Język programowania Java

dr hab. inż. Marek Wojciechowski

Wprowadzenie

Czym jest Java?

- **Język programowania**
  - prosty
  - zorientowany obiektowo
  - rozproszony
  - kompilowany i interpretowany
  - wydajny
  - bezpieczny
  - wielowątkowy
  - przenaszalny
  - dynamiczny
- **Platforma**
  - Platforma software'owa
  - Rozumiana jako środowisko do uruchamiania programów
  - Oparta o maszynę wirtualną Java (JVM)
  - Nie tylko dla języka Java!

Język programowania Java

- Zorientowany obiektowo język programowania
  - prosta składnia
  - rozbudowane biblioteki
- Zaprojektowana przez firmę Sun (twórca: James Gosling)
  - Obecnie Java należy do Oracle
  - Specyfikacje techniczne tworzone w ramach Java Community Process (JCP)
- Język o składni wzorowanej na C++
  - Podobieństwo składni, inna filozofia
- Zaprojektowana "od zera", początkowo pod nazwą Oak
- Początkowo pomyślana do zastosowań w urządzeniach elektronicznych codziennego użytku
- Przyjęła się jako język do zastosowań sieciowych (aplikacje intra- i internetowe, szczególnie korporacyjne)

Edycje Javy (platformy Java)

- **Java Platform, Standard Edition (Java SE, dawniej J2SE)**
  - Trzon technologii Java
  - Wyznacza wersje języka Java (11 LTS – 09.2018, 13 – 09.2019)
  - Java Runtime Environment (JRE) = maszyna wirtualna + biblioteki
  - Java Development Kit (JDK) = JRE + narzędzia deweloperskie
- **Java Platform, Enterprise Edition (Java EE, dawniej J2EE)**
  - Zestaw specyfikacji (technologii składowych) dotyczących tworzenia aplikacji internetowych/rozproszonych klasy enterprise
  - Bazuje na Java SE
  - Aplikacje instalowane na serwerach aplikacji lub zawierające wbudowany serwer
  - Podarowana fundacji Eclipse, dalszy rozwój pod nazwą Jakarta EE
- **Java Platform, Micro Edition (Java ME, dawniej J2ME)**
  - Edycja dla urządzeń mobilnych i systemów wbudowanych
  - Okrojona wersja Javy SE + biblioteki dla urządzeń tej klasy
  - Brak związku z Androidem! (z którym przegrywa na smartfonach i TV)
- **Java Card**
  - Dla zastosowań w kartach chipowych (ang. smart cards)

Ekosystem Java

- Java Runtime Environment (JRE)
  - Maszyna wirtualna (JVM) + biblioteki
- Java Development Kit (JDK) = JRE + narzędzia deweloperskie:
  - `javac` - kompilator java
  - `java` - maszyna wirtualna Javy
  - `jar` – obsługa archiwów
  - `jdb` - debugger klas Java
  - `javadoc` - generator dokumentacji klas w formacie HTML
- Środowiska IDE: NetBeans, Eclipse, IntelliJ IDEA, JDeveloper
- Narzędzia do budowania projektów (command line + integracja z IDE): Ant, Maven, Gradle
- Serwery aplikacji:
  - GlassFish, WildFly / JBoss, WebLogic, WebSphere, Tomcat, TomEE …
- Frameworki: Spring, …, Struts, Vaadin, Tapestry, …

Charakterystyka języka Java (1)

- Java jest prosta (stosunkowo…)
  - składnia podobna do C++ (i C#)
  - nie ma w Javie:
    - przeciążania operatorów (poza jednym wyjątkiem)
    - plików nagłówkowych i preprocesora
    - operacji arytmetycznych na wskaźnikach
    - struktur i unii
    - dziedziczenia wielokrotnego (po klasach)
- Java jest zorientowana obiektowo
  - klasy i interfejsy
  - od Java 8 elementy programowania funkcyjnego (lambda)

Charakterystyka języka Java (2)

- Java jest kompilowana, ale niezależna od architektury
  - kompilacja do kodu pośredniego - bajtkodu (ang. byte code)
  - kod pośredni jest interpretowany przez Java Virtual Machine (JVM)
  - definicja języka i maszyna wirtualna są w pełni wyspecyfikowane
  - nie ma elementów niezdefiniowanych lub zależnych od implementacji (np. typy proste są takie same na wszystkich maszynach)
  - pełna przenaszalność kodu źródłowego i wynikowego: "Write Once, Run Anywhere" (WORA)
- Java jest wielowątkowa
  - wątki Javy w miarę możliwości przekładane są na wątki systemu operacyjnego
  - możliwość synchronizacji

