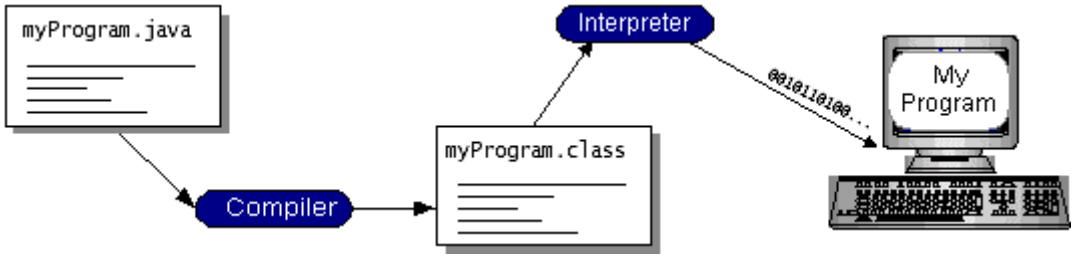
Programowanie w języku JAVA

Informacje ogólne



- The *Java Virtual Machine* (Java VM)
- The *Java Application Programming Interface* (Java API)
- JIT just in time

features:

- **The essentials**: Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
- **Applets**: The set of conventions used by applets.
- **Networking**: URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.
- **Internationalization**: Help for writing programs that can be localized for users worldwide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.
- **Security**: Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
- **Software components**: Known as JavaBeans™, can plug into existing component architectures.
- **Object serialization**: Allows lightweight persistence and communication via Remote Method Invocation (RMI).
- **Java Database Connectivity (JDBC™)**: Provides uniform access to a wide range of relational databases.

```
/**
 *  Przyklad programu konsolowego, wyswietlajacy napis Hello World!
 */
class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!"); //Display the string.
   }
}
```

Dokumentowanie kodu

Komentarze w kodzie źródłowym mogą być różnorakie:

- komentarz bloku pomiędzy ogranicznikami /** oraz */ - używany przez javadoc
- komentarz do końca linii, rozpoczynający się od znaków //
- komentarz bloku pomiędzy ogranicznikami /* oraz */

Tag	Wprowadzony w JDK/SDK

@author	1.0
{@code}	1.5
{@docRoot}	1.3
@deprecated	1.0
@exception	1.0
{@inheritDoc}	1.4
{@link}	1.2
{@linkplain}	1.4
{@literal}	1.5
@param	1.0
@return	1.0
@see	1.0
@serial	1.2
@serialData	1.2
@serialField	1.2
@since	1.1
@throws	1.2
{@value}	1.4
@version	1.0

Deklaracja zmiennych

typ nazwazmiennej deklaracja pojedynczej zmiennej
typ nazwamiennej, nazwazmiennej deklaracja wielu zmiennych

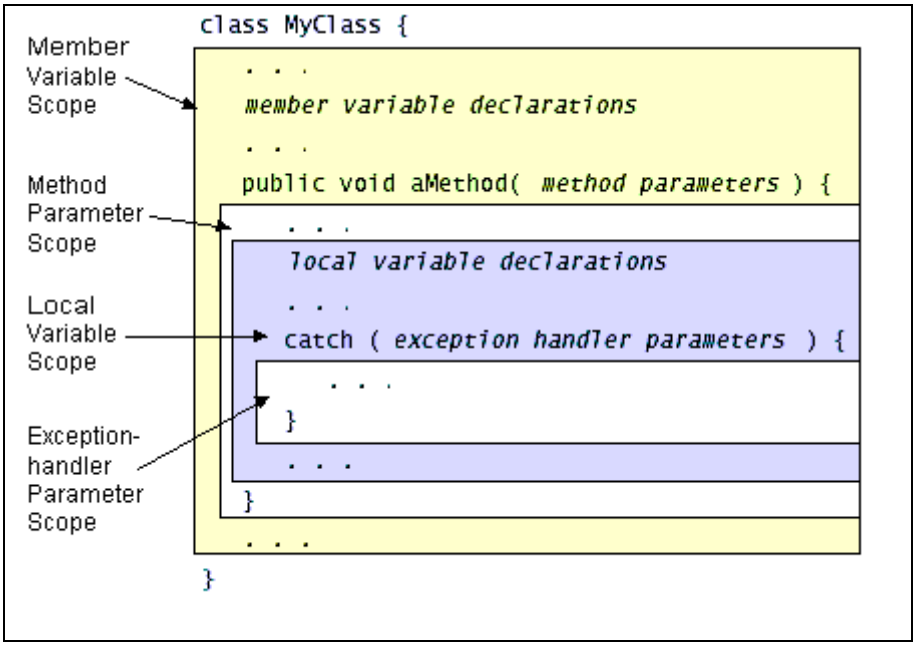
(*nazwazmiennej*

1) jest ciągiem znaków Unicode o dowolnej długości zaczynającym się literą,
2) nie pokrywa się ze słowami kluczowymi,
3) nie powtarza się w zasięgu obszaru, w którym została zdefiniowana,
4) przez konwencję, nazwy zmiennych zaczynają się małą literą, nazwy klas – dużą. W nazwach można łączyć słowa (*toJestZmienna*), podkreślenie używa się tylko w stałych, które składają się w całości z dużych liter.

Zasięg zmiennych

Jest to obszar programu, wewnątrz którego do zmiennej można odwoływać się podając jej nazwę własną. Obszar ten decyduje o momencie, w którym system zarezerwuje oraz zwolni fragment pamięci dla zmiennej. Zasięg zmiennej nie jest tym samym, co obszar jej widzialności. Widzialność zmiennych odnosi się wyłącznie do zmiennych wewnątrz klasy (parametrów) i określa on, czy zmienna może być użyta na zewnątrz klasy, w której została zadeklarowana.

- Widzialność określana jest przez modyfikatory dostępu (public, private, protected).
- Zasięg zmiennej określony być może w jeden z czterech sposobów:
    - parametr klasy (member variable), zmienna lokalna, parametr metody, obsługa wyjątku.



Zmienne finalne

Raz zainicjalizowane nie mogą być więcej modyfikowane
final int aFinalVar=0;
final int aBlankFinal; // zmienna zadeklarowana, ale jeszcze nie zainicjalizowana
aBlankFinal=0; // od tego momentu nie można modyfikować tej zmiennej;

Typy podstawowe:

[byte | short | int | long] całkowite;
[float | double] zmiennoprzecinkowe;
[char | boolean] inne

Typy od klas

nazwaklasy

Łańcuchy znaków

String
StringBuffer.append

Typ wyliczeniowy

deklaruje się używając słowa kluczowego enum

```
enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
            FRIDAY, SATURDAY };

for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
        System.out.println(d);
```

Taka deklaracja definiuje klasę (!). Najważniejsze cechy typów enum:

- Wartości wypisywane niosą informację
- Są testowane na zgodność typów w czasie kompilacji
- Istnieją we własnej przestrzeni nazw

- Zbiór stałych musi być stały przez cały czas
- Można dokonywać wyboru ze względu na stałą wyliczeniową (switch)
- Posiadają metodę o wartościach statycznych, która zwraca tablicę zawierającą wszystkie wartości typu enum w porządku ich zadeklarowania. Metoda ta zazwyczaj jest używana w kombinacji z konstrukcją for-each aby iterować poprzez wartości typu wyliczeniowego
- Można dostarczyć metody oraz pola, implementację interfejsu, itd.
- Dostarczają implementacji wszystkich metod klasy Object. Są Comparable oraz Serializable, oraz forma serial jest zaprojektowana tak, aby przeciwstawiać się zmianom w typie enum.

