Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Język Java – wykład 5

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

www.lsosnowski.pl

Część 1 – Obsługa wejścia i wyjścia

Strumienie wejścia i wyjścia

- Java rozróżnia strumienie bajtowe i znakowe. Istnieją oddzielne hierarchie klas dla obu typu strumieni.
- Strumienie bajtowe definiowane są przez dwie hierarchie klas, na których szczycie są dwie klasy: *InputStream* i *OutputStream*.
- Klasa *InputStream* definiuje wspólną charakterystykę dla wszystkich strumieni bajtowych wejścia.
- Klasa *OutputStream* definiuje wspólną charakterystykę dla wszystkich strumieni bajtowych wyjścia.
- Klasa Reader stanowi szczyt hierarchii klas dotyczących obsługi wejścia dla strumieni znakowych.
- Klasa Writer stanowi szczyt hierarchii klas dotyczących obsługi wyjścia dla strumieni znakowych.

Klasy strumieni bajtowych

Klasa strumienia	Opis
BufferedInputStream	Buforowany strumień wejścia
BufferedOutputStream	Buforowany strumień wyjścia
ByteArrayInputStream	Strumień wejścia odczytujący tablicę bajtów
ByteArrayOutputStream	Strumień wyjścia zapisujący tablicę bajtów
DataInputStream	Strumień wejścia zawierający metody standardowych typów danych
DataOutputStream	Strumień wyjścia zawierający metody standardowych typów danych
FileInputStream	Strumień wejścia odczytujący pliki
FileOutputStream	Strumień wyjścia zapisujący pliki
FilterInputStream	Klasa zapewniająca implementację dla InputStrem
FilterOutputStream	Klasa zapewniająca implementację dla OutputStream
InputStream	Klasa abstrakcyjna definiująca strumień wejściowy

Klasy strumieni bajtowych c.d.

Klasa strumienia	Opis
ObjectInputStream	Strumień wejściowy dla obiektów
ObjectOutputStream	Strumień wyjściowy dla obiektów
OutputStream	Klasa abstrakcyjna definiująca operacje strumienia wyjściowego
PipedInputStream	Potok wejścia
PipedOutputStream	Potok wyjścia
PrintStream	Strumień wyjścia zawierający metody println(), itp
PushbackInputStream	Strumień wejściowy umożliwiający zwracanie bajtów do innego strumienia
SequencaInputStream	Strumień wejścia stanowiący kombinację dwu lub więcej strumieniu wejścia, czytanym bezpośrednio po sobie

Klasy strumieni znakowych

Klasa strumienia	Opis
BufferedReader	Buforowany znakowy strumień wejścia
BufferedWriter	Buforowany znakowy strumień wyjścia
CharArrayReader	Strumień wejścia umożliwiający odczyt tablicy znaków
CharArrayWriter	Strumień wyjścia umożliwiający zapis tablicy znaków
FileReader	Strumień wejścia umożliwiający odczyt pliku
FileWriter	Strumień wyjścia umożliwiający zapis pliku
FilterReader	Filtrowany strumień wejścia
FilterWriter	Filtrowany strumień wyjścia
InputStreamReader	Strumień wejścia tłumaczący bajty na znaki
LineNumberReader	Strumień wejścia zliczający linie
OutputStreamWriter	Strumień wyjścia tłumaczący znaki na bajty

Klasy strumieni znakowych c.d.

Klasa strumienia	Opis
PipedReader	Potok wejścia
PipedWriter	Potok wyjścia
PrintWriter	Strumień wyjścia zawierający metody print, println
PushbackReader	Strumień wejściowy umożliwiający zwracanie bajtów do innego strumienia
Reader	Klasa abstrakcyjna definiująca znakowy strumień wejścia
StringReader	Strumień wejścia umożliwiający odczyt łańcucha znaków
StringWriter	Strumień wyjścia umożliwiający zapis łańcucha znaków
Writer	Klasa abstrakcyjna definiująca znakowy strumień wyjścia

Odczyt i zapis plików strumieniami bajtowymi

- Do odczytu i zapisu plików używamy strumieni bajtowych odpowiednio *FileInputStream* i *FileOutputStream.*
- W celu odczytu danych z pliku tworzymy obiekt klasy *FileInputStream*, podając w argumencie ścieżkę do pliku.
- Jeśli plik nie istnieje zostanie wygenerowany wyjątek *FileNotFoundException.*
- Do odczytu danych używamy metody *read*(), której każde wywołanie powoduje wczytanie jednego bajtu danych z pliku. Metoda zwróci wartość -1 po napotkaniu końca pliku.
- Po zakończeniu odczytu danych zasób plikowy należy zwolnić poprzez jego zamknięcie z użyciem metody *close*().