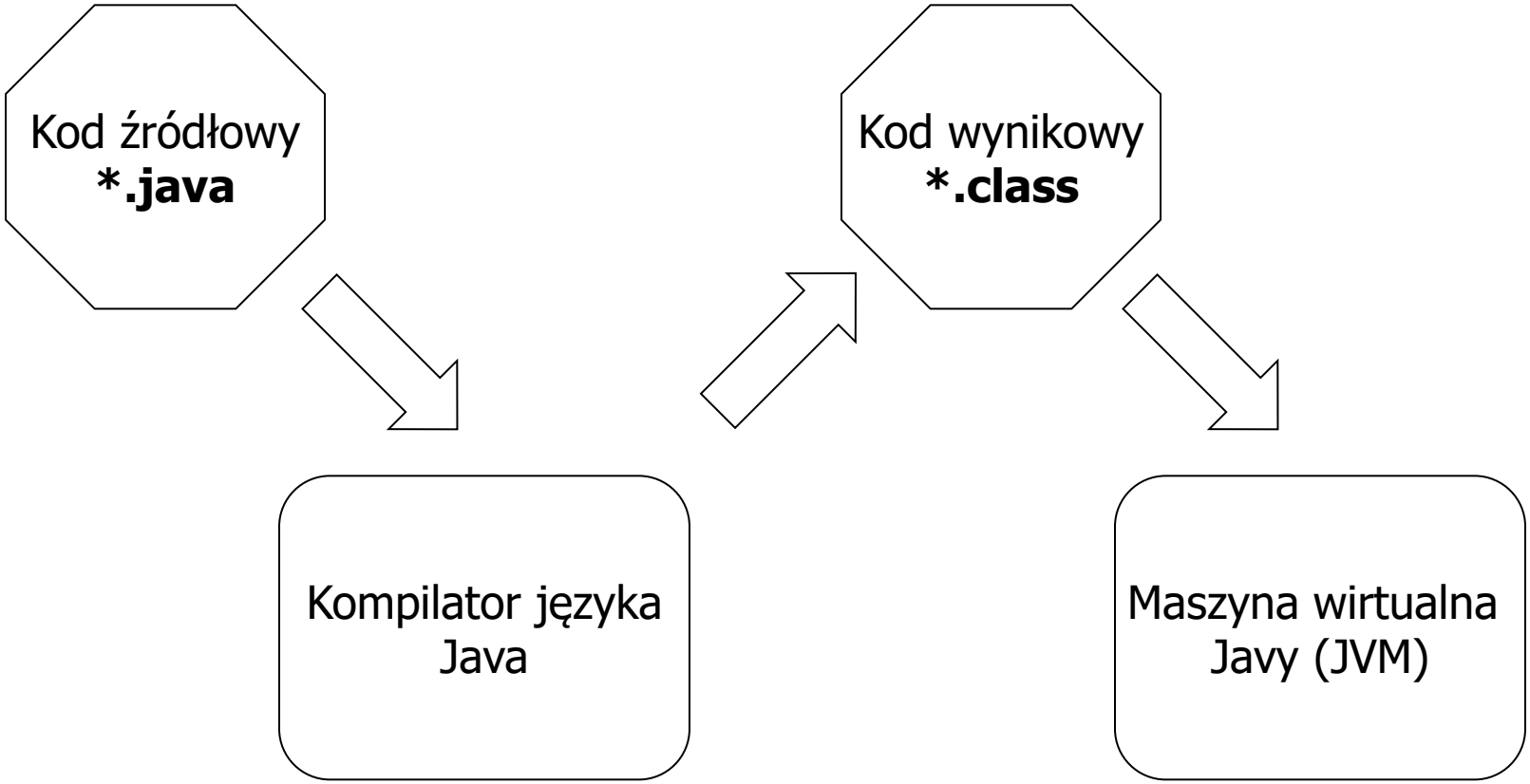
Charakterystyka języka Java (3)

- Java nadzoruje pamięć
  - nie ma wskaźników takich jak w C++
  - dostęp do obiektów przez **referencje**
    - referencje zachowują się jak "bezpieczne wskaźniki"
    - każdy dostęp do pamięci jest kontrolowany
  - odśmiecanie pamięci (ang. garbage collection)
- Java jest odporna na błędy i bezpieczna
  - ściśle określona forma kodu pośredniego
  - kontrola dostępów do pamięci
  - obsługa błędów w oparciu o wyjątki
  - ograniczenie dostępu do zasobów (Security Manager)

Charakterystyka języka Java (4)

- Java jest niewielka (stosunkowo…)
  - pomyślana dla małych systemów
- Java jest szybka (stosunkowo…)
  - szybka w porównaniu z innymi językami interpretowanymi
  - wolniejsza od C (może to nie mieć znaczenia w programach wykorzystujących komunikację sieciową lub często oczekujących na reakcję użytkownika)
  - JVM może wykorzystywać kompilatory typu Just-In-Time, aby poprawić efektywność przetwarzania
- Java jest rozszerzalna
  - istnieje możliwość wykorzystywania bibliotek napisanych w innych językach
- Java jest dynamiczna (stosunkowo…)
  - np. możliwość pobierania klas z Internetu w trakcie pracy programu

Uruchamianie programów w języku Java



- Java 9: jshell – interaktywne narzędzie do nauki języka i prototypowania kodu (REPL - Read-Evaluate-Print Loop)
- Java 11: uruchamianie pojedynczych plików źródłowych (kompilacja w pamięci)

Maszyna wirtualna Javy (JVM)

- Zachowuje się jak "wirtualny komputer" interpretujący bajtkod Javy
- Stanowi bezpieczne środowisko do uruchamiania programów
- Musi być zaimplementowana dla konkretnej platformy
- JVM może wykorzystywać kompilację Just-In-Time (JIT)
  - kompilacja "w locie" bajtkodu do instrukcji maszynowych
  - szczególnie efektywna w przypadku powtarzalnych fragmentów kodu (np. pętle)
- JVM może stanowić samodzielną aplikację lub być wbudowana w inny program np. przeglądarkę, serwer bazy danych