Uwagi:

- konstruktor jest prywatny, inny typ konstruktora będzie odrzucany przez kompilator;
- chociaż typ enum jest klasą, typy enum nie mogą dziedziczyć po sobie
- w java.util są specjalne implementacje klas korzystających z typu enum: EnumSet, EnumMap.

```
public enum Planet {
   MERCURY (3.303e+23, 2.4397e6),
   VENUS   (4.869e+24, 6.0518e6),
   EARTH   (5.976e+24, 6.37814e6),
   MARS    (6.421e+23, 3.3972e6),
   JUPITER (1.9e+27,   7.1492e7),
   SATURN  (5.688e+26, 6.0268e7),
   URANUS  (8.686e+25, 2.5559e7),
   NEPTUNE (1.024e+26, 2.4746e7),
   PLUTO   (1.27e+22,  1.137e6);

   private final double mass;   //in kilograms
   private final double radius; //in meters
   Planet(double mass, double radius) {
      this.mass = mass;
      this.radius = radius;
   }
   public double mass()   { return mass; }
   public double radius() { return radius; }

   //universal gravitational constant  (m3 kg-1 s-2)
   public static final double G = 6.67300E-11;

   public double surfaceGravity() {
      return G * mass / (radius * radius);
   }
   public double surfaceWeight(double otherMass) {
      return otherMass * surfaceGravity();
   }
}
```

kod programu	wynik działania

public static void main(String[] args) {
  double earthWeight = Double.parseDouble(args[0]);
  double mass =
earthWeight/Planet.EARTH.surfaceGravity();
  for (Planet p : Planet.values()) {
  System.out.printf("Your weight on %s is %f%n",
  p, p.surfaceWeight(mass));
  }
}
	$ java Planet 175
	Your weight on MERCURY is 66.107583
	Your weight on VENUS is 158.374842
	Your weight on EARTH is 175.000000
	Your weight on MARS is 66.279007
	Your weight on JUPITER is 442.847567
	Your weight on SATURN is 186.552719
	Your weight on URANUS is 158.397260
	Your weight on NEPTUNE is 199.207413

	Your weight on PLUTO is 11.703031

kiepskie rozwiązanie	rozwiązanie z metodą specyficzną (constant-specific)

public enum Operation {
  PLUS, MINUS, TIMES, DIVIDE;
 	
  // Do arithmetic op represented by this constant
  double eval(double x, double y){
  switch(this) {
  case PLUS: return x + y;
  case MINUS: return x - y;
  case TIMES: return x * y;
  case DIVIDE: return x / y;
  }
  throw new AssertionError("Unknown op: " + this);
  }
}
	public enum Operation {
	  PLUS {
	  double eval(double x, double y) { return x + y; }
	},
	  MINUS {
	  double eval(double x, double y) { return x - y; }
	},
	  TIMES {
	  double eval(double x, double y) { return x * y; }
	},
	  DIVIDE {
	  double eval(double x, double y) { return x / y; }
	};
	  // Do arithmetic op represented by this constant
	  abstract double eval(double x, double y);
	}

kod programu	wynik działania

public static void main(String args[]) {
  double x = Double.parseDouble(args[0]);
  double y = Double.parseDouble(args[1]);
  for (Operation op : Operation.values())
  System.out.printf("%f %s %f = %f%n", x, op, y, op.eval(x, y));
  }
	$ java Operation 4 2
	4.000000 PLUS 2.000000 = 6.000000
	4.000000 MINUS 2.000000 = 2.000000
	4.000000 TIMES 2.000000 = 8.000000
	4.000000 DIVIDE 2.000000 = 2.000000

Tablice

Tablice w Javie są obiektami (!). Tablice tworzy się albo przez ich statyczną inicjalizację, albo przez jawne wywołanie operatora new. Tablice wielowymiarowe są tablicami tablic (zagnieżdża się to aż to tablic jednowymiarowych).

```
int [] taba = {1,2,3,4}; // tablica czterech elementów typu int
int[] tabb = new int[10]; // choć można i tak: int tab[] = new int[10]
MyClass[] tabc = new tabc[5]; // tablica 5 referencji
for(int i=0; i<5; i++) tabc[i] = new MyClass(); // utworzenie referencji
```

inicjalizacja statyczna	operator new

public class ArrayOfArraysDemo {
  public static void main(String[] args) {
  String[][] cartoons =
  {
  { "Flintstones", "Fred", "Wilma",
  "Pebbles", "Dino" },
  { "Rubbles", "Barney", "Betty",
  "Bam Bam" },
  { "Jetsons", "George", "Jane",
  "Elroy", "Judy", "Rosie", "Astro" },
  { "Scooby Doo Gang", "Scooby Doo",
  "Shaggy", "Velma", "Fred", "Daphne" }
  };
	public class ArrayOfArraysDemo2 {
	  public static void main(String[] args) {
	  int[][] aMatrix = new int[4][];
	
	  //populate matrix
	  for (int i = 0; i < aMatrix.length; i++) {
	  aMatrix[i] = new int[5]; //create sub-array
	  for (int j = 0; j < aMatrix[i].length; j++) {
	  aMatrix[i][j] = i + j;
	  }
	  }

```
    for (int i = 0; i < cartoons.length; i++) {
        System.out.print(cartoons[i][0] + ": ");
      for (int j = 1; j < cartoons[i].length; j++) {
          System.out.print(cartoons[i][j] + " ");
        }
        System.out.println();
    }
  }
}
```

```
    //print matrix
    for (int i = 0; i < aMatrix.length; i++) {
        for (int j = 0; j < aMatrix[i].length; j++) {
            System.out.print(aMatrix[i][j] + " ");
        }
        System.out.println();
    }
  }
}
```

```
Flintstones: Fred Wilma Pebbles Dino
Rubbles: Barney Betty Bam Bam
Jetsons: George Jane Elroy Judy Rosie Astro
Scooby Doo Gang: Scooby Doo Shaggy Velma
Fred Daphne
```

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

Deklaracje z modyfikatorami (dotyczą zmiennych statycznych oraz instancyjnych)

[static] *deklaracjazmiennej* zmienna statyczna
[final] *deklaracjazmiennej* zmienna ustalona
[public | private | protected] *deklaracjazmiennej* zmienna z kontrolą dostępu

Przykłady deklaracji z inicjalizacją

```
byte largestByte = Byte.MAX_VALUE; // 127 , 8 bit
short largestShort = Short.MAX_VALUE; // 32767, 16 bit
int largestInteger = Integer.MAX_VALUE; // 2147483647,  32
long largestLong = Long.MAX_VALUE; // 9223372036854775807, 64

// real numbers
float largestFloat = Float.MAX_VALUE; // 3.40282e+38 , 32-bit IEEE 754
double largestDouble = Double.MAX_VALUE; // 1.79769e+308, 64-bit IEEE 754

// other primitive types
char aChar = 'S'; // 16-bit Unicode character
boolean aBoolean = true;  // true or false
```

Szablony (generic type)

Można go traktować trochę jak definicję szablonu w C++. W odniesieniu do klasy wygląda to tak:

```
public class Stack2<T> {
   private ArrayList<T> items;
   ...
   public void push(T item) {}
   public T pop() {}
   public boolean isEmpty() {}
}
```

Gdy w konstrukcji wyrażenia z typem rodzajowym wystąpi nazwa klasy, to nie obowiązuje prawo dziedziczenia (stąd tylko obiekty zdefiniowanej klasy mogą tam wystąpić).

```
public void printAll(Collection<Object> c) {
   for (Object o : c) {
      System.out.println(o);
   }
}
List<String> list = new ArrayList<String>();
```

```
public void printAll(Collection<Object> c) {
   for (Object o : c) {
      System.out.println(o);
   }
}
List<Object> list = new ArrayList<Object>();
```

... printall(list); //błąd	... printall(list); //dobrze

daje się to ograniczenie jednak obejść przez deklarację z dzikim typem.

Dziki typ (wildcard type):
Można stosować podobnie do typu rodzajowego, z tym, że teraz oczekiwany jest typ, który zgadza się z zadeklarowanym schematem.
Np. dla czegokolwiek w postaci Collection<?> da się wywołać metodę printAll:

```
public void printAll(Collection<?> c) {
    for (Object o : c) {
        System.out.println(o);
    }
}
```

Dziki typ można ograniczać od góry <? extends *Type*> do klas dziedziczącyc po *Type*, lub od dołu <? super *Type*> do klas ojcowskich dla *Type*. Zasada ta działa również przy deklaracji metod generycznych.