Odczyt i zapis plików strumieniami bajtowymi

- W celu zapisu danych do pliku tworzymy obiekt klasy *FileOutputStream*, podając w argumencie ścieżkę do pliku i opcjonalnie dodatkowo włącza tryb dopisywania (poprzez wywołanie odpowiednich przeciążonych konstruktorów). W przypadku konstruktora wywołanego tylko ze ścieżką do pliku, plik jest tworzony (nadpisuje ewentualnie już istniejący). Drugi konstruktor powoduje dopisywanie na końcu pliku.
- Do zapisu pojedynczego bajtu danych używamy metody *write(int bajt)*
- Po zakończeniu zapisu należy zwolnić zasób plikowy z użyciem metody *close*();
- Przy tworzeniu obiektu może być wygenerowany wyjątek *FileNotFoundException.*

Przykład odczytu z pliku

```
@Test
public void readFileTest() {
	FileInputStream fIn=null;
	String filePath = "./src/test/resources/plik1.txt";
	StringBuilder sB = new StringBuilder();
	int i;
	try {
		fIn = new FileInputStream(filePath);
		do {
			i = fIn.read();
			if(i!=-1)
				sB.append((char)i);
		}while(i!=-1);
	}catch(IOException e) {
		log.error(e,e);
	}
	finally {
		try {
			if(fIn!=null)
				fIn.close();
		}catch(IOException e) {
			log.error(e,e);
		}
	}
	if(sB.isEmpty())
		log.trace("Nic nie odczytano");
	else
		log.trace("Odczytano: ".concat(sB.toString()));
	Assert.assertTrue(sB.toString()
		.startsWith("Lorem ipsum dolor sit amet"));
}
```

Przykład zapisu do pliku

```
@Test
public void writeFileTest() {
	FileOutputStream fOut=null;
	String filePath = "./src/test/resources/plik2.txt";
	StringBuilder sB = new StringBuilder();
	for(int i=0;i<100;i++)
		sB.append("Ala ma kota ("+(i+1)+").\n");

	try {
		fOut = new FileOutputStream(filePath);
		String content = sB.toString();
		for(int i=0;i<content.length();i++)
			fOut.write(content.charAt(i));
	}catch(IOException e) {
		log.error(e,e);
	}
	finally {
		try {
			if (fOut != null)
				fOut.close();
		} catch (IOException e) {
			log.error(e,e);
		}
	}
	File f = new File(filePath);
	assertTrue(f.exists());
}
```

Automatyczne zamykanie pliku

- Od JDK 7 dostępny jest mechanizm automatycznego zarządzania zasobami, który przy użyciu innej wersji instrukcji try może być wykorzystany do uproszczonego zarządzania dostępem do pliku.
- Ogólna postać instrukcji:

```
try(specyfikacja zasobu){
//użycie zasobu
}
```

- Sposób ten może być wykorzystywany dla zasobów implementujących interfejs *AutoCloseable* definiujący metodę *close().*
- Instrukcja *try* tego typu może również zawierać *catch* oraz *finally.*
- Zasób jest zwalniany automatycznie po opuszczeniu bloku *try.*

Obsługa plików z użyciem strumieni znakowych

- Zaletą używania strumieni znakowych jest możliwość bezpośredniego operowania na znakach *UNICODE*.
- W celu czytania zawartości pliku ze znakami używamy klasy *FileReader*.
- W celu zapisywania znaków do pliku używamy klasy *FileWriter*.
- Analogicznie jak dla strumieni bajtowych w konstruktorze podawana jest ścieżka do pliku. W przypadku braku pliku może być wygenerowany stosowny wyjątek.
- Strumień *FileReader* może być łączony z obiektem klasy BufferedReader dzięki któremu dostępne jest wczytywanie po linii a nie po pojedynczym znaku.
- Strumień FileWriter można połaczyć z BufferedWriter zyskując dodatkowe metody (np. newLine()).

Przykład

```
@Test
public void readFileTxtBufferedTest() {
     String filePath =
"./src/test/resources/plik1.txt";
     StringBuilder sB = new StringBuilder();
     String line;
     try (BufferedReader br=
     new BufferedReader(new FileReader(filePath))){
          while((line=br.readLine())!=null) {
               sB.append(line);
          }
     }catch(IOException e) {
          log.error(e,e);
     }
     if(sB.isEmpty())
          log.trace("Nic nie odczytano");
     else
          log.trace("Odczytano:
".concat(sB.toString()));
     Assert.assertTrue(sB.toString()
     .startsWith("Lorem ipsum dolor sit amet"));
}
```

```
@Test
public void writeFileTxtBufferedTest() {
     String filePath =
     "./src/test/resources/plik2.txt";
     StringBuilder sB = new StringBuilder();
     for(int i=0;i<100;i++)
          sB.append("Ala ma kota ("+(i+1)+").\n");

     try(BufferedWriter bw = new BufferedWriter(new
     FileWriter(filePath))) {
          bw.write(sB.toString());
     }catch(IOException e) {
          log.error(e,e);
     }

     File f = new File(filePath);
     assertTrue(f.exists());
}
```