Inne języki dla JVM

- Początkowo Java była jedynym językiem kompilowanym do bajtkodu uruchamianego na JVM
- Z czasem pojawiły się kompilatory dla innych języków generujące bajtkod dla JVM
- Interpretery istniejących języków:
  - JRuby (Ruby)
  - Jython (Python)
  - Rhino (JavaScript)
- Nowe języki dla JVM:
  - Groovy (dynamiczny język skryptowy)
  - Scala (łączy programowanie obiektowe z funkcyjnym)
  - Closure (dialekt języka LISP)
  - Kotlin, Ceylon, …

Java vs. .NET

- Microsoft .NET Framework oparty o podobne koncepcje
  - Wirtualne środowisko uruchomieniowe (CLR vs. JVM)
  - Podstawowy język – C#, podobny koncepcyjnie i składniowo do Javy (choć występują w obu tych obszarach różnice), również nawiązujący do C++, ale o bardziej rozbudowanej składni niż Java
- Java od początku dostępna na wiele platform, .NET przez wiele lat w pełni dostępny tylko pod Windows
- .NET od początku zorientowany na wiele języków kompilowanych do tego samego formatu kodu pośredniego, Java przez wiele lat jako jedyny język platformy Java

Zastosowania języka Java

- Samodzielne aplikacje (desktopowe)
  - graficzne (Swing, JavaFX)
  - pracujące w trybie tekstowym
- Aplety
  - małe aplikacje zagnieżdżane w dokumentach HTML
  - w dużym stopniu przyczyniły się do popularności Javy
- Komponentowe aplikacji Java EE
  - Serwlety, strony JSP, różne rodzaje beans, EJB, Web Services, …
- Podprogramy składowane w bazach danych
  - Oracle, DB2
- Natywne aplikacje mobilne (Android)
  - dedykowana wersja JVM: Dalvik

Programowanie obiektowe (1)

- Abstrakcja
  - identyfikacja obiektów i operacji na nich
  - klasyfikacja podobnych obiektów za pomocą klas
  - zdefiniowanie atrybutów i dopuszczalnych operacji dla poszczególnych klas
- Hermetyczność (enkapsulacja)
  - ukrycie wewnętrznej implementacji klas
  - dostęp do obiektów poprzez wyspecyfikowany interfejs
  - interakcja z obiektami przez wysyłanie *komunikatów* (wywołania publicznych metod)
  - zalety: ochrona i elastyczność kodu

Programowanie obiektowe (2)

- Dziedziczenie
  - technika wykorzystania istniejących fragmentów kodu
  - polega na tworzeniu nowych klas na bazie już istniejących
  - cechy wspólne dla wszystkich podklas definiowane są w nadklasie
  - podklasa może
    - dziedziczyć cechy nadklasy
    - nadpisywać zachowanie nadklasy
    - dodawać nowe atrybuty i zachowania
- Polimorfizm
  - pozwala w jednolity sposób traktować obiekty klas z hierarchii dziedziczenia przy zachowaniu ich charakterystycznego zachowania
  - od strony technicznej sprowadza się do tzw. późnego wiązania metod przy ich wywołaniu (wybór metody na podstawie typu obiektu)
  - w Javie wszystkie metody (niestatyczne) zachowują się jak metody wirtualne w C++ / C#

Aplikacje w języku Java

- Kod programu w języku Java jest pogrupowany w klasy
- Klasy pogrupowane są w pakiety
- Aplikacje Java posiadają w jednej z klas wyróżnioną metodę o nazwie `main()`, od której rozpoczyna się wykonanie programu

Hello.java

```
package hello;

public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Przekazywanie parametrów wywołania aplikacjom Java

- Parametry przekazywane jako tablica obiektów klasy `String`
- Tablica w Javie jest pełnoprawnym obiektem

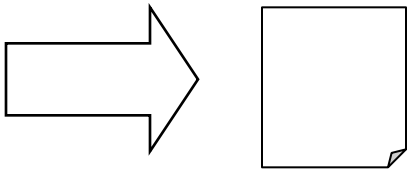
Hello.java

```
public class Hello {
   public static void main(String[] args) {
       if (args.length > 0)
          System.out.println("1st parameter: " + args[0]);
       else
          System.out.println("No parameters");   }
}
```

Kompilacja i uruchomienie programu

```
public class SayHello {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Plik SayHello.java



```
C:> javac SayHello.java

C:> java SayHello
Hello world
```

Kompilacja pliku .java do pliku .class

Uruchomienie programu

Wielkość znaków jest istotna

- Maszyna wirtualna Javy szuka wywoływanych klas w katalogach i archiwach wymienionych w zmiennej środowiskowej CLASSPATH, np.: CLASSPATH=.;c:\classes;c:\lib\mylib.jar
  - Kropka oznacza katalog bieżący, separatorem w Win jest „;” a w UNIX „:”

Typy danych, operatory, instrukcje sterujące

Typy proste

- Liczby, znaki i wartości logiczne w Javie nie są obiektami (w odróżnieniu od „w pełni obiektowych języków")
- Java oferuje 8 typów prostych:
  - **boolean** (*true* lub *false*)
  - **char** (16-bitowy Unicode)
  - **byte** (8-bitowy typ całkowity, ze znakiem, U2)
  - **short** (16-bitowy typ całkowity, ze znakiem, U2)
  - **int** (32-bitowy typ całkowity, ze znakiem, U2)
  - **long** (64-bitowy typ całkowity, ze znakiem, U2)
  - **float** (32-bitowy typ zmiennoprzecinkowy, IEEE 754)
  - **double** (64-bitowy typ zmiennoprzecinkowy, IEEE 754)
- W przypadku gdy istnieje konieczność traktowania wartości prostych jak obiektów należy (jawnie lub niejawnie od 1.5) skorzystać z tzw. klas opakowujących (ang. wrapper)
  - np. by przypisać im null, umieścić w kolekcjach