Deklaracja metod

Deklaracje metod generycznych

static <T> void fill(List<? super T> list, T obj)

Metody ze zmienną liczbą argumentów

wcześniej posługiwano się tablicą	w Javie 1.5 może to już być lista argumentów

Object[] arguments = {
  new Integer(7),
  new Date(),
  "a disturbance in the Force"
};
 	
String result = MessageFormat.format(
  "At {1,time} on {1,date}, there was {2} on planet
"
  + "{0,number,integer}.", arguments);
	String result = MessageFormat.format(
	  "At {1,time} on {1,date}, there was {2} on planet
	"
	  + "{0,number,integer}.",
	  7, new Date(), "a disturbance in the Force");

Deklaracja metody ze zmienną liczbą argumentów polega na umieszczeniu wyrażenia:
Object ...nazwa jako ostatniego argumentu metody.

```
public static String format(String pattern, Object... arguments);
```

Wtedy arguments może być tablicą lub też listą argumentów.

printf i format

W Javie 1.5 wprowadzona została funkcja printf umożliwiająca wypisywanie na konsolę sformatowanego ciągu znaków. Funkcja ta korzysta z możliwości, jakie daje zmienna liczba argumentów.

typy numeryczne: %[argument_index$][flags][width][.precision]conversion
typy daty i czasu: %[argument_index$][flags][width]conversion
format, w którym znaki formatujące nie odpowiadają argumentom: %[flags][width]conversion

Przykład formatowania: String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
Synopsis funkcji printf: public PrintStream printf(String format, Object... args)
Przykład wywołania: System.out.printf("passed=%d; failed=%d%n", passed, failed);

Operatory

unary, binary, ternary – operatory wymagające, odpowiednio, jednego, dwóch i trzech operandów (argumentów)

```
Unary:  operator op          //prefix notation
        op operator          //postfix notation
Binary: op1 operator op2     //infix notation
Ternary: op1 ? op2 : op3     //infix notation
```

Operatory arytmetyczne

można je stosować do liczb całkowitych i zmiennoprzecinkowych.

operatory binarne: + (dodawanie), - (odejmowanie), / (dzielenie), % (dzielenie modulo).

Data Type of Result	**Data Type of Operands**
long	Neither operand is a float or a double (integer arithmetic); at least one operand is a long.
int	Neither operand is a float or a double (integer arithmetic); neither operand is a long.
double	At least one operand is a double.
float	At least one operand is a float; neither operand is a double.

operatory unarne: +, - (negacja)

Operator	Use	Description
+	+op	Promotes op to int if it's a byte, short, or char
-	-op	Arithmetically negates op

operatory unarne: ++, -- (post i prefix'owe)

Operator	**Use**	**Description**
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Przyklad:

```
for (int i = 20; --i >= 0; ) {
    ...
}

int i = 10;
int n = i++%5;
```

Operatory relacji

Operator	**Use**	**Returns true if**
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Operatory warunkowe (stosowane do wyrażeń logicznych)

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
||	op1 || op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false (unary opeartor)
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
|	op1 | op2	either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

Operatory warunkowe **|** i **^** jeśli zastosowane są do liczb, a nie do wyrażeń logicznych, dokonują operacji na bitach.

Operatory przesunieć i operacji bitowych

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2 (unsigned)

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op2	bitwise complement

Operatory przypisania

Operator	Use	Equivalent to
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2 (wstawiane jest 0)

Inne operatory

Operator	Use	Description
?:	op1 ? op2 : op3	If op1 is true, returns op2. Otherwise, returns op3.
[]	*type* []	Declares an array of unknown length, which contains *type* elements.
[]	*type*[op1]	Creates and array with op1 elements. Must be used with the new operator.
[]	op1[op2]	Accesses the element at op2 index within the array op1. Indices begin at 0 and extend through the length of the array minus one.
.	op1.op2	Is a reference to the op2 member of op1.
()	op1(*params*)	Declares or calls the method named op1 with the specified

		parameters. The list of parameters can be an empty list. The list is comma-separated.
(type)	(type) op1	Casts (converts) op1 to type. An exception will be thrown if the type of op1 is incompatible with type.
new	new op1	Creates a new object or array. op1 is either a call to a constructor, or an array specification.
instanceof	op1 instanceof op2	Returns true if op1 is an instance of op2

Priorytety operatorów:

postfix operators	[] . (*params*) *expr*++ *expr*--
unary operators	++*expr* --*expr* +*expr* -*expr* ~ !
creation or cast	new (*type*)*expr*
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Uwagi:

Operatory o równym priorytecie są wykonywane zgodnie z powyższymi regułami. Wszystkie binarne operatory obliczane są z lewej do prawej, wszystkie operatory przypisania : od prawej do lewej. Jedynym przeciążanym operatorem jest operator +. Pozwala on na łączenie ciągu znaków.

```
System.out.println("The largest byte value is " + largestByte);
```

Pętle i instrukcje warunkowe

Pętle

while (*expression*) { *statement* }
do { *statement(s)* } while (*expression*);
for (*initialization*; *termination*; *increment*) { *statement* }

for(;;) – przykład pętli nieskończonej.
W Javie 1.5 wprowadzono pętle for służącą do przeglądania elementów kolekcji (omawiana przy okazji prezentacji kolekcji).

Warunki

if (*expression*) { *statement(s)* }
if (*expression*) { *statement(s)* } else { *statement(s)* }
switch (test) { case wartość: instrukcja; [break;] default: instrukcja;}
try { *statement(s)* } catch (*exceptiontype name*) { *statement(s)* } finally { *statement(s)*}

Skoki w strumieniu sterowania

return, return value, break [etykieta], continue [etykieta], etykieta:
Przykład:

```
search:
      for ( ; i < arrayOfInts.length; i++) {
         for (j = 0; j < arrayOfInts[i].length; j++) {
            if (arrayOfInts[i][j] == searchfor) {
               foundIt = true;
               break search;
            }
         }
      }
```

Klasy

Deklaracja

public	klasa publicznie dostępna
abstract	klasa abstrakcyjna
final	klasa nie może mieć potomków
class *NameOfClass*	nazwa klasy
extends *Super*	klasa, po której odbywa się dziedziczenie
implements *Interfaces*	interfejsy implementowane przez klasę

{
ClassBody
}
	ciało klasy

Metody i parametry klasy

Jeśli klasa ma parametr x, który pod tą samą nazwą występuje w klasie nadrzędnej, to parametr z klasy nadrzędnej zostanie przysłonięty. Wtedy x lub this.x będzie odwołaniem do parametru klasy, zaś super.x będzie odwołaniem do parametru klasy nadrzędnej.
Aby użyć przysłoniętego parametru x z klasy MyClass, która to klasa stoi w drzewie dziedziczenia powyżej klasy nadrzędnej do klasy bieżącej, należy posłużyć się konstrukcją: ((MyClass) this).x

Przykład: A jest klasą bazową, B dziedziczy z A, C dziedziczy z B oraz wszystkie klasy mają zadeklarowany parametr x. Wtedy w klasie C:
x oraz this.x są odwołaniami do x z klasy C, super.x and
((B) this).x jest odwołaniem do x z klasy B, ((A) this).x jest odwołaniemdo x z klasy A, zaś super.super.x jest odwołaniem błędnym.

program	wynik

class A {
int k=2;
int f() { return k ; }
}
class B extends A {
int k = 3;
int f() { return -k; } // przesłonięcie f z A.
}
public class Test {
public static void main(String arg[]) {
B b = new B();
System.out.println(b.k+" wywołano b.k”);
System.out.println(b.f()+" wywołano b.f()”);
A a = (A) b;
System.out.println(a.k+" wywołano a.k”);
System.out.println(a.f()+" wywołano a.f()”);
	3 wywołano b.k
	-3 wywołano b.f()
	2 wywołano a.k
	-3 wywołano a.f()

```
    }
}
```

Kontrola dostępu do parametrów i metod:

Specifier	class	subclass	package	world
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

private:

```
class Alpha {
   private int iamprivate;
   private void privateMethod() {
      System.out.println("privateMethod");
   }
}
class Beta {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprivate = 10;      // illegal
      a.privateMethod();      // illegal
   }
}
class Alpha {
   private int iamprivate;
   boolean isEqualTo(Alpha anotherAlpha) {
      if (this.iamprivate == anotherAlpha.iamprivate) //legal
         return true;
      else
         return false;
   }
}
```

protected:

```
package Greek;

public class Alpha {
   protected int iamprotected;
   protected void protectedMethod() {
      System.out.println("protectedMethod");
   }
}
***************
package Greek;

class Gamma {
   void accessMethod() {
      Alpha a = new Alpha();
      a.iamprotected = 10;    // legal
      a.protectedMethod();    // legal
   }
}
****************
package Latin;
```

```
import Greek.*;

class Delta extends Alpha {
   void accessMethod(Alpha a, Delta d) {
      a.iamprotected = 10;    // illegal
      d.iamprotected = 10;    // legal
      a.protectedMethod();    // illegal
      d.protectedMethod();    // legal
   }
}
```

Modyfikatory metod

abstract
static
native – używany, gdy definiowana jest metoda rodzima; w deklaracji klasy zawierającej metodę rodzimą powinien się również pojawić statyczny inicjalizator:

```
static { System.loadLibrary("simple");}
```

final – nie można takich metod przysłaniać, głównie stosowane do optymalizacji kodu
synchronized – zabezpieczenie przy wielowątkowości (monitor)

Konstruktory

nazwaklasy() blok, nazwaklasy(...) blok
przy czym niejawnie (jeśli nie zadeklarowano inaczej) jako pierwszy w bloku wywoływany jest konstruktor klasy bazowej super();

private - żadna inna klasa nie może stworzyć obiektu danej klasy. Jednak dana klasa może zawierać metody publiczne (factory methods), które mogą skonturować obiekt, zwracając go komendą return.
protected - tylko podklasy danej klasy oraz klasy w tym samym pakiecie mogą stworzyć obiekt danej klasy.
public - każda klasa może stworzyć obiekt danej klasy.
no specifier (dostęp package) - Tylko klasy z tego samego pakietu co dana klasa mogą stworzyć jej instancję.