Odczyt i zapis danych binarnych

- Do odczytu i zapisu danych konkretnych typów służy DataInputStream i DataOutputStream.
- Strumień DataOutputStream udostępnia metody: *writeBoolean, writeByte, writeChar, writeDouble, writeFloat, writeInt, writeLong, writeShort.*
- Strumień DataInputStream udostępnia metody: r*eadBoolean, readByte, readChar, readDouble, readFloat, readInt, readLong, readShort.*
- Argumentem konstruktora jest strumień wejścia lub wyjścia (w zależności od operacji) wskazujący na plik: FileInputStream lub FileOutputStream.

Przykład

```
@Test
public void writeFileDataTest() {
     String filePath =
"./src/test/resources/data1.txt";
     String strItem="Ala ma kota";
     int intItem=9;
     double dItem = 8.67;
     boolean bItem = true;

     try(DataOutputStream dataOut = new
      DataOutputStream(new
FileOutputStream(filePath))) {
          dataOut.writeInt(strItem.length());
          dataOut.writeBytes(strItem);
          dataOut.writeUTF(strItem);
          dataOut.writeInt(intItem);
          dataOut.writeDouble(dItem);
          dataOut.writeBoolean(bItem);
     }catch(IOException e) {
          log.error(e,e);
     }

     File f = new File(filePath);
     assertTrue(f.exists());
}
```

```
@Test
public void readFileDataTest() {
      String filePath = "./src/test/resources/data1.txt";
      int strLen=0;
      String strItem=null;
      byte strInBytes[]=null;
      int intItem=0;
      double dItem=0d;
      boolean bItem=false;

      try(DataInputStream dataIn = new DataInputStream(new
       FileInputStream(filePath))) {
            strLen = dataIn.readInt();
            strInBytes=dataIn.readNBytes(strLen);
            strItem = dataIn.readUTF();
            intItem=dataIn.readInt();
            dItem=dataIn.readDouble();
            bItem=dataIn.readBoolean();
      }catch(IOException e) {
            log.error(e,e);
      }
assertEquals(11,strLen);
assertEquals("Ala ma kota",strItem);
assertEquals(9,intItem);
assertEquals(8.67,dItem,0.0);
assertTrue(bItem);
assertArrayEquals("Ala ma kota".getBytes(), strInBytes);
}
```

Pliki o dostępie swobodnym

- Java umożliwia dostęp do danych plikowych nie tylko w dostępie sekwencyjnym. Istnieje możliwość czytania i zapisu w tzw. dostępie swobodnym.
- *RandomAccessFile* jest klasą zapewniającym operacje na plikach w dostępie swobodnym. Nie jest to jednak klasa pochodna ani *InputStream* ani *OutputStream*. Implementuje za to interfejsy *DataInput* i *DataOutput,* które narzucają implementacje odpowiednich metod dla operacji wejścia i wyjścia.
- Tworząc obiekt należy przekazać ścieżkę do pliku oraz tryb dostępu: **r** – tylko odczyt, **rw** – odczyt i zapis, **rws** – odczyt i zapis oraz aktualizacja pliku przy każdej zmianie zawartości lub metadanych pliku, **rwd** – odczyt i zapis oraz aktualizacja pliku przy każdej zmianie zawartości.

Przykład

```
@Test
public void writeAndReadFileRandowmTest() {
     String filePath = "./src/test/resources/data2.txt";
     double data[] = {-2.34,5.65,3.65,7.99,0.98,4.567,0.0};

     try(RandomAccessFile raf = new RandomAccessFile(filePath,"rw")) {
          //Zapis danych do pliku
          for(double d:data)
               raf.writeDouble(d);

          double d;
          //Odczyt danych z pliku

          for(int i=0;i<data.length;i++) {
               raf.seek(8*i);
               d=raf.readDouble();
               log.info(""+d+", ");
          }
     }catch(IOException e) {
          log.error(e,e);
     }
```

Część 2 – Wprowadzenie do wyrażeń lambda

Wyrażenia lambda – podstawowe informacje

- Wyrażenie lambda jest metodą anonimową (metodą bez nazwy). Dostarcza implementację dla metody zdefiniowanej w danym interfejsie funkcyjnym.
- Interfejs funkcyjny to taki który zawiera dokładnie jedną zdefiniowaną metodę abstrakcyjną. Zazwyczaj metoda ta określa przeznaczenie interfejsu. Określa również typ docelowy.
- Wyrażenie lambda pozwala utworzyć klasę anonimową.
- Wyrażenia lambda nazywamy również domknięciami (ang. closure).
- Interfejsy funkcyjne są określane jako typy SAM (ang. Single Abstract Method).
- Operator lambda (->) dzieli wyrażenie lambda na dwie części: lewą określającą parametry i prawą określającą ciało wyrażenia.