Komentarze

```
/*
  Multi-line comment
  (like in C and C++).
*/
```

```
// Single-line comment (like in C++)
```

```
/**
  Comment used by javadoc
  (HTML documentation generator)
*/
```

Zmienne

- Każda zmienna posiada typ
  - typ prosty (int, float, boolean, ...)
  - typ obiektowy
  - typ tablicowy
  - typ wyliczeniowy (od 1.5)
- Zmienne muszą być deklarowane przed użyciem
- Deklaracja może być połączona z nadaniem wartości
- Java jest językiem o ścisłej kontroli typów
- Od Java 10 typ zmiennej lokalnej nie musi być podany jawnie (var)
  - Kompilator wywiedzie typ zmiennej na podstawie przypisanej wartości (inicjalizacja konieczna w deklaracji!)

```
int age;

double pi = 3.14;

boolean truthy = true, falsy = false;

var txt = "Hello";  // of String type (from Java 10)
```

Nazwy zmiennych

- Zaczynają się od litery, znaku podkreślenia lub $
- Na kolejnych pozycjach mogą występować również cyfry
- Wielkość liter w języku Java ma znaczenie!
- Nazwy zmiennych nie mogą pokrywać się z zastrzeżonymi słowami kluczowymi języka Java:
  - `boolean, byte, char, double, float, int, long, short, void`
  - `false, null, true`
  - `abstract, final, native, private, protected, public, static, synchronized, transient, volatile`
  - `break, case, catch, continue, default, do, else, finally, for, if, return, switch, throw, try, while`
  - `class, extends, implements, interface, throws`
  - `import, package, instanceof, new, super, this`
  - `strictfp, assert, enum`

Literały

- Całkowitoliczbowe
  - 12, -12, 0123, 0x12f, 0X7A3, 0b101010 (from Java 7), 15L
- Zmiennoprzecinkowe
  - 8.31, 3.00e+8, 8.31F , 3.00e+8f
- Logiczne
  - true, false
- Znakowe
  - 'a', '\n', '\t', '\u00ff'
- Tekstowe
  - "Hello\n"
- Literał null (pusta referencja do obiektu dowolnego typu))
- Literały klasy (obiekt reprezentujący klasę)
  - String.class

Operator przypisania =

- Operator o wiązaniu prawostronnym
- Przypisania można łączyć (instrukcja przypisania zwraca wartość)

```
int myAge, yourAge;
double pi;
boolean truthy;


pi = 3.14;
truthy = true;
myAge = yourAge = 28;
```

Jawna i niejawna konwersja typów

- Niejawna konwersja:
  - z mniejszych do większych typów liczbowych (byte -> short -> int -> long -> float ->double)
  - char -> int
  - z podtypu do nadtypu (w ramach hierarchii dziedziczenia) (np. String -> Object)
- W innych przypadkach konieczne jest jawne rzutowanie

```
int i = 1;
short s = 3;
byte b;


i = s; // OK
b = s; // Error
b = (byte) s; // OK but information can be lost
```

Operatory (1)

- Arytmetyczne
  - +, -, *, /, % (modulo)
- Inkrementacja, dekrementacja
  - ++, -- (2 warianty: przedrostkowy, przyrostkowy)
- Porównania
  - >, <, >=, <=, ==, != (ich wynikiem jest wartość typu **`boolean`**)
- Logiczne
  - &&, & (**and** z/bez krótkiego wartościowania)
  - ||, | (**or** z/bez krótkiego wartościowania)
  - ^ (**xor**)
  - ! **(not)**
- Bitowe
  - &, |, ^, <<, >>, >>>
- instanceof, ? :

Operatory (2)

- Przypisanie złożone
  - op=, gdzie *op* jest operatorem dwuargumentowym np. +=, -=
- Konkatenacja łańcuchów znaków (obiektów klasy `String`)
  - +, += (jedyny przeciążony operator w Javie)
- Priorytet operatorów

Precedence	Operator	Associativity
1	++ -- + - ~ ! (typ)	R
2	* / %	L
3	+ - +	L
4	<< >> >>>	L
5	< > <= >= instanceof	L
6	== !=	L
7	&	L
8	^	L
9	|	L
10	&&	L
11	||	L
12	?:	L
13	= op=	R

Blok kodu

- Sekwencja instrukcji ograniczona nawiasami klamrowymi
- Może wystąpić w instrukcjach warunkowych i pętlach zamiast pojedynczej instrukcji

```
{
  int i = 1;
  short s = 3;
  byte b;

  i = s;
  b = (byte) s;
}
```

Instrukcja `if`

```
if ( boolean_expr )
    statement1;
else
    statement2;
```

```
if ( a > 10 )
    b = 1;
else
    b = 2;
```

Operator ?:

```
boolean_expr ? expr1 : expr2
```

```
b = (a>10 ? 1 : 2);
```

Instrukcja `switch`

int, byte, short i char oraz typy opakowujące dla nich, enum (od 1.5), String (od 1.7)

```
switch (expr ) {

    case constant_expr1:
        statement1;
        break;
    case constant_expr2:
        statement2;
        break;
    default:
        statement3;
        break;
}
```

```
char c = 'a';
switch ( c ) {
  case 'a': System.out.println('A');
            break;
  case 'b': System.out.println('B');
            break;
  default:  System.out.println('?');
            break;
}
```

A

```
char c = 'a';
switch ( c ) {
  case 'a': System.out.println('A');
  case 'b': System.out.println('B');
  default:  System.out.println('?');
}
```

A
B
?