Metoda finalizująca

Domyślnie każdy obiekt ma metodę finalize(), która nie robi niczego.

```
protected void finalize() throws Throwable { super.finalize(); }
```

Należy ją przesłonić, aby np. zaimplementować zwalnianie zasobów. finalize() można wywołać jawnie (jednak nie powoduje to usunięcia obiektu) lub też metodę to można pozostawić do wywołania przez sam system (stanie się to podczas odśmiecania, gdy znikną wszystkie referencje do obiektu).

Odwołania w konstruktorach:

this, nazwaklasy.this, super, super.nazwametody(), this(...), super(...), typ.class

Klasy wewnętrzne

Mają nieograniczony dostęp do metod i parametrów klasy w której zostały zadeklarowane (nawet do pól private). Tak jak inne klasy, klasy wewnętrzne można deklarować jako abstract lub final. Również można używać modyfikatorów dostępu: private, public, protected, package.

```
class EnclosingClass{
   . . .
   static class AStaticNestedClass {
      . . .
   }
   class InnerClass {
```

```
    . . .
  }
}
```

klasa wewnętrzna	wewnętrzna klasa anonimowa

public class Stack {
  private Vector items;
 	
//code for Stack's methods and constructors not shown...
 	
  public Enumeration enumerator() {
  return new StackEnum();
  }
 	
  class StackEnum implements Enumeration {
  int currentItem = items.size() - 1;
  public boolean hasMoreElements() {
  return (currentItem >= 0);
  }
  public Object nextElement() {
  if (!hasMoreElements())
  throw new NoSuchElementException();
  else
  return items.elementAt(currentItem--);
  }
  }
}
	public class Stack {
	  private Vector items;
	
	//code for Stack's methods and constructors not shown...
	
	  public Enumeration enumerator() {
	  return new Enumeration() {
	  int currentItem = items.size() - 1;
	  public boolean hasMoreElements() {
	  return (currentItem >= 0);
	  }
	  public Object nextElement() {
	  if (!hasMoreElements())
	  throw new NoSuchElementException();
	  else
	  return items.elementAt(currentItem--);
	  }
	  }
	  }
	}

Lokalna klasa wewnętrzna.

- Lokalna klasa wewnętrzna jest klasą zagnieżdżoną, posiadającą nazwę i nie będącą członkiem żadnej innej klasy.
- Każda deklaracja klasy lokalnej jest zawarta w bloku (a nie w klasie)
- Zasięg obowiązywania klasy lokalnej zadeklarowanej w bloku wyznacza koniec tego bloku.
- Klasa lokalna zdefiniowana w metodzie ma dostęp do finalnych zmiennych metody oraz do parametrów klasy zewnętrznej.

Przykład:
Klasa "Local" ma dostęp do parametrów klasy zewnętrznej. Ma ona również dostęp do zmiennych finalnych zadeklarowanych w bloku, w którym sama jest zadeklarowana.

```
package przyklady;        // Package declaration
 public class Global {
  private int someNumber;
  public void someMethod(final int number1,String name) {  //……
    class Local {
     public void accessTest() {
       System.out.println("Outer class variable someNumber =  " + someNumber);
       System.out.println("Method variable number1 =  " + number1);  // valid
       System.out.println("Method variable name =  " + name);           // invalid
     }
    }
   }
}
```

Inny przykład:

public class OuterClass {
  Vector v;
  public Enumeration returnEnumerator() {
  class Enum implements Enumeration {
  int count = 0;
  public Boolean hasMoreElements(){
  return (count < v.size()); }
  public Object nextElement() {
	Użycie klasy wewnętrznej
	  interface Enumeration {
	  boolean hasMoreElements ();
	  Object nextElement ();
	  }
	  /***************/
	  Enumeration e =
	  container.getEnumeration ();

```
            return v.elementAt(count++); }
    }
   return new Enum();
  }
}
```

```
        while (e.hasMoreElements ())
        {
        Object obj = e.nextElement ();
        System.out.println (obj);
        }
/*****************/
```

Klasa wewnętrzna niestatyczna

- jest klasą wewnętrzną, której deklaracja jest umieszczona bezpośrednio w deklaracji innej klasy bądź interfejsu.
- może być zadeklarowana jako: public, private, protected lub default/friendly.
- ma dostęp do parametrów klasy zewnętrznej.
- aby powstał obiekty tej klasy, musi powstawać obiekt klasy zewnętrznej..

Klasy wewnętrzne związane są z instancją klasy i jako takie mogą się odwoływać bezpośrednio do zmiennych instancyjnych i do metod klasy zewnętrznej. Można mówić, że klasy te odzwierciedlają wewnętrzne stosunki w klasie. Klasa wewnętrzna jest klasą której instancja istnieje wewnątrz instancji klasy zewnętrznej i klasa ta ma bezpośredni dostęp do parametrów instancji klasy zewnętrznej. W klasie wewnętrznej nie można deklarować pól statycznych.

Przykład:
Klasa "Inner" ma ona dostęp do parametrów klasy zewnętrznej. Fragment kodu:

```
new przyklady.Outer().new Inner().accessTest();
```

wyprodukuje na ekranie:

```
"Outer class variable someNumber = 0"
```

```
package przyklady;       // Package declaration
  public class Outer {
    private int someNumber;
      public class Inner {
        public void accessTest() {
          System.out.println("Outer class variable someNumber =  " + someNumber);
        }
    }
}
```

Klasy zagnieżdżone statyczne.

- Są to klasy wewnętrzne statyczne. Mogą mieć dostęp tylko do statycznych parametrów klasy zewnętrznej.
- Obiekty tych klas można tworzyć (lub też wywoływać metody samej klasy) bez tworzenia klasy zewnętrznej. Klasa ta jest osiągalna jak każdy inny parametr statyczny klasy zewnętrznej.

Klasy statyczne związane są z klasą, nie mają więc bezpośredniego dostępu do zmiennych instancyjnych i metod zdefiniowanych w klasie zewnętrznej. Do zmiennych tych mogą się odwoływać tylko przez referencje do obiektu. Można mówić, że kod jednej klasy występuje wewnątrz drugiej klasy.
Klasy statyczne nie przechowują (w przeciwieństwie do klas niestatycznych) referencji do klasy w której zostały zagnieżdżone. Ich istnienie (trwanie) nie pozostaje w żadnej zależności z istnieniem tamtej klasy (klasy, w której zostały zagnieżdżone).

Przykład: (po dodaniu modyfikatora static do klasy Inner)
Wywołanie metody accessTest() publicznej zagnieżdżonej klasy statycznej "Inner":

```
new przyklady.Outer.Inner().accessTest();
```

Klasy wewnętrzne anonimowe.

- nie posiadają nazwy
- mogą dziedziczyć po klasie lub mogą implementować interfejs, ale nie mogą robić tych dwóch rzeczy na raz.
- definicja, konstrukcja i użycie klasy odbywa się w jednym miejscu. Programista nie może zdefiniować specyficznego konstruktora dla takiej klasy, jednak może on przekazać konstruktorowi argumenty (przez wywołanie wprost konstruktora klasy basowej „super”.)

- Klasy anonimowe nigdy nie są abstrakcyjne. Klasy anonimowe zawsze są klasami wewnętrznymi, ale nigdy statycznymi. Klasy anonimowe zawsze rozumiane są jako final.
- Klasa anonimowa ma dostęp do parametrów finalnych metod oraz do parametrów klasy zewnętrznej.

```
interface Switch {
   void Off ();
   void On ();
}
class CeilingFan {
  private boolean running;
  public Switch getSwitch () {
    return new Switch () {
       public void Off () { running = false; }
       public void On () { running = true; }
    }
  }
}
```

```
class SimpleContainer
{
Object[] array = new Object [100];
Enumeration getEnumeration () {
   return new Enumeration ()  {
     int i = 0;
     public boolean hasMoreElements () {
          return i < 100; }
    public Object nextElement () {
          return array [i++]; }
   }
 }
}
```

Inicjalizacja parametrów klasy

Można inicjalizować przez użycie statycznych inicjalizatorów lub inicjalizatorów instancji dla członków klasy (parametrów), jeśli tylko są one zadeklarowane w klasie:

```
class BedAndBreakfast {
   static final int MAX_CAPACITY = 10;
   boolean full = false;
}
```

Ograniczenia:

1) inicjalizatory dają się zapisać instrukcją przypisania;
2) nie mogą wywoływać metod zgłaszających zwykłe wyjątki
3) jeśli zgłoszony zostanie wyjątek run-time, wtedy nie ma szans na zareagowanie na błąd.