Wyrażenia lambda – podstawowe informacje c.d.

- (lista parametrów) -> ciało wyrażenia lambda.
- Ciało wyrażenia może mieć dwie postacie: pojedynczej wyrażenia lub bloku kodu.
- Przykłady prostych wyrażeń lambda:
  ()-> 4.3 - wyrażenie nie pobiera parametrów (ich lista jet pusta), zwraca wartość stałą 4.3.
- (n) -> (n % 10)==0 – wyrażenie przyjmuje jeden parametr, zwraca wartość true lub false w zależności czy przekazany parametr jest podzielny przez 10.
- () -> Math.random() * 50 – wyrażenie bezparametrowe, zwraca obliczenie liczby pseudolowej przemnożonej przez 50.

Wyrażenia lambda – interfejsy funkcyjne

- Metoda abstrakcyjna to taka, która w interfejsie nie będzie zawierała ciała metody. Pomimo braku słowa kluczowego abstract, wszystkie tak zadeklarowane metody są metodami abstrakcyjnymi.
- Przykład:

```
public interface ICalculator {
     double calculate(double x, double y);
}

@Test
public void calculateTest() {
     ICalculator cal = (double a,double b) -> a*b;
     assertEquals(12d,cal.calculate(3d, 4d),0.0d);
     cal = (double a,double b) -> a+b;
     assertEquals(7d,cal.calculate(3d, 4d),0.0d);
     cal = (double a,double b) -> a/b;
     assertEquals(2d,cal.calculate(8d, 4d),0.0d);
     cal = (double a,double b) -> a-b;
     assertEquals(-1.7d,cal.calculate(1.5d, 3.2d),0.00001);
}
```

Blokowe wyrażenia lambda

- Ciało wyrażenia lambda zdefiniowane przez blok kodu nazywamy **ciałem blokowym**. Blok może się składać z wielu instrukcji.
- Składnia blokowych wyrażeń lambda:

```
ZmiennaInterfejsuFunkcyjnego nazwa = (parametry) -> {
                        blok kodu
                    };
```

- Blok kodu zwraca wartość, którą określa metoda abstrakcyjna interfejsu funkcyjnego. Wartość zwracana jest przy użyciu instrukcji *return*.

Przykład:

```
public interface ICalculator {
	double calculate(double x, double y);
}
public interface IComputer {
	double[] compute(double[] arr,double a, double b);
}
@Test
	public void calculateBlockTest() {
		double arrC[] = new double[] {9.99,2.45,4.55,7.76,5.33,8.62,10.001,9,2};
		ICalculator cal = (double a,double b) -> {
			double sum = 0.0;
			for(double d:arrC) {
				sum=sum+d;
			}
			return sum+a-b;
		};
		System.out.println(""+cal.calculate(1d, 0d));
	}
@Test
	public void calculateBlock2Test() {
		double arrC[] = new double[] {9.99,2.45,4.55,7.76,5.33,8.62,10.001,9,2};
		IComputer comp = (double[] arr, double a,double b) -> {
			double arrResult[] = new double[arr.length];
			double sum = 0.0;
			for(int i=0;i<arr.length;i++) {
				arrResult[i] = arr[i]*a-b;
				sum=sum+arrResult[i];
			}
			return arrResult;
		};

		System.out.println(""+Arrays.toString(comp.compute(arrC,1.4d, -0.3d)));
	}
```

Parametryzacja interfejsów funkcyjnych

- Wyrażenia lambda jako takie nie mogą mieć parametryzowanych typów.
- Można jednak stworzyć parametryzowany interfejs funkcyjny, np.

```
public interface ICalculator2<T> {
     T calculate(T x, T y);
}

@Test
public void calculateTTest() {
     ICalculator2<Double> cal = (a,b) -> a*b;
     assertEquals(12d,cal.calculate(3d, 4d),0.0d);
     ICalculator2<Integer> cal2 = (Integer a,Integer b) -> a+b;
     assertEquals(7,cal2.calculate(Integer.valueOf(3), Integer.valueOf(4)).intValue());
     ICalculator2<Float> cal3 = (a, b) -> a/b;
     assertEquals(2f,cal3.calculate(Float.valueOf(8f), Float.valueOf(4f)),0.0f);

}
```

- W ten sposób przy użyciu jednego interfejsu funkcyjnego można obsłużyć wiele wyrażeń lambda zwracające różne typy danych.

