }
      assertEquals(3,mapSalaries2.keySet().size());
}
```

Konsola

```
Kowalski Jan:1000.0
Nowakowska Janina:1200.0
Testowy Jan:1150.0
Testowy Jan:1100.0
-----------------
Nowakowska Janina:1200.0
Testowy Jan:1100.0
Kowalski Jan:1000.0
```

Klasa EnumMap

- Mapa przystosowana do pracy z kluczami w postaci stałych wyliczeniowych.
- Klasa rozszerza klasę AbstracMap oraz implementuje interfejs Map.
- Klasa sparametryzowana w postaci: *class EnumMap<K extends Enum<K>. V>*, gdzie K oznacza typ klucza rozszerzający Enum<K> co powoduje, iż musi to być stała wyliczeniowa.
- Klasa nie definiuje żadnych swoich metod.
- Dostarcza 3 konstruktory: pierwszy tworzący pustą mapę, oraz 2 inicjalizujące mapę inną mapą.

Przykład

```
@Test
public void enumMapTest() {
      Consumer<String> console = (n) -> System.out.println(n);


      // Tworzy pustą mapę
      Map<EnType, Double> mapRoles = new EnumMap<EnType, Double>(EnType.class);
      mapRoles.put(EnType.admin,10d);
      mapRoles.put(EnType.publisher,20d);
      mapRoles.put(EnType.user,30d);
      mapRoles.forEach((n,m)->console.accept(n+":"+m));
      assertTrue(mapRoles.size()==3);
      assertTrue(mapRoles.containsKey(EnType.publisher));
      assertTrue(mapRoles.containsKey(EnType.user));
      assertTrue(mapRoles.containsKey(EnType.admin));

}
```

Wyższa Szkoła Informatyki Stosowanej i Zarządzania
pod auspicjami Polskiej Akademii Nauk
WYDZIAŁ INFORMATYKI

Część 2 – Obsługa sieci i elementy programowania klient - serwer

java.net - komunikacja sieciowa

- Standardowy pakiet zawierający narzędzia do obsługi operacji sieciowych w JAVA.
- Obsługują one protokół TCP/IP poprzez odpowiednie rozszerzenia interfejsu operacji wejścia-wyjścia opartego na strumieniach.
- Najważniejsze klasy pakietu:
  - InetAddress
  - Socket
  - ServerSocket
  - URL
  - URLConnection
  - HttpURLConnection

Klasa InetAddress

- Reprezentuje adresy zarówno w postaci numerycznych adresów IP jak również nazw domenowych.
- Klasa może używać zarówno adresacji IPv4 jak i IPv6.
- Klasa nie posiada konstruktorów udostępnionych dla programisty.
- Obiekt powoływany jest poprzez metody wytwórcze tej klasy. Wybrane z nich to:
  - getLocalHost(),
  - getByName(hostName),
  - getAllByName(hostName),
  - getByAddress(ip).
- Metody w przypadku nie odnalezienia adresu mogą wygenerować wyjątek UnkownHostException.

Programowanie klient – serwer w JAVA

- Dostępnych jest wiele modeli do programowania klient serwer:
  - Programowanie bazujące na gniazdach (peer to peer)
  - Programowanie z użyciem zdalnego wywoływania metod (RMI)
  - Programowanie z użyciem serwera aplikacji oraz klientów (np.. REST serwer oraz klient JS)
- Programowanie aplikacji trójwarstwowych w JAVA
  - Programowanie z użycie serwera WWW oraz cienkiego klienta HTML np.. Tomcat i JSP lub JSF
  - Istnieje wiele szkieletów udostępniających programowanie aplikacji trójwarstwowych:
    - SpringMVC
    - STRUTS
    - Vaadin

Gniazda klientów

- Mają zastosowanie do dwukierunkowych, trwałych, opartych na strumieniach połączeniach typu punkt-punkt pomiędzy komputerami w internecie.
- Reprezentowane poprzez klasę Socket, zaprojektowaną z myślą o nawiązywaniu połączeń z gniazdami serwerów.
- Utworzenie obiektu powoduje niejawne nawiązanie połączenia między klientem a serwerem.
- Konstruktory przyjmują adres serwera w różnych postaciach.
- Dostępne są metody, za pomogą których można uzyskać informacje o adresie i porcie połączenia: getInetAddress(), getPort(), getLocalPort
- Metody getInputStream() i getOutputStream udostępniają strumienie wejściowe i wyjściowe.

Gniazda serwerów

- Aplikacje serwerów budujemy na bazie klasy ServerSocket.
- Służy do tworzenia serwerów nasłuchujących na zadanych portach, w oczekiwaniu na połączenie klienta lokalnego lub zdalnego.
- W momencie rejestracji klienta, odpowiedni mechanizm automatycznie rejestruje w systemie gniazdo oczekujące na połączenia klientów.
- Konstruktory tej klasy przyjmują w argumencie nr portu oraz opcjonalnie długość kolejki żądań (domyślnie ustawioną na 50). Po przekroczeniu ustawionej liczby, kolejne żądania będą otrzymywały odmowę.
- Klasa ServerSocket definiuje metodę accept(), której wywołanie blokuje aplikację aż do momentu zainicjowania przez klienta komunikacji.

Przykład