Można też korzystać z statycznych bloków inicjalizacji (członkowie klasy inicjalizowani w bloku inicjalizacji statycznej, parametry obiektu (instance members) w konstruktorze).
Bloków statycznych może być dowolna ilość.

statyczny blok

```
import java.util.ResourceBundle;
class Errors {
static    ResourceBundle errorStrings;
static {
       try {
          errorStrings = ResourceBundle.
                        getBundle("ErrorStrings");
        }
         catch (java.util.MissingResourceException e) {
          // error recovery code here
       }
   }
}
```

inicjalizacja w konstruktorze

```
import java.util.ResourceBundle;
class Errors {
   ResourceBundle errorStrings;
   Errors() {
      try {
         errorStrings = ResourceBundle.
                        getBundle("ErrorStrings");
      }
         catch (java.util.MissingResourceException e) {
          // error recovery code here
      }
   }
}
```

Interfejsy

Interfejs jest nazwanym zbiorem definicji metod (bez implementacji). Interfejs może również deklarować stałe. Interfejs nie może implementować żadnych metod (a klasy abstrakcyjne mogą). Klasa może dziedziczyć tylko po jednej klasie bazowej, natomiast implementować interfejsów może wiele.
Interfejs nie należy do hierarchii klas. Klasy w żaden sposób nie związane mogą implementować ten sam interfejs. Istnieją pewne specjalne interfejsy, które nie implementują żadnych metod: Cloneable (legalizuje użycie metody Object.clone()) oraz Serializable (jeśli trzeba specjalnego traktowania, należy

zaimplementować metody: private void writeObject(java.io.ObjectOutputStream out) throws IOException, private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;).
.

Deklaracja interfejsu

public	dostęp publiczny (bez tego słowa dostęp package)
interface *InterfaceName*	nazwa interfejsu
extends *SuperInterfaces*	lista interfejsów

{
body
}
	ciało interfejsu

ciało interfejsu może zawierać deklaracje metod oraz stałych:

```
nazwaMetody(); // metody są interpretowane jako public abstract,
                  // dlatego nie potrzeba jawnie dodawać modyfikatorów
final int stala=1; // parametry są interpretowane jako public, static, final
                  // nie potrzeba dodawać public i static,
                  // nie można zaś używać transient, volatile oraz private, protected
```

Przykład:

```
public interface StockWatcher {
   final String sunTicker = "SUNW";
   final String oracleTicker = "ORCL";
   final String ciscoTicker = "CSCO";
   void valueChanged(String tickerSymbol, double newValue);
}
```

Implementacja interfejsu

W definicji klasy używa się słowa kluczowego implements. Wszystkie metody interfejsu powinny być zaimplementowane, w przeciwnym przypadku klasa musi być zadeklarowana jako abstract.
Dostęp do metod z interfejsu uzyskuje się dzięki operatorowi „.”, np. klasa.metodaInterfejsu.
Przykład:

```
public class StockApplet extends Applet implements StockWatcher {
   ...
   public void valueChanged(String tickerSymbol, double newValue) {
      if (tickerSymbol.equals(sunTicker)) {
         ...
      } else if (tickerSymbol.equals(oracleTicker)) {
         ...
      } else if (tickerSymbol.equals(ciscoTicker)) {
         ...
      }
   }
```

Użycie interfejsu jako typu

Wszędzie tam, gdzie może wystąpić nazwa klasy (jakiś typ), może również wystąpić nazwa interfejsu. Jednak tylko instancje, które implementują dany interfejs mogą być przypisane zmiennym referencyjnym typu będącego interfejsem.

Uwaga: z zasady nie modyfikuje się interfejsów (ze względu na problemy związane z koniecznością zaimplementowania wszystkich zadeklarowanych metod w klasach ich (interfejsów) używających)

Pakiety

Pakiety są kolekcją klas i interfejsów zapewniającą kontrolę nad dostępem i zasięgiem nazw.
Do deklaracji pakiety używa się słowa kluczowego package – jest to pierwsze słowo występujące w kodzie źródlowym. Po package dodaje się nazwę pakiety (która jest też związana ze strukturą katalogów).
Jeśli mamy w źródle deklarację: package mojeklasy.mojetesty, to w kartotece CLASSPATH powinny być katalog mojeklasy a w nim katalog mojetesty.

Wymuszanie finalizacji i odśmiecania

System.runFinalization();
Metoda ta wywołuje metody finalize wszystkich obiektów czekających na uprzątnięcie.
System.gc();
Wywołanie odśmiecacza (programista może stwierdzić sam, kiedy jest najlepszy czas na robienie porządków z pamiecią –np. przed operacjami wymagającymi dużych zasobów pamięci).

Kolekcje

Tablice

int[] anArray; // deklaracja zmiennej tablicowej
anArray = new int[10]; // stworzenie tablicy liczb całkowitych
for (int i = 0; i < anArray.length; i++) anArray[i] = i; // dostep do elementu tablicy
boolean[] answers = { true, false, true, true, false }; // inicjalizacja tablicy
arrayname.length // długość tablicy
String[] anArray = { "String One", "String Two", "String Three" }; // deklaracja i inicjalizacja tablicy obiektów
String[5] anArray; // przy takiej deklaracji każdy z pięciu elementów tablicy musiałby być zainicjalizowany przez użycie operatora new, np. anArray[0]=new String(„tekst”);

int[][] aMatrix = new int[4][]; // tablica tablic i jej inicjalizacja przez wywołanie operatora new (poniżej)

```
//populate matrix
      for (int i = 0; i < aMatrix.length; i++) {
            aMatrix[i] = new int[5];   //create sub-array
            for (int j = 0; j < aMatrix[i].length; j++) {
               aMatrix[i][j] = i + j;
            }
      }
```

System:
public static void arraycopy(Object *source*, int *srcIndex*, Object *dest*, int *destIndex*, int *length*)

```
char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
char[] copyTo = new char[7];
System.arraycopy(copyFrom, 2, copyTo, 0, 7);
```

Tablice hashowe

extends Dictionary
implements Map, Cloneable, Serializable

Obiekty używane jako klucze muszą implementować metody: hashCode oraz equals.

```
    Hashtable numbers = new Hashtable();
    numbers.put("one", new Integer(1));
    numbers.put("two", new Integer(2));
    numbers.put("three", new Integer(3));
    Integer n = (Integer)numbers.get("two");
    if (n != null) {
       System.out.println("two = " + n);
```

```
}
```

Kolekcje z interfejsami - informacje podstawowe

Kolekcje Javy są klasami, które powstały, aby ułatwiać programistom organizowanie danych w złożone struktury. Pod tym względem są one podobne do klas STL języka C++. Początkowo kolekcje były zaprojektowane tak, aby można było w nich przechowywać referencje do dowolnych obiektów. W wersji Javy 1.5.0 dodano deklarację typu przechowywanych referencji.

Przykład:

Kolekcje pierwotne	Kolekcje Javy 1.5.0(tzw. Generic)

import java.util.*;
 	
public class BasicCollection {
  public static void main(String args[]) {
  ArrayList list = new ArrayList();
 	
  list.add(new String("One"));
  list.add(new String("Two"));
  list.add(new String("Three"));
 	
  Iterator itr = list.iterator();
  while(itr.hasNext()) {
  String str = (String)itr.next();
  System.out.println(str);
  }
  }
  }
	import java.util.*;
	
	public class GenericCollection {
	  public static void main(String args[]) {
	  ArrayList<String> list = new ArrayList<String>();
	
	  list.add(new String("One"));
	  list.add(new String("Two"));
	  list.add(new String("Three"));
	  //list.add(new StringBuffer()); // illegal
	
	  Iterator<String> itr = list.iterator();
	  while(itr.hasNext()) {
	  String str = itr.next();
	  System.out.println(str);
	  }
	  }
	  }

Do obsługi nowych kolekcji dodano do języka nową postać instrukcji for

```
for( [collection item type] [item access variable] :
     [collection to be iterated] )
```

oto przykład:

```
import java.util.*;

  public class EnhancedForCollection {
    public static void main(String args[]) {
      ArrayList<String> list = new ArrayList<String>();

      list.add( new String("One") );
      list.add( new String("Two") );
      list.add( new String("Three") );

      for( String str : list  ) {
        System.out.println( str );
      }
    }
  }
```

Nowy sposób obsługi kolekcji pozwala na bezpośrednie wstawianie i odzyskiwanie wartości dla zmiennych typów podstawowych.