Przechwytywanie zmiennych

- Używanie zmiennych lokalnych pochodzących z zasięgu w jakim zostały zdefiniowane wewnątrz wyrażenia lambda nazywamy **przechwyceniem zmiennej (ang. capture variable)**.
- Takie zmienne otrzymują status praktycznie sfinalizowanych (ang. effectively final), co powoduje iż po pierwszym przypisaniu nie można zmienić ich wartości.
- Jednakże wyrażenia lambda mogą używać (łącznie z ich modyfikacjami) zmiennych instancyjnych klasy w której zostały zdefiniowane.
- Przechwycona zmienna lokalna do wyrażenia lambda pozostaje w statusie praktycznie sfinalizowanej zmiennej również poza wyrażeniem lambda.

Przykład:

```
@Test
public void captureVariableTest() {
     double arrC[] = new double[] {9.99,2.45,4.55,7.76,5.33,8.62,10.001,9,2};
     double sum=0.0;
     IComputer comp = (double[] arr, double a,double b) -> {
          double arrResult[] = new double[arr.length];
          for(int i=0;i<arr.length;i++) {
               arrResult[i] = arr[i]*a-b;
               arrC[i]=arr[i]*a-b;//To jest ok, ale baz zmiany rozmiaru tablicy
          //arrC = Arrays.copyOf(arrC,arrC.length+1);  - niedozwolone - utrata statusu zmiennej p. sf.
          //sum=sum+arrResult[i];  - niedozwolone - utrata statusu zmiennej p. sf.
          }
          return arrResult;
     };
     double arrCAssert[] = new double[] {9.99,2.45,4.55,7.76,5.33,8.62,10.001,9,2};
     Assert.assertArrayEquals(arrCAssert,arrC, 0.0);
     System.out.println(""+Arrays.toString(comp.compute(arrC,1.4d, -0.3d)));
     try {
          Assert.assertArrayEquals(arrCAssert,arrC, 0.0);
          fail();
    } catch (AssertionError e) {
        return;
   }
}
```

Zgłaszanie wyjątków w wyrażeniach lambda

- Wyrażenia lambda mogą zgłaszać wyjątki.
- Można stosować zarówno instrukcję try catch jak również generować wyjątek, który będzie przekazywany do obsługi w innym miejscu.
- W celu umożliwienia generowania wyjątku odpowiednia deklaracja wyjątków, musi się znaleźć w interfejsie funkcyjnym przy metodzie abstrakcyjnej. Do tego celu używamy słowa kluczowego *throws*.

```
public interface IComputer2 {
     double[] compute(double[] arr,double a, double b) throws Exception;
}
```

Przykład:

```
public interface IComputer2 {
	double[] compute(double[] arr,double a, double b) throws Exception;
}
	@Test
	public void calculateExceptionTest() {
		double arrC[] = new double[] {9.99,2.45,4.55,7.76,5.33,8.62,10.001,9,2};
		IComputer2 comp = (double[] arr, double a,double b) -> {
			double arrResult[] = new double[arr.length];
			double sum = 0.0;
			for(int i=0;i<arr.length;i++) {
				arrResult[i] = arr[i]*a-b;
				sum=sum+arrResult[i];
			}
			try {
				double avg = sum/arrResult.length;
			}catch(ArithmeticException e) {
				throw new Exception("...");
			}
			return arrResult;
		};

		try {
			double arrResult2[] = comp.compute(arrC,1.4d, -0.3d);
			System.out.println(""+Arrays.toString(arrResult2));
			assertEquals(9,arrResult2.length);
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			fail();
		}
	}
```

```
Console: [14.286, 3.73, 6.669999999999999, 11.164, 7.762, 12.367999999999999, 14.3014, 12.9, 3.0999999999999996]
```

Referencje do metod statycznych

- W zmiennej interfejsu funkcyjnego można zapisać referencję do metody
- Pozwala to odwoływać się do metody bez jej wykonywania.
- Referencję metody statycznej tworzymy poprzez podanie nazwy klasy oraz nazwy metody statycznej rozdzielonych podwójnym znakiem „:”

  NazwaKlasy::nazwaMetody

- Referencja do metody może być używana wszędzie tam gdzie zachodzi zgodność typu docelowego.

Przykład:

```
public interface IIntTester {
     boolean test(int n);
}
public class ExampleStaticInt {
     static boolean isEven(int n) {return (n%2==0); }
     static boolean isNonNegative(int n) {return n>=0;}
}
public class MStaticReferenceDemo {
     static boolean intTest(IIntTester t,int n) { return t.test(n);}
}
     @Test
     public void methodStaticReferenceTest() {
          boolean result1 = MStaticReferenceDemo.intTest(ExampleStaticInt::isEven, 3);
          System.out.println("Liczba 3 ".concat(result1?"jest parzysta":"jest nieprzysta"));
          boolean result2 = MStaticReferenceDemo.intTest(ExampleStaticInt::isNonNegative, 4);
          System.out.println("Liczba 4 ".concat(result2?"jest nieujemna":"jest ujemna"));
          boolean result3 = MStaticReferenceDemo.intTest(ExampleStaticInt::isEven, 32);
          System.out.println("Liczba 32 ".concat(result3?"jest parzysta":"jest nieprzysta"));
          boolean result4 = MStaticReferenceDemo.intTest(ExampleStaticInt::isNonNegative, -42);
          System.out.println("Liczba -42 ".concat(result4?"jest nieujemna":"jest ujemna"));
     }
```

```
Konsola:

Liczba 3 jest nieprzysta
Liczba 4 jest nieujemna
Liczba 32 jest parzysta
Liczba -42 jest ujemna
```