```
public class MyServer {
private int portNumber;
public MyServer(int portNumber) throws IOException {
      this.portNumber=portNumber;
      startServer();
}
private void startServer() throws IOException {
      try (
                  ServerSocket serverSocket = new ServerSocket(portNumber);
                  Socket clientSocket = serverSocket.accept();
                  PrintWriter out =
                      new PrintWriter(clientSocket.getOutputStream(), true);
                  BufferedReader in = new BufferedReader(
                      new InputStreamReader(clientSocket.getInputStream()));

            ){
            System.out.println("Server listen on port:"+portNumber);
                  String inputLine, outputLine;
            while ((inputLine = in.readLine()) != null) {
                  System.out.println(inputLine);
            if (inputLine.equals("STOP")) {
                  break;
         }
       }

      }
}
}
```

```
@Test
      public void clientHelloTest() throws
UnknownHostException, IOException,
InterruptedException {
try(Socket client = new
Socket("localhost",7777)){

PrintWriter out = new
PrintWriter(client.getOutputStream(), true);


                  out.println("Hello world");
                  out.println("Ala ma kota");
                  Thread.sleep(500);
                  out.println("Ala ma kota2");

            }
      }

Konsola:
Server listen on port:7777
Hello world
Ala ma kota
Ala ma kota2
```

Klasa URL

- URL – ang. Uniform Resource Locator, ustandaryzowany format adresowania zasobów w internecie.
- Składa się z czterech składników:
  - Protokołu (http, ftp, gopher, file)
  - Nazwa lub adres ip
  - Numer portu
  - Ścieżka do pliku
- Klasa URL definiuje wiele konstruktorów, ale najpopularniejszy przyjmuje łańcuch znaków z adresem.
- Po utworzeniu obiektu dostępne są metody pobierające poszczególne wartości składowe: getProtocol(), getPort(), getHost(), getFile().

Przykład

```
@Test
	public void urlSimpleTest() {
		Consumer<String> console = (n)->System.out.println(n);
		try {
			URL url = new URL("https://www.google.pl/index.html");
			console.accept("Protokół:"+url.getProtocol());
			console.accept("Port:"+url.getPort());
			console.accept("Host:"+url.getHost());
			console.accept("Plik:"+url.getFile());
			console.accept("Pełen adres:"+url.toExternalForm());
			assertEquals("https",url.getProtocol());
			assertEquals(-1,url.getPort());
			assertEquals("www.google.pl",url.getHost());

		} catch (MalformedURLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
```

Konsola

```
Protokół:https
Port:-1
Host:www.google.pl
Plik:/index.html
Pełen adres:https://www.google.pl/index.html
```

Klasa URLConnection

- Uniwersalna klasa zwracająca dostęp do atrybutów zdalnego dostępu po nawiązaniu połączenia, lecz przed jego fizycznym transferem na lokalny komputer (poprzez protokół HTTP).
- Ważna część funkcjonalności udostępnionej przez klasę dotyczy nagłówka składającego się z par (klucz – wartość) w postaci łańcuchów znaków.
- Metoda *getHeaderField(key)* zwraca wartość konkretnego klucz nagłówka, natomiast *getHeaderFields* zwraca strukturę mapy nagłówka (jego wszystkie pola).
- Metoda *openConnection()* klasy URL zwraca obiekt klasy *UrlConnection*, dzięki któremu można dokonać analizy przedstawionych zasobów.

Przykład