Pętla `while`

```
while ( boolean_expr )
    statement;
```

```
int[] tab = {10,20,30,40};
int i = 0;
while ( i < tab.length )
{
  System.out.println(tab[i]);
  i++;
}
```

Pętla `do-while`

```
do
    statement;
while ( boolean_expr );
```

```
int[] tab = {10,20,30,40};
int i = 0;
do
{
  System.out.println(tab[i]);
  i++;
}
while ( i < tab.length );
```

Pętla `for`

```
for ( init_expr; boolean_expr; update_expr )
  statement;
```

```
int[] tab = {10,20,30,40};
for (int i = 0; i < tab.length; i++ )
  System.out.println(tab[i]);
```

Pętla `for` : wersja ”for-each” (od 1.5)

```
for ( variable : array/collection )
  statement;
```

```
int[] tab = {10,20,30,40};
for (int elem : tab )
  System.out.println(elem);
```

Instrukcje `break` i `continue`

- **break** służy do opuszczenia pętli lub instrukcji **`switch`**
- **continue** służy do przejścia do następnej iteracji pętli
- Powyższe instrukcje domyślnie "wyskakują" z najbardziej zagnieżdżonej pętli
- Powyższe instrukcje mogą posiadać etykiety
  - możliwość wyskoku z kilku zagnieżdżonych pętli
  - kompensacja braku instrukcji *goto* w Javie

```
outer_loop:
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 5; j++) {
        System.out.println(i, j);
        if (i + j > 7)
            break outer_loop;
    }
}
```

Tablice

- Tablice są obiektami
  - tworzone dynamicznie (`new` lub przy inicjalizacji wartościami ({})
  - zmienne tablicowe są referencjami
- Rozmiar tablic specyfikowany przy tworzeniu
- Indeksowanie tablicy od zera
- Elementy są automatycznie inicjalizowane (dla liczb: zerami, dla obiektów: referencjami pustymi `null`)
- Odwołanie się poprzez niewłaściwy indeks generuje wyjątek
- Rozmiar tablicy dostępny jako pole o nazwie `length`
- Tablice wielowymiarowe jako tablice tablic

Korzystanie z tablic

```
int [] tab = new int[10];
for (int i=0; i < tab.length; i++)
  tab[i] = i;

for (int elem : tab)              // from 1.5
  System.out.println(elem);

int [][] chessBoard;
chessBoard = new int[8][8];

int [] primes = {1,2,3,5,5+2};
String [] verbs = {"go", "sleep"};
```

Programowanie obiektowe

Klasy

```
[ClassModifier] class ClassName [extends Superclass]
      [implements Interface1, ...] {
  // methods and fields
}
```

- ClassModifier może być kombinacją słów kluczowych:
  - modyfikator widzialności
    - public - może być używana w dowolnym miejscu
      - jedna klasa publiczna w pliku (nazwa pliku = nazwa klasy)
    - protected (tylko dla klas zagnieżdżonych) - dostępna w klasach z pakietu i wszystkich podklasach
    - <brak> (domyślny) - dostęp w ramach pakietu, w którym występuje
    - private (tylko dla klas zagnieżdżonych) - dostępna tylko dla metod z tej samej klasy
  - final - nie może posiadać podklas
  - abstract - nie można tworzyć instancji (obiektów) klasy

Metody

```
[MethodModifier] ReturnType MethodName (Type arg1, ...) {
  // method implementation
}
```