Kolekcje pierwotne	Kolekcje Javy 1.5.0(tzw. Generic)
import java.util.*;	import java.util.*;

```
public class PrimitiveCollection {
    public static void main(String args[]) {
      ArrayList list = new ArrayList();

  // box up each integer as they are added
      list.add( new Integer(1) );
      list.add( new Integer(2) );
      list.add( new Integer(3) );

  // now iterate over the collection and unbox
      Iterator itr = list.iterator();
      while( itr.hasNext() ) {
        Integer iObj = (Integer)itr.next();
        int iPrimitive = iObj.intValue();
        System.out.println( iPrimitive );
     }
    }
   }
```

```
public class AutoBoxCollection {
    public static void main(String args[]) {
      ArrayList<Integer> list = new ArrayList<Integer>();

      // box up each integer as it's added
      list.add( 1 );
      list.add( 2 );
      list.add( 3 );

      // now iterate over the collection
      for( int iPrimitive: list  ) {
        System.out.println( iPrimitive );
     }
    }
  }
```

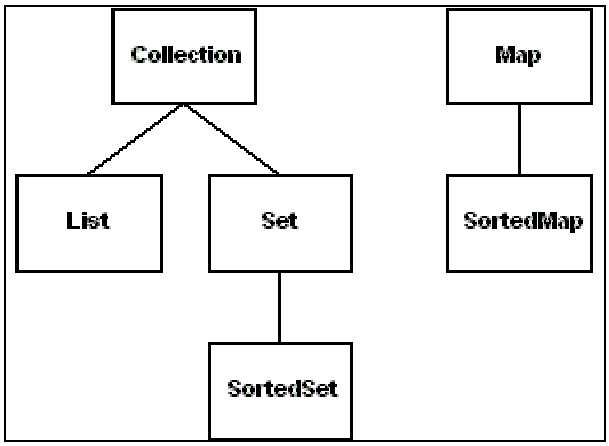
```
import java.util.*;

// Prints a frequency table of the words on the command line
public class Frequency {
   public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<String, Integer>();
      for (String word : args) {
         Integer freq = m.get(word);
         m.put(word, (freq == null ? 1 : freq + 1));
      }
      System.out.println(m);
   }
}

java Frequency if it is to be it is up to me to do the watusi
{be=1, do=1, if=1, is=2, it=2, me=1, the=1, to=3, up=1, watusi=1}
```

Hierarchia interfejsów

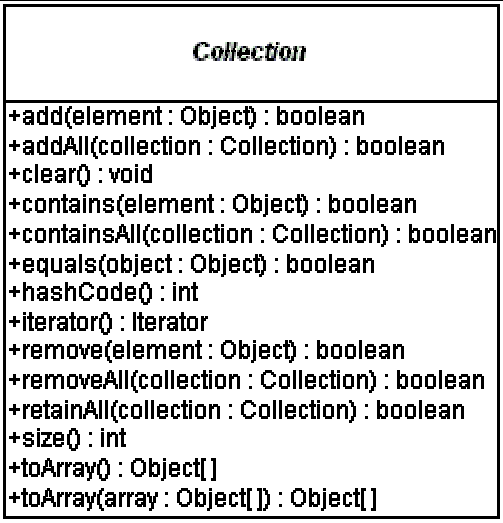
Analizę dostępnych metod w implementacjach kolekcji dobrze jest zacząć od schematu używanych w Javie 1.5.0 interfejsów:



W schemacie tym występują dwa rozłączne grupy interfejsów: interfejsy wywodzące się z Collection i interfejsy wywodzące się z Map. Zasadnicza różnica pomiędzy nimi polega na, odpowiednio, nieuwzględnianiu i uwzględnianiu dodatkowych atrybutów skojarzonych z przechowywanymi danymi (tzw. hashowanie).
W Javie dostarczono klasy, które implementują interfejsy z obu wymienionych grup.

Interfejs	Implementacja				Poprzednio
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		Hashtable Properties

Interfejs Collection
Część z jego metod implementuje klasa: AbstractCollection, z abstrakcyjnymi iterator() oraz size().
Podczas kodowania klasy implementującej ten interfejs (lub jakikolwiek inny z omawianego schematu) należy uwzględniać wyjątki typu: UnsupportedOperationException – dziedziczące z RuntimeException.

Model interfejsu Collection	Model interfejsu Iterator
	

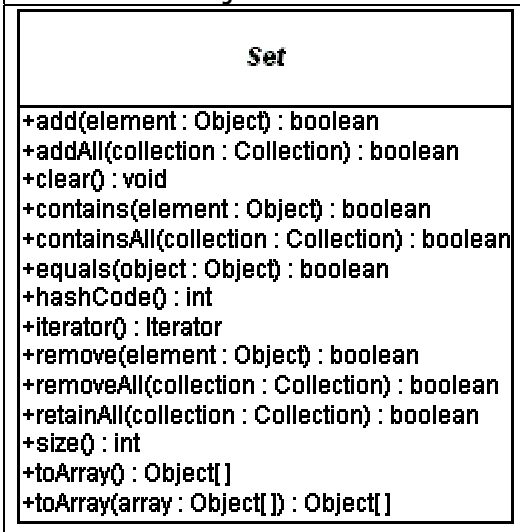
Przykład użycie iteratora:

```
Collection collection = ...;
Iterator iterator =
collection.iterator();
while (iterator.hasNext()) {
  Object element = iterator.next();
  if (removalCheck(element)) {
    iterator.remove();
  }
}
```

Interfejs Set

Klasy Javy implementujące jego metody to: HashSet oraz TreeSet. Do rozwijania własnych klas implementujących ten interfejs jest dostarczona klasa AbstractSet z nadpisanymi metodami equals() oraz hashCode().

Model interfejsu Set	Przykład użycia

	import java.util.*;
	
	public class SetExample {
	  public static void main(String args[]) {
	  Set set = new HashSet();
	  set.add("Bernadine"); set.add("Elizabeth");
	  set.add("Gene"); set.add("Elizabeth");
	  set.add("Clara"); System.out.println(set);
	  Set sortedSet = new TreeSet(set);
	  System.out.println(sortedSet);
	  }
	}
	wynik:
	[Bernadine, Gene, Clara, Elizabeth]
	[Bernadine, Clara, Elizabeth, Gene]