```
@Test
public void urlConnectionTest() {
Consumer<String> console = (n)->System.out.println(n);
URL url;
try {
      url = new URL("https://www.interia.pl");
      URLConnection urlCon = url.openConnection();
      long answerDate = urlCon.getDate();
      if(answerDate!=0)
            console.accept("Data odpowiedzi = "+(new Date(answerDate)).toString());

      console.accept("typ zawartości="+urlCon.getContentType());
      urlCon.getExpiration();
      console.accept("Data ostatniej modyfikaji = "
+new Date(urlCon.getLastModified()).toString());
      console.accept("Wielkość zasobu="+urlCon.getContentLengthLong());
      Map<String,List<String>> mapHeader = urlCon.getHeaderFields();
      mapHeader.forEach((k,v)->console.accept(k+":"+v.toString()));
} catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}

}
```

Konsola

```
Data odpowiedzi = Wed May 24 17:57:25
CEST 2021
typ zawartości=text/html;
charset=utf-8
Data ostatniej modyfikaji = Wed May
24 17:56:43 CEST 2021
Wielkość zasobu=252894
Accept-Ranges:[bytes]
Keep-Alive:[timeout=20]
null:[HTTP/1.1 200 OK]
Server:[nginx]
ETag:["60ae6fbb-3dbde"]
Connection:[keep-alive]
Set-Cookie:
[inpl_mobile=d;Domain=.interia.pl;Max
-Age=260000]
Last-Modified:[Wed, 26 May 2021
15:56:43 GMT]
Content-Length:[252894]
Date:[Wed, 24 May 2021 15:57:25 GMT]
Content-Type:[text/html; charset=utf-
8]
```

Klasa HttpURLConnection

- Dedykowana klasa do obsługi połączeń HTTP, będąca klasą pochodną URLConnection.
- Obiekt tej klasy uzyskujemy poprzez wywołanie metody openConnection klasy URL a jej wynik rzutujemy na zmienną tej klasy.
- Klasa dostarcza wiele dodatkowych metod. Wybrane z nich:
  - getFollowRedirects – zwraca true dla przekierowań włączonych
  - getRequestMethod – pobiera typ żądania (GET,POST)
  - getResponseCode – kod odpowiedzi HTTP lub -1 jeśli brak
  - getResponseMessage – komunikat odpowiedzi lub null
  - setRequestMethod(how) – ustawianie typu żądania protokołu HTTP. Domyślnie jest ustawione GET.

Przykład

```
@Test
public void httpUrlConnectionTest() {
Consumer<String> console = (n)->System.out.println(n);
URL url;
try {
      url = new URL("https://www.interia.pl");
      HttpURLConnection urlCon = (HttpURLConnection) url.openConnection();


      console.accept("typ żądania="+urlCon.getRequestMethod());
      console.accept("kod odpowiedzi= "+urlCon.getResponseCode());
      console.accept("komunikat odpowiedzi="+urlCon.getResponseMessage());

} catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}

}
```

Konsola

```
typ żądania=GET
kod odpowiedzi= 200
komunikat odpowiedzi=OK
```

Pakiet java.net.http

- Od JDK11 dostępne jest API klienta HTTP.
- Rozszerza możliwości dla wielu typów operacji sieciowych względem tradycyjnej obsługi z użyciem java.net .
- Zapewnia wsparcie dla operacji asynchronicznych
- Wspiera protokół HTTP/2
- Obsługuje dwukierunkowy protokół WebSocket
- Zapewnia ulepszony i łatwiejszy dostęp i obsługę tradycyjnych funkcjonalności starego pakietu.
- Pakiet zawiera trzy kluczowe elementy:
  - Klasa *HttpClient*
  - Klasa *HttpRequest*
  - Interfejs *HttpResponse*

Klasa HttpClient

- Klasa hermetyzująca mechanizm wysyłania żądań i odbierania odpowiedzi protokołu HTTP.
- Obsługuje komunikację synchroniczną jak i asynchroniczną.
- Klasa abstrakcyjna, do tworzenia obiektów przygotowane są specjalne metody wytwórcze tworzące tzw. buildery, które to tworzą wymagany obiekt, np..
  HttpClient myClient = HttpClient.newBuilder().build();
- Interfejs *HttpClient.Builder* definiuje klika metod, za pomocą których można konfigurować buildery, przed wytworzeniem obiektu.
- Dla domyślnej konfiguracji dostępna jest metoda tworząca nową instancję obiektu: *newHttpClient()*
- Klasa definiuje metodę *send()* wysyłającą żądanie synchroniczne

Klasa HttpRequest

- Klasa abstrakcyjna do reprezentowania żądań HTTP.
- Do tworzenia obiektów tej klasy służą tzw. buildery, które udostępniane są poprzez metodę newBuilder() i newBuilder(uri).
- HttpRequest.Builder umożliwia określenie różnych aspektów żądania, np.. metody wykonania (GET,POST), ustalenie zawartości nagłówka, wersję protokołu HTTP, itd..
- W celu utworzenia żądania należy wywołać metodę build() na pobranym obiekcie buildera.
- Posiadając instancję klasy HttpRequest, można jej użyć w metodzie send() klasy HttpClient.

Interfejs HttpResponse

- Odpowiedzi HTTP są reprezentowane przez obiekty klas implementujących interfejs HttpResponse<T>.
- T określa typ odpowiedzi np. InputStream, String, File, etc.
- Odpowiedzi są obsługiwane przez klasy implementujące interfejs HttpResponse.BodyHandler
- Kod statusu odpowiedzi można pobrać poprzez użycie metody statusCode, gdzie kody zwrócone są kodami odpowiedzi protokołu HTTP, np.. 200 – ok, 403 – permission denied, etc.
- Za pomocą metody headers() można pobrać nagłówki odpowiedzi, które można pobrać w postaci mapy przy użcyciu metody map() zwracającej Map<String,List<String>>.

Przykład

```
@Test
public void httpClientTest() {
Consumer<String> console = (n)->System.out.println(n);

HttpClient myClient = HttpClient.newHttpClient();

try {
      HttpRequest req = HttpRequest
.newBuilder(new URI("https://www.interia.pl/")).build();
      HttpResponse<InputStream> resp =
myClient.send(req, HttpResponse.BodyHandlers.ofInputStream());
      console.accept("Status odpowiedzi:"+resp.statusCode());
      console.accept("Metoda żądania:"+req.method());
      HttpHeaders headers = resp.headers();
      Map<String,List<String>> mapHeaders =  headers.map();
      for(Map.Entry<String,List<String>> entry:mapHeaders.entrySet())
            console.accept(entry.getKey()+"->"+entry.getValue().toString());
      StringBuilder sbContent = new StringBuilder();
      InputStream input = resp.body();
      int i;
      while((i = input.read())!=-1) {
            sbContent.append((char)i);
      }

      console.accept("body length="+sbContent.length());
      console.accept("body="+sbContent.toString());
} catch (URISyntaxException | IOException | InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}
}
```

Konsola

```
Status odpowiedzi:200
Metoda żądania:GET
:status->[200]
accept-ranges->[bytes]
content-length->[256082]
content-type->[text/html; charset=utf-8]
date->[Wed, 24 May 2021 16:18:54 GMT]
etag->["60ae74c4-3e852"]
last-modified->[Wed, 24 May 2021 16:18:12
GMT]
server->[nginx]
set-cookie-
>[inpl_mobile=d;Domain=.interia.pl;Max-
Age=260000]
body length=256082
Body= … (cała zawartość html)...
```

Podsumowanie

- Wprowadzenie do map
- Interfejsy map
- Klasy implementującej map
- Przykłady różnych rodzajów map
- Klasy adresów sieciowych
- Gniazda serwerowe
- Gniazda klienckie
- Obsługa żądań HTTP
- Elementy java.net.http

Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Dziękuję za uwagę!