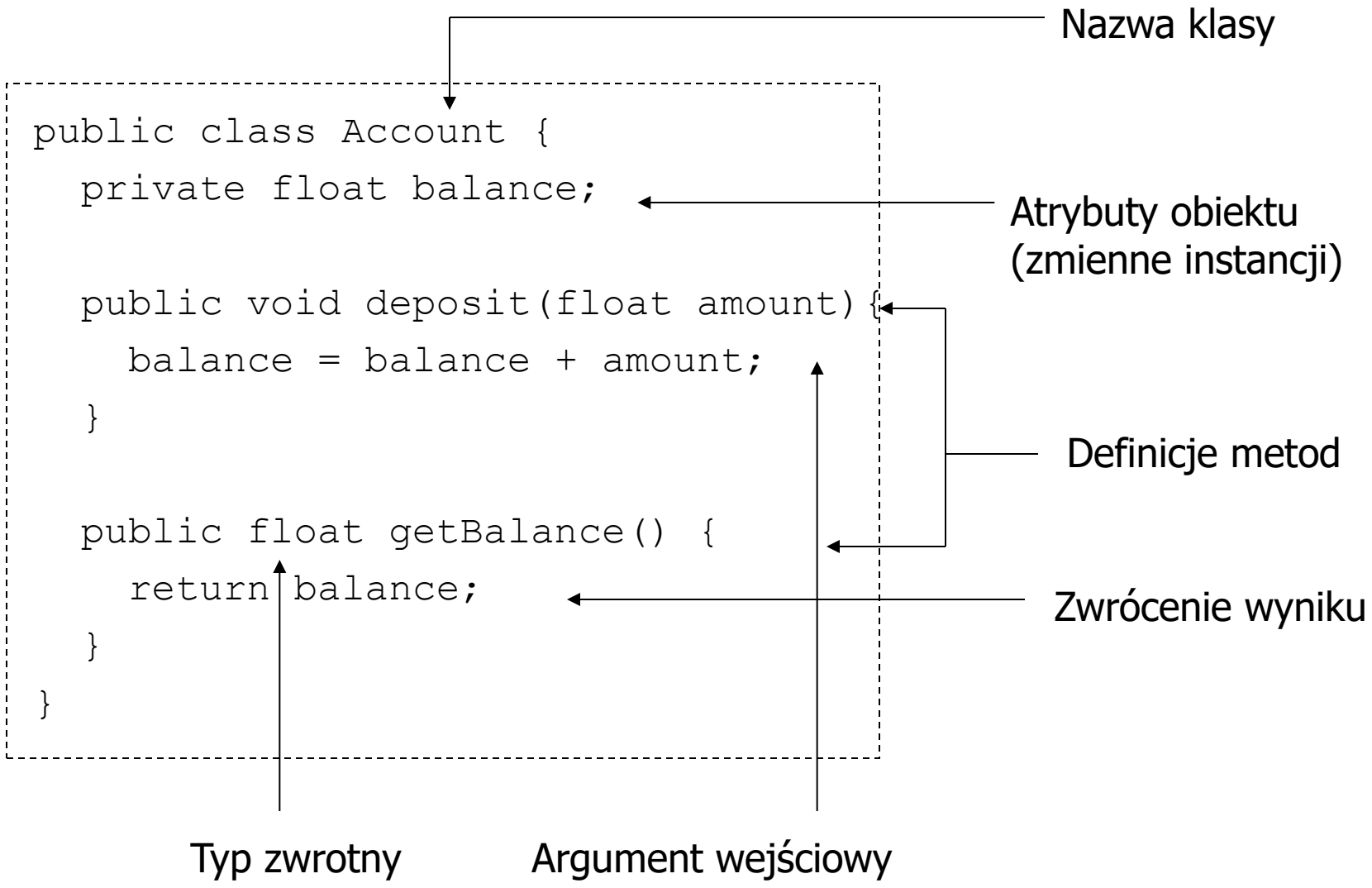
- MethodModifier może być kombinacją słów kluczowych:
  - modyfikator widzialności
    - public - może być używana w dowolnym miejscu
    - protected - dostępna w klasach z pakietu i wszystkich podklasach
    - <brak> (domyślny) - dostępna w klasach z tego samego pakietu
    - private - dostępna tylko w obrębie tej samej klasy
  - static - metoda wołana na rzecz klasy, a nie konkretnego jej obiektu
    - nie może bezpośrednio odwoływać się do składowych instancji (a jedynie do innych składowych statycznych)
  - final - metoda nie może zostać nadpisana w podklasie
  - abstract - metoda bez implementacji
  - synchronized - do synchronizacji wątków działających na obiekcie
  - native - zaimplementowana w innym języku

Pola

```
[FieldModifier] Type FieldName [ = value];
```

- FieldModifier może być kombinacją słów kluczowych:
  - modyfikator widzialności
    - public - może być używane w dowolnym miejscu
    - protected - dostępne w klasach z pakietu i wszystkich podklasach
    - <brak> (domyślny) - dostępne w klasach z tego samego pakietu
    - private - dostępne tylko w obrębie tej samej klasy
  - static – pole obecne w klasie, a nie jej obiektach
    - wspólne dla wszystkich wystąpień obiektu
    - dostępne nawet gdy nie istnieją żadne obiekty klasy
  - final – efektywnie stała, musi być zainicjalizowana przed użyciem

Definiowanie klasy - przykład



Składowe statyczne klasy

```
public class Account {
   static double overdraft = 1000.0;
    public static double getOverdraft() {
        return overdraft;
    }  ...

   static {
     // complex initialization
     // of static fields
   }
}
```