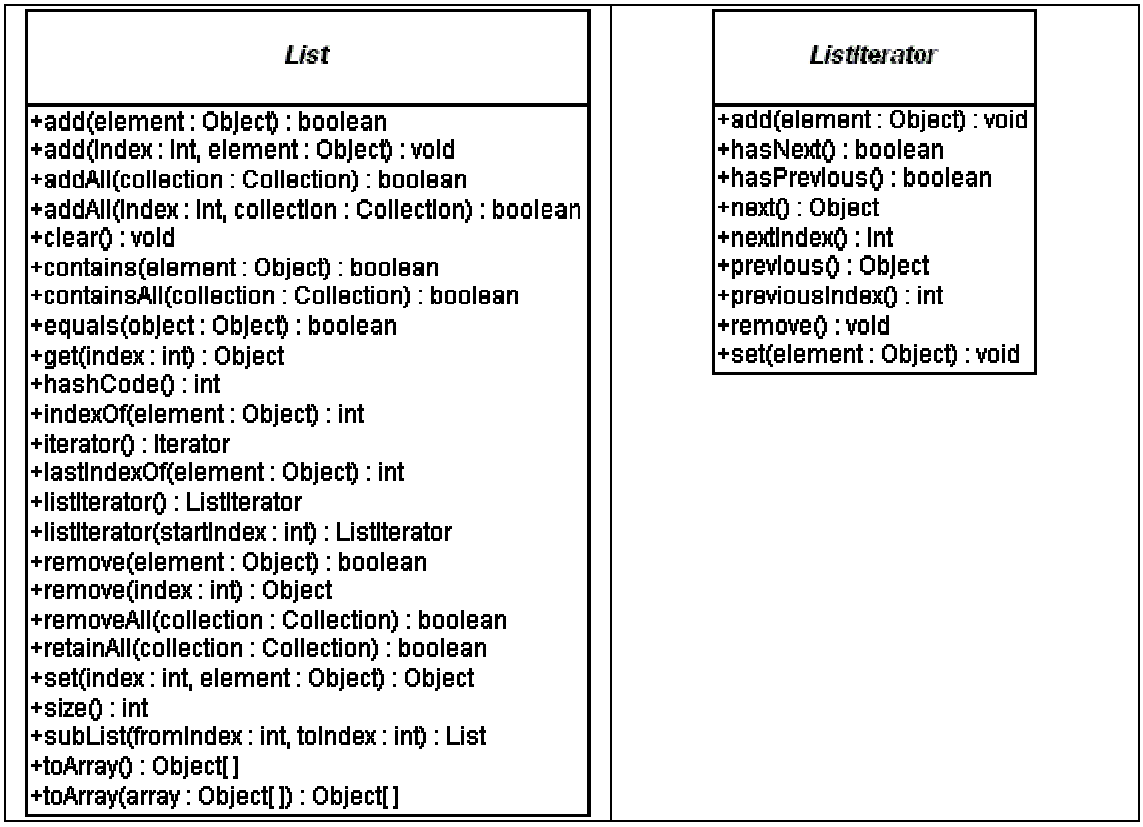
Interfejs List

Implementacje (implementujące interfejs Clonnable) dostarczone to:

ArrayList (stosowany, kiedy dodawanie i usuwanie elementów z końca)

LinkedList (stosowany, kiedy elementy wstawiane w dowolne miejsce, przeglądane zaś sekwencyjnie)

Model interfejsu List	Model interfejsu ListIterator



Przykład użycia iteratora:

```
List list = ...;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
  Object element = iterator.previous();
  // Process element
}
```

LinkedList

Historycznie wcześniej odpowiadały mu klasy Vector, Stack.

Rozszerzenie interfejsu LinkedList	Przykład użycia

	LinkedList queue = ...;
	queue.addFirst(element);
	Object object = queue.removeLast();
	LinkedList stack = ...;
	stack.addFirst(element);
	Object object = stack.removeFirst();

Przykład użycia	Wynik

import java.util.*;
 	
public class ListExample {
  public static void main(String args[]) {
  List list = new ArrayList();
  list.add("Bernadine");
  list.add("Elizabeth");
  list.add("Gene");
	[Bernadine, Elizabeth, Gene, Elizabeth, Clara]
	2: Gene
	0: Bernadine
	[Clara, Elizabeth, Gene, Elizabeth, Bernadine]
	[Clara, Elizabeth, Gene]

```
    list.add("Elizabeth");
    list.add("Clara");
    System.out.println(list);
    System.out.println("2: " + list.get(2));
    System.out.println("0: " + list.get(0));
    LinkedList queue = new LinkedList();
    queue.addFirst("Bernadine");
    queue.addFirst("Elizabeth");
    queue.addFirst("Gene");
    queue.addFirst("Elizabeth");
    queue.addFirst("Clara");
    System.out.println(queue);
    queue.removeLast();
    queue.removeLast();
    System.out.println(queue);
  }
}
```

Klasy abstrakcyjne implementujące interfejs List:
AbstractList
AbstractSequentialList

W obu nadpisane są metody equals() oraz hashCode()

	AbstractList	AbstractSequentialList
unmodifiable	Object get(int index) int size()	ListIterator listIterator(int index) - boolean hasNext() - Object next() - int nextIndex() - boolean hasPrevious() - Object previous() - int previousIndex() int size()
modifiable	unmodifiable + Object set(int index, Object element)	unmodifiable + ListIterator - set(Object element)
variable-size and modifiable	modifiable + add(int index, Object element) Object remove(int index)	modifiable + ListIterator - add(Object element) - remove()

Przykładowa implementacja

```
/* Poniższy przykład ilustruje użycie kolekcji Javy do implementacji
 * operacji na zbiorach obiektów (uznawanych za jednakowe, jeśli mają
 * taką samą wartość parametrów x). Dodawanie dwóch zbiorów powinno
 * w wyniku dawać zbiór, w którym elementy nie powtarzają się.
 *
 *
 * Moja - klasa z dwoma parametrami: x,y.
 *      Dwa obiekty tej klasy traktowane będą jako jednakowe,
 *      jeśli będą one miały jednakową wartość parametru x.
 *      Obiekty tej klasy dodawane będą do generycznej kolekcji TreeSet
 *      (zgodnie z konwecją JDK wersji 1.5).
 *      Do porównywania obiektów w kolekcji TreeSet<Moja> wykorzystany
 *      zostanie obiekt implementujący interfejs Comparator<Moja>. Metoda
 *      tego interfejsu compare() służy do organizacji kolekcji TreeSet
 *      (jak również kolekcji MapSet oraz implementacji algorytmu sortowania
 *      kolekcji). Interfejsu Comparator nie stosuje się do organizacji
 *      kolekcji HashSet.
 *
```

```
 * Moja2 - generalizacja klasy Moja, która implementuje interfejs Comparable
 *       Metoda compareTo() tego intererfejsu potrzebna jest do
 *       rozpoznawania obiektów o jednakowych wartościach x
 *       w kolekcji TreeSet (jeśli używanie kolekcji odbywa się zgodnie
 *       z konwencją JDK wersji >= 1.4).
 *                  compareTo() jest wywoływana przy każdym dodawaniu elementów do
 *       kolekcji. Bez niej pojawia się wyjątek ClassCastException.
 *
 * Moja3 - generalizacja klasy Moja, która przysłania metody hashCode()
 *       oraz equals() z klasy Object. Przysłanianie jest konieczne w
 *       przypadku, gdy do implementacji zbioru użyta zostanie klasa
 *       HashSet. W klasie tej używa się właśnie tych metod do porównywania
 *       obiektów. Bez przysłonięcia różne obiekty o jednakowych
 *       wartościach x byłyby traktowane jako obiekty różne.
 *
 * Op1<T>  klasa, która dostarczająca metodę sumującą dwa zbiory (TreeSet<T>)
 *       zawierające obiekty klasy T (wynikiem jest zbiór TreeSet<T>).
 *       Warunkiem poprawnego działania jest wcześniejsze
 *       dostarczenie do konstruktorów obiektów TreeSet
 *       obiektu implementującego interfejs Comparator<T>
 *
 * Op2    klasa, która dostarczająca metodę sumującą dwa zbiory (TreeSet)
 *       Warunkiem poprawnego działania jest umieszczanie w zbiorach
 *       obiektów implementującego interfejs Comparable
 *
 * Op3<T>  klasa, która dostarczająca metodę sumującą dwa zbiory (HashSet<T>)
 *       zawierające obiekty klasy T (wynikiem jest zbiór HashSet<T>).
 *       Warunkiem poprawnego działania jest umieszczenie w zbiorach
 *       obiektów przysłaniających metody equals() oraz hashCode()
 *
 * Op4    klasa, która dostarczająca metodę sumującą dwa zbiory (HashSet)
 *       Warunkiem poprawnego działania jest umieszczenie w zbiorach
 *       obiektów przysłaniających metody equals() oraz hashCode()
 *
 *
 */

import java.util.*;
class Moja {
          public int x, y;
          Moja(int xx, int yy){x=xx;y=yy;}
}

class Moja2 extends Moja implements Comparable{
          Moja2(int xx, int yy){super(xx,yy);}
          public int compareTo(Object o2){
                              if(x>((Moja2)o2).x) return -1;
                              if(x<((Moja2)o2).x) return  1;
                              else return 0;
                              }

}

class Moja3 extends Moja {
          Moja3(int xx, int yy){super(xx,yy);}

          public int hashCode(){
                    return x;
          }

          public boolean equals(Object obj){
                    if (x !=(((Moja)obj).x))
                     return false;
                    else return true;
          }
}
```

```
class Op1 <T>{
	public Set<T> suma(TreeSet<T> s1, TreeSet<T> s2){
		TreeSet<T> s0= new TreeSet<T>(s1);
		s0.addAll(s2);
		return s0;

	}

}

class Op2 {
	public Set suma(Set s1, Set s2){
		TreeSet s0 = new TreeSet(s1);
		s0.addAll(s2);
		return s0;
	}
}

class Op3 <T> {
	public Set<T> suma(HashSet<T> s1, HashSet<T> s2){
		HashSet<T> s0 = new HashSet<T>(s1);
		s0.addAll(s2);
		return s0;
	}
}
class Op4 {
	public Set suma(Set s1, Set s2){
		HashSet s0 = new HashSet(s1);
		s0.addAll(s2);
		return s0;
	}
}

class Comp implements Comparator<Moja>{
	public int compare(Moja o1, Moja o2){
			if(o1.x>o2.x) return -1;
			if(o1.x<o2.x) return  1;
			else return 0;
			}
}

class Glowna {

	/**
	 * Method main
	 *
	 *
	 * @param args
	 *
	 */
	public static void main(String[] args) {

	Moja m;

// Rozwiązanie z generycznym TreeSet
	TreeSet<Moja> s11 = new TreeSet<Moja> (new Comp());
	TreeSet<Moja> s12 = new TreeSet<Moja> (new Comp());
	m = new Moja(1,2);	s11.add(m);
	m = new Moja(2,3);	s11.add(m);
	m = new Moja(2,3);	s11.add(m);	s12.add(m);
	m = new Moja(4,3);				s12.add(m);
	m = new Moja(3,3);				s12.add(m);

	Op1<Moja> op1 = new Op1<Moja>();
	Set<Moja> s13 = op1.suma(s11,s12);
	for(Moja o: s11){
		System.out.print("s11  "+o.x);
```

```
                System.out.println(" " + o.y);
                }
                System.out.println("");
        for(Moja o: s12){
                System.out.print("s12 "+o.x);
                System.out.println(" " + o.y);
                }
                System.out.println("");
        for(Moja o: s13){
                System.out.print("s13 "+o.x);
                System.out.println(" " + o.y);
                }
                System.out.println("");

// rozwiązanie z tradycyjnym TreeSet,
// kompilator ostrzega, że taka implementacja jest kiepska,
// bo może dojść do niezgodności typów przy wywołaniach
// funkcji (unchecked call)

   TreeSet s21 = new TreeSet();
   TreeSet s22 = new TreeSet();
        m = new Moja2(1,2);s21.add(m);
        m = new Moja2(2,3);s21.add(m);
        m = new Moja2(2,3);s21.add(m);          s22.add(m);
        m = new Moja2(4,3);                        s22.add(m);
        m = new Moja2(3,3);                        s22.add(m);

        Op2 op2 = new Op2();
        Set s23 = op1.suma(s21,s22);
        Iterator i;
        i = s21.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s21  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");
        i = s22.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s22  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");
        i = s23.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s23  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");

//*******
// Rozwiązanie z generycznym HashSet
        HashSet<Moja3> s31 = new HashSet<Moja3> ();
        HashSet<Moja3> s32 = new HashSet<Moja3> ();
        Moja3 m3;
        m3 = new Moja3(1,2);   s31.add(m3);
        m3 = new Moja3(2,3);   s31.add(m3);
        m3 = new Moja3(2,3);   s31.add(m3);    s32.add(m3);
        m3 = new Moja3(4,3);                                  s32.add(m3);
        m3 = new Moja3(3,3);                                  s32.add(m3);

        Op3<Moja3> op3 = new Op3<Moja3>();
        Set<Moja3> s33 = op3.suma(s31,s32);
        for(Moja o: s31){
                System.out.print("s31  "+o.x);
```

```
                System.out.println(" " + o.y);
                }
                System.out.println("");
        for(Moja o: s32){
                System.out.print("s32 "+o.x);
                System.out.println(" " + o.y);
                }
                System.out.println("");
        for(Moja o: s33){
                System.out.print("s33 "+o.x);
                System.out.println(" " + o.y);
                }
                System.out.println("");

// rozwiązanie z tradycyjnym HashSet,
// kompilator ostrzega, że taka implementacja jest kiepska,
// bo może dojść do niezgodności typów przy wywołaniach
// funkcji (unchecked call)

   HashSet s41 = new HashSet();
   HashSet s42 = new HashSet();
        m = new Moja3(1,2);s41.add(m);
        m = new Moja3(2,3);s41.add(m);
        m = new Moja3(2,3);s41.add(m);          s42.add(m);
        m = new Moja3(4,3);                        s42.add(m);
        m = new Moja3(3,3);                        s42.add(m);

        Op4 op4 = new Op4();
        Set s43 = op4.suma(s41,s42);
        i = s41.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s41  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");
        i = s42.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s42  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");
        i = s43.iterator();
        while(i.hasNext()) {
                Moja o = (Moja) i.next();
                System.out.print("s43  "+o.x);
                System.out.println(" " + o.y);
        }
        System.out.println("");

        }


}
```