Referencje do metod instancyjnych

- Referencję do metody instancyjnej danego obiektu można utworzyć w następujący sposób:

  referencjaObiektu::nazwaMetody

- Metoda do której odwołuje się stworzona referencja operuje w kontekście obiektu na który wskazuje referencja *„referencjaObiektu”.*
- Istnieje również możliwość pobrania referencji do metody lecz dla dowolnego obiektu danej klasy a nie konkretnego: *NazwaKlasy::nazwaMetodyInstancyjnej*
- W takim przypadku pierwszy parametr interfejsu odpowiada obiektowi, a pozostałe (o ile są) parametrom wywołania met.

Przykład:

```
public interface IIntTester2 {
	boolean test();
}
public class ExampleInstanceInt {
	private int n;
	ExampleInstanceInt(int n){this.n=n;}
	public int getN() { return n;}
	public boolean isPrimeNumber() {
		if(n<2)return false;

		for(int i=2;i<=n/i; i++) {
			if((n%i)==0) return false;
		}
		return true;
	}
}
@Test
public void methodInstanceReference2Test() {
	ExampleInstanceInt i1 = null;
	IIntTester2 intTester=null;
	boolean result1 = false;
	for(int i=2;i<20;i++) {
		i1 = new ExampleInstanceInt(i);
		intTester = i1::isPrimeNumber;
		result1 = intTester.test();
		if(result1)
			System.out.println("Liczba "
		.concat(Integer.toString(i1.getN())
		.concat(" jest liczbą pierwszą")));
		if(i==2) assertTrue(result1);
	}
}
```

```
public interface IIntTester3 {
	boolean test(ExampleInstanceInt obj);
}
public class ExampleInstanceInt {
	private int n;
	ExampleInstanceInt(int n){this.n=n;}
	public int getN() { return n;}
	public boolean isPrimeNumber() {
		if(n<2)return false;

		for(int i=2;i<=n/i; i++) {
			if((n%i)==0) return false;
		}
		return true;
	}
}
@Test
public void methodInstanceReference3Test() {
	ExampleInstanceInt i1 = null;
IIntTester3 intTester=ExampleInstanceInt::isPrimeNumber;
	boolean result1 = false;
	for(int i=2;i<20;i++) {
		i1 = new ExampleInstanceInt(i);
		result1 = intTester.test(i1);
		if(result1)
		System.out.println("Liczba"
		.concat(Integer.toString(i1.getN())
		.concat(" jest liczbą pierwszą")));
		if(i==2) assertTrue(result1);
		}
	}
```

Referencje do konstruktorów

- Java umożliwia również utworzenie referencji do konstruktorów.
- Ogólna postać wyrażenia:

NazwaKlasy::new

- Utworzoną w ten sposób referencję można przypisać do dowolnej zmiennej referencyjnej typu interfejsu funkcyjnego, który definiuje metodę abstrakcyjną ze zgodnym zwracanym typem.
- Wybór konstruktora do którego odwołuje się referencja następuje na podstawie przekazywanych parametrów, zgodnie z sygnaturą konstruktora.
- Metoda pozwala również na tworzenie tablic: *typ[]::new*

Przykład:

```
public interface IIntTester4 {
     ExampleInstanceInt create(int n);
}
public class ExampleInstanceInt {
     private int n;
     ExampleInstanceInt(int n){ this.n=n;}
     public int getN() { return n;}
     public boolean isPrimeNumber() {
          if(n<2) return false;
          for(int i=2;i<=n/i; i++) {  if((n%i)==0) return false;  }
          return true;
     }
}
@Test
     public void constructorReference4Test() {
          ExampleInstanceInt instInt = null;
          IIntTester4 intTester=null;
          boolean result1 = false;
          for(int i=2;i<20;i++) {
               intTester=ExampleInstanceInt::new;
               instInt = intTester.create(i);
               result1 = instInt.isPrimeNumber();
               if(result1) System.out.println("Liczba "
                              .concat(Integer.toString(instInt.getN())
                              .concat(" jest liczbą pierwszą")));
               if(i==2) assertTrue(result1);
          }
     }
```

Predefiniowane interfejsy funkcyjne

Interfejs	Opis
UnaryOperator<T>	Przyjmuje jeden argument typu T oraz zwraca wartość typu T. Udostępnia metodę apply().
BinaryOperator<T>	Przyjmuje dwa argumenty typu T oraz zwraca wartość typu T. Udostępnia metodę *apply*.
Consumer<T>	Przyjmuje jeden argument typu T, nie zwraca wartości. Udostępnia metodę *accept.*
Supplier<T>	Zwraca wartość typu T, nie przyjmuje argumentu. Udostępnia metodę *get*.
Function<T,R>	Przyjmuje jeden argument typu T, zwraca wartość typu R. Udostępnia metodę *apply*.
Predicate<T>	Przyjmuje jeden argument typu T, zwraca wartość typu *boolean*. Udostępnia metodę *test*.

Przykład:

```
    @Test
    public void predefinedInterfaces4Test() {
        UnaryOperator<Integer> quadratInt = null;
        quadratInt = (n)-> n*n;
        System.out.println("4^2="+quadratInt.apply(4));
        assertEquals(16,quadratInt.apply(4),0.0d);
        BinaryOperator<Double> productNumb = (n,m) -> n*m;
        System.out.println("3*4="+productNumb.apply(3d, 4d));
        assertEquals(12,productNumb.apply(3d, 4d),0.0d);
        Consumer<Float> printFloat = (n)->System.out.println("Przekazano "+n);
        printFloat.accept(43.4f);

        Supplier<String> yearAsString = () -> Integer.toString(LocalDate.now().getYear());
        System.out.println(yearAsString.get());
        assertEquals("2021",yearAsString.get());
        Function<Integer,String> starsComment = (n) -> {
            String comment="";
            for(int i=0;i<n;i++)
                comment+="*";
            return comment;
        };
        System.out.println(starsComment.apply(5)+"TEST"+starsComment.apply(5));

        Predicate<Integer> isEven = (n)->(n%2)==0;
        System.out.println("4 "+(isEven.test(4)?" jest ":" nie jest ")+" parzyste");
    }
```

Konsola:

```
4^2=16                  2021
3*4=12.0                *****TEST*****
Przekazano 43.4         4  jest  parzyste
```

Część 3 – Podstawy API strumieni

Interfejsy strumieni

- Strumień (w tym kontekście) jest określonym kanałem dla danych. Reprezentuje sekwencję obiektów.
- Strumień operuje na źródle danych, którym może być kolekcja lub tablica. Strumień nie przechowuje danych a jedynie wykonuje operacje na danych.
- Interfejsy strumieniu znajdują się w pakiecie *java.util.stream*
- Typ bazowy do interfejs *BaseStream,* który definiuje podstawowe metody dostępne we wszystkich strumieniach.
- Interfejs *BaseStream* rozszerza interfejs *AutoCloseable, a zatem strumienie mogą być zarządzanie przez instrukcję tr*y (dla zasobów).

Tworzenie strumieni

- Strumień może być utworzony z kolekcji - za pomocą metody *stream().*
- Strumień może być utworzony z tablicy - z użyciem klasy Arrays posiadającą metodę stream().
- Strumień może być utworzony z pliku – metoda lines() z klasy FileReader
- Strumienie dla typów prostych mogą tworzone być za pomocą odpowiednich klas *IntStream*, *LongStream* i *DoubleStream* oraz odpowiednich metod typu: *of(wartosci), generate()* lub dla strumieni całkowitoliczbowych *range(start, end)*.
- Strumień może powstać w wyniku wykonania wyrażenia regularnego oraz metody *splitAsStream*().
- Pusty strumień – metoda empty().

Przykład:

```
@Test
    public void streamFromCollectionTest() {
        Set<String> setTest = new LinkedHashSet<String>();
        setTest.add("Ala");
        setTest.add("ma");
        setTest.add("kota");

        Stream<String> newStream = null;
        newStream = setTest.stream();
        assertNotNull(newStream);
        assertTrue(newStream.anyMatch((n)-> n.equals("ma")));
    }

    @Test
    public void streamForPrimitivesTest() {
        IntStream ints = IntStream.range(0, 20);
        LongStream longs = LongStream.generate(() -> (new Random()).nextInt(100));
        DoubleStream doubles = DoubleStream.of(11, 21, 31);
        assertNotNull(ints);
        assertNotNull(longs);
        assertNotNull(doubles);

        int sum = ints.peek((n)->System.out.print(""+n+", ")).reduce(0, (a, b) -> a + b);
        assertEquals(190,sum);
        System.out.print("suma="+sum);
    }
```

```
Konsola:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, suma=190
```

Operacje na strumieniach zwracające nowy strumień

- Filtrowanie z użyciem metody *filter(predykat).* Metoda zwraca nowy strumień zwierający tylko te elementy, które spełniają predykat.
- Odwzorowywanie – zmiana typu elementu strumienia poprzez utworzenie nowego typu elementu na bazie przetwarzanych elementów. Typowa metoda to map(), ale dostępne są również metody dla typów prostych: *mapToInt*(), *mapToLong*(), *mapToDouble*() lub *mapToObj*() dla mapowania na obiekty.
- Limitowanie z użyciem metody *limit(rozmiar)* - zwraca strumień ograniczony do zadanej liczby elementów.
- Wykonywanie akcji na elemencie poprzez metodę *peek*() pozwalającą wykonać dodatkową operację z użyciem elementu (bez jego modyfikacji) oraz zwrócić strumień o tych samych elementach.

Operacje kończące

- Redukcja – zwrócenie wartości typu innego niż strumień, obliczonej na podstawie elementów strumienia. Do wykonania redukcji została przygotowania metoda *reduce(identityVal, BinaryOperator<T> accumulator).*
- Zliczanie – z użyciem metody count().
- Element największy – z użyciem metody max().
- Element najmniejszy – z użyciem metody min().
- Tworzenie kolekcji – z użyciem metody collect() oraz Klasy Collectors zawierającej m. in metody: toList(), toSet().
- Wykonanie akcji kończącej na każdym elemencie poprzez wywołanie metody *forEach()*
- Tworzenie tablicy – z użyciem metody *toArray().*
- *Inne: findFirst, findAny, noneMatch, allMatch, anyMatch(), itd.*

Przykład:

```
@Test
    public void streamFromFileTest() {
        String filePath = "./src/test/resources/plik2.txt";

        try (Stream<String> lines = new BufferedReader(new FileReader(filePath)).lines()) {
            String itemsWith20 = lines.peek((n)->System.out.println(n))
                .filter((n)->n.contains("1"))
                .filter((n)->n.contains("2"))
                .reduce("", (a,b)->((!a.equals(""))?";"+a:"").concat(b));
            System.out.println("Wyniki:"+itemsWith20);
        } catch (FileNotFoundException e) {
            System.out.println(e);
        }
    }
@Test
    public void streamMapTest() {
        int arr[] = new int[] {1,3,5,7,9};
        List<String> lst = null;
        lst = Arrays.stream(arr).mapToObj((n)-> "element".concat(Integer.toString(n)))
                .peek((n)->System.out.println(n)).collect(Collectors.toList());
        assertNotNull(lst);
        assertEquals(arr.length,lst.size());

    }
Konsola:
element1
element3
element5
element7
element9
```

Dalsze informacje o strumieniach

- Strumienie mogą być sekwencyjne lub równoległe. Metoda *stream*() zwraca strumień sekwencyjny, metoda *parallelStream* zwraca strumień równoległy. Strumień sekwencyjny można zamienić na równoległy wywołując metodę *parallel*(). Strumień równoległy można również zamienić na sekwencyjny przy użyciu metody *sequential*().
- Strumienie przetwarzają elementy dopiero po zdefiniowaniu operacji kończącej.
- Strumienie mogą zwracać *iterator*, za pomocą którego można iterować się w pętli *while* poprzez elementy strumienia.
- Strumienie posiadają metodę *distinct*() zwracają strumień z unikalnymi elementami.

Przykład:

```
@Test
	public void streamFromFileTest() {
		String filePath = "./src/test/resources/plik2.txt";//Plik ze 100 wierszami Ala ma kota (nr).

		try (Stream<String> lines = new BufferedReader(new FileReader(filePath)).lines()) {
		    String itemsWith20 = lines.peek((n)->System.out.println(n))
		            .filter((n)->n.contains("1"))
		            .filter((n)->n.contains("2"))
		            .reduce("", (a,b)->(a.equals("")?a:a.concat(";")).concat(b));
		    System.out.println("Wyniki:"+itemsWith20);
		} catch (FileNotFoundException e) {
		     System.out.println(e);
		}
	}

Konsola: Wyniki:Ala ma kota (12).;Ala ma kota (21).

@Test
	public void streamMapTest() {
		int arr[] = new int[] {1,3,5,7,9};
		List<String> lst = null;
		lst = Arrays.stream(arr).mapToObj((n)-> "element".concat(Integer.toString(n)))
		            .peek((n)->System.out.println(n)).collect(Collectors.toList());
		assertNotNull(lst);
		assertEquals(arr.length,lst.size());

	}

Konsola:
element1
element3
element5
element7
element9
```

Podsumowanie

- Obsługa wejścia i wyjścia
  - Strumienie bajtowe
  - Strumienie znakowe
  - Odczyt i zapis danych binarnych
- Wprowadzenie do wyrażeń lambda.
- Przykłady wyrażeń lambda.
- Referencje do metod statycznych.
- Referencje do metod instancyjnych.
- Referencje do konstruktorów.
- Podstawy przetwarzania strumieni.
- Przykłady przetwarzania strumieni.

.

Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Dziękuję za uwagę!