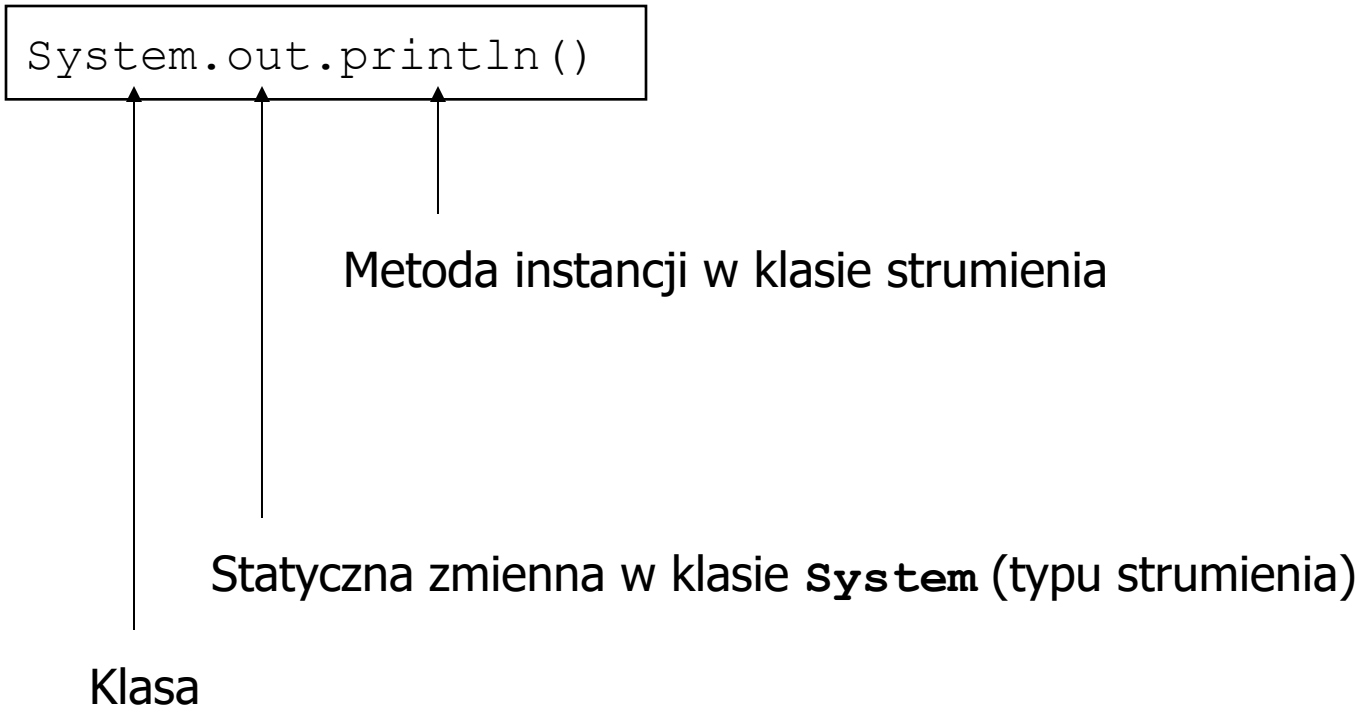
Zmienna klasy (statyczna)

Metoda klasy (metoda statyczna)

Statyczny blok inicjalizacyjny

Przykłady składowych statycznych

- Metoda **`main`**()
- Stałe w klasie **`Math`**: **`Math.PI`**, **`Math.E`**
- Metody klasy **`Math`**: **`Math.sin`**(), **`Math.cos`**()
- Zmienne klasowe w klasie **`System`**, np. **`System.out`**



Typy wyliczeniowe (od 1.5)

- Typy wyliczeniowe (enum) w Javie są klasami
  - Nie ma konwersji do wartości całkowitoliczbowych
  - Mogą posiadać atrybuty i metody
  - Każdy typ wyliczeniowy posiada kilka predefiniowanych metod, np. **`values`**()

```
enum GENDER {FEMALE, MALE};
class Student {
  GENDER gender;

  public Student(GENDER g)   { gender = g; }
  public Student()           { gender = GENDER.FEMALE; }
}
```

```
for (GENDER g : GENDER.values())
  System.out.println(g);
```

```
FEMALE
MALE
```

Tworzenie i korzystanie z obiektów

- Obiekty ZAWSZE tworzone są dynamicznie (zwykle za pomocą `new`)
- Obiekty są dostępne przez referencje
- Obiekty są niszczone przez *garbage collection*, gdy nie ma do nich referencji
  - Niekoniecznie natychmiast i nie ma żadnych gwarancji kiedy
  - Nie można zmusić mechanizmu GC aby zadziałał w określonej chwili
- Odwołanie do składowych obiektu odbywa się przez operator kropki (podobnie do składowych statycznych klasy)

```
Car c = new Car();
Car c1 = c;      // two references to the same object instance
c.move(12,34);
c1 = null;
c = null;        // object is available for garbage collection
                 // (not known when will be destroyed!)
```

Przekazywanie parametrów do metod

- Zmienne typów prostych przekazywane są przez wartość
  - aby zmienić ich wartość w metodzie należy przekazać je w ramach otaczającego obiektu lub jako element tablicy
- Obiekty są przekazywane przez referencję
  - można je zmieniać poprzez przekazaną referencję
  - referencja jest przekazywana przez wartość (nie można podmienić obiektu poprzez zmianę referencji)

```
public class Tests {
   static void timesTwo(int num)
   {
     num = num * 2;
   }
   public static void main(String[] s) {
       int a = 3;
       timesTwo(a);
       System.out.println(a); // 3
   }
}
```

```
public class Tests {
   static void timesTwo (int [] arr)
   {
     arr[0] = arr[0] * 2;
   }
   public static void main(String[] s) {
     int[] t = new int[1];
     t[0] = 3;
     timesTwo(t);
     System.out.println(t[0]); // 6
   }}
```

Konstruktor

- Prosta inicjalizacja zmiennych instancji klasy możliwa przez przypisanie (jak dla statycznych)
  - Domyślna inicjalizacja składowych domyślną wartością dla typu

```
public class Book {
  String category = "SF";
  int numOfPages;   // 0
}
```

- Konstruktor – „metoda” służącą do tworzenia obiektów klasy
  - Umożliwia złożoną inicjalizację zmiennych instancji
- Jego nazwa musi pokrywać się z nazwą klasy
- Konstruktor nie może posiadać typu zwrotnego
- Konstruktory (i normalne metody) mogą być przeciążane
- Przy braku definicji konstruktora, kompilator dostarcza domyślny, pusty konstruktor bezargumentowy

Konstruktor - przykład

```
public class MyClass {
    String t;

    public MyClass(String s) {
     t = s;
    }

    public MyClass() {
     t = "xyz";
    }
}
```

Referencja `this` (1/2)

- Referencja wskazująca na bieżący obiekt
- Pozwala odsłonić składowe klasy przesłonięte przez argumenty metod

```
public class MyClass {
    String t;

    public MyClass(String t) {
     this.t = t;
    }

    public MyClass() {
     t = "xyz";
    }
}
```

Referencja `this` (2/2)

- Umożliwia wywołanie konstruktora z innego konstruktora tej samej klasy
  - Pozwala uniknąć duplikowania kodu

```
public class MyClass {
    String t;

    public MyClass(String t) {
     this.t = t;
    }

    public MyClass() {
     this("xyz");
    }
}
```

Niszczenie obiektu

- Nie ma w Javie destruktora
- Obiekt jest niszczony przez mechanizm *garbage collection*
- Przed zniszczeniem obiektu wywoływana jest metoda `finalize()`
- Ponieważ nie wiadomo kiedy *garbage collection* zadziała, krytyczne zasoby należy zwalniać jawnie, a nie w metodzie `finalize()`
  - Metoda finalize() jest w zasadzie bezużyteczna
- Od Javy 7 dostępna jest specjalna wersja instrukcji **`try`** ułatwiająca zarządzanie zwalnianiem zasobów

Dziedziczenie klas

```
public class Account{
  protected float balance;
  public void deposit(float amount) {
    balance = balance + amount;
  }
  public float getBalance() {
    return balance;
  }
}
```

Nazwa nowej klasy (podklasy)

Nazwa nadklasy

Nadpisanie odziedziczonej metody

```
public class BonusAccount extends Account{

  public void deposit(float amount) {
    balance = balance + amount * 1.1;
  }
}
```

- Zawsze dokładnie jedna klasa bazowa
  - Domyślnie **`java.lang.Object`** (korzeń hierarchii klas)

Klasa Object

- W języku Java każda klasa dziedziczy bezpośrednio lub pośrednio z **`java.lang.Object`**
  - Klasa **`Object`** jest korzeniem hierarchii klas
  - Każdy obiekt posiada implementacje metod klasy **`Object`**
- Metody klasy **`Object`**:
  - **`clone`**()
    - Do klonowania (tworzenia kopii obiektu)
    - Działa tylko dla klas implementujących interfejs **`Cloneable`**
  - **`equals`**()
    - Domyślnie porównuje przez ==
    - Nadpisana w **`String`**, **`Integer`**, …
  - **`hashCode`**()
  - **`finalize`**()
  - **`toString`**()
  - **`getClass`**()
  - **`notify`**(), **`notifyAll`**(), and **`wait`**() (do synchronizacji wątków)

Przeciążanie a nadpisywanie metod w podklasie

- Metoda w klasie może być **przeciążona** (ang. overload)
  - wiele metod o tej samej nazwie różniących się liczbą i/lub typami parametrów
- Metoda może zostać **przeciążona** w podklasie
  - w podklasie dostępne będą obie wersje metody
- Metoda **nadpisuje** (ang. override) metodę z nadklasy gdy ma taką samą nazwę, liczbę i typy parametrów i taki sam typ zwrotny
  - Zakres widzialności metody nadpisującej może być większy niż nadpisywanej
  - Typ zwrotny metody nadpisującej może być podklasą typu zwrotnego nadpisywanej
  - Nadpisująca nie może rzucać wyjątków, których nie rzuca metoda nadpisywana

```
public class MyClass {
  // overriding inherited method from Object
  public String toString() {...}
  // overloading
  public String toString(String prefix) {...}
}
```

Nadpisywanie i przesłanianie metod (1/2)

```
public class Animal {
  public static void hide() {
    System.out.println("The hide method in Animal.");
  }
  public void override() {
    System.out.println("The override method in Animal.");
  }
}
```

```
public class Cat extends Animal {
  public static void hide() {
    System.out.println("The hide method in Cat.");
  }
  public void override() {
    System.out.println("The override method in Cat.");
  }
}
```

Nadpisywanie i przesłanianie metod (2/2)

```
public class Animal {
  public static void hide() {    ... }
  public void override() { ... } }
```

```
public class Cat extends Animal {
  public static void hide() { ... }
  public void override() { ... }
  public static void main(String[] args) {
    Cat myCat = new Cat();
    Animal myAnimal = myCat;
    myAnimal.hide();       // The hide method in Animal.
    myAnimal.override();   // The override method in Cat.
} }
```

- Metody statyczne są **przesłaniane** w podklasie
  - Tylko statyczna może przesłonić statyczną
- Metody instancji są **nadpisywane** w podklasie (zawsze są **wirtualne**!)
  - Tylko metoda instancji może nadpisać metodę instancji

Przesłanianie pól w podklasie

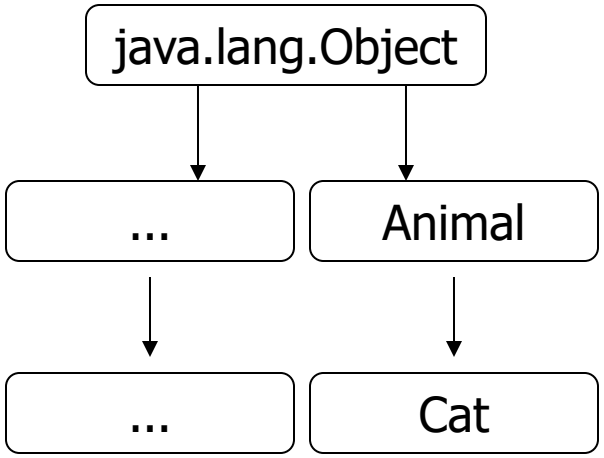
- Pole