Wyjątki

Wyjątek jest zdarzeniem, które występuje podczas działania programu, przerywając normalne jego wykonywanie (normalny strumień instrukcji).

Tworzenie wyjątków w Javie i przekazywanie je do działającego programu (run time) nazywane jest zgłaszaniem wyjątków (*exceptions throwing*). W czasie zgłoszenia wyjątku system poszukuje obsługi wyjątku (*exceptions handlers*), tj. przegląda metody na stosie wywołań, począwszy od najbardziej zagnieżdżonej, aż znajdzie taką, która implementuje obsługę wyjątku. Mówi się, że metoda obsługująca wyjątek przechwytuje go. Jeśli wyjątek nie zostanie obsłużony, następuje zakończenie działania programu.

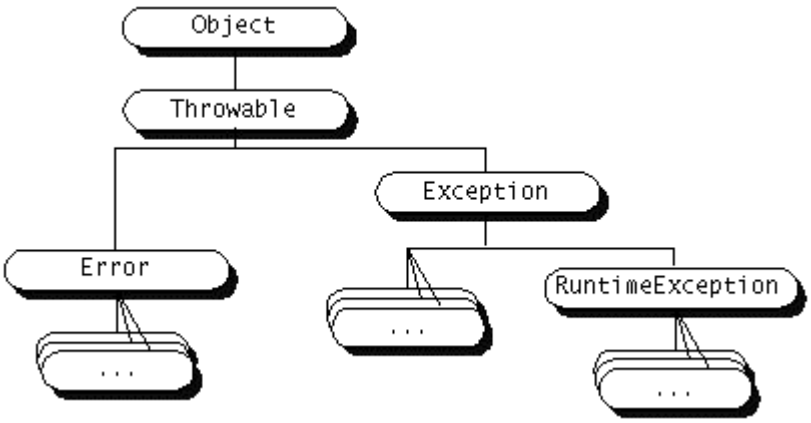
Deklaracja metody mogącej zgłaszać wyjątki:

```
public type metoda() throws Wyj1, Wyj2, Wyj3 {}
```

Jeśli w klasie potomnej jest przysłonięta metoda zgłaszająca wyjątki, to ta nowa metoda może być zadeklarowana z mniejszą liczbą wyjątków, lub też jako metoda nie zgłaszająca żadnych wyjątków. Metoda w klasie potomnej nie może natomiast zgłaszać więcej wyjątków, niż robi to metoda klasy nadrzędnej.

Deklaracja wyjątku:

Wyjątki są obiektami. Zorganizowane one są w strukturę klas:



Każdy wyjątek musi być obiektem z klasy będącej pochodną klasy Throwable.

Exception – klasa bazowa wszystkich jawnych wyjątków
IOException – klasa związana z błędami we/wy, pochodna Exception
RuntimeException - klasa związana z błędami runtime, pochodna Exception

Aby zadeklarować własny wyjątek wystarczy zdefiniować klasę:

```
class MyException extends Exception { ...}
```

Istnieją dwa standardowe konstruktory wyjątków: bez parametrów i z łańcuchem znaków.
Dlatego, jeśli łańcuch znaków ma zostać użyty jako argument konstruktora, należy we własnym konstruktorze wywołać konstruktor klasy nadrzędnej: super(str)
Dobrą praktyką jest nadawanie nazw takich nazw wyjątkom, które mówią coś o klasie bazowej. Np. MyException to dobra nazwa dla klasy dziedziczącej po klasie Exception, MyError – po klasie Error.

Obsługa wyjątków:

Jeśli w programie wywołana jest metoda zgłaszająca wyjątek, to musi być ona ujęta w blok try-catch.

```
try {
   metodaZglaszajacaWyjatek();
} catch ( . . . ) {
   . . .
```

```
} catch ( . . . ) {
    . . .
} . . .
```

Jeśli w jakiejś metodzie zadeklarowanej jako metoda zgłaszająca wyjątki użyta jest inna metoda zgłaszająca wyjątek, to nie trzeba tej użytej metody chronić blokiem try-catch. Nastąpi niejawne przekazanie wyjątku do metody zewnętrznej.

```
method1 {
    try {
        call method2;
    } catch (exception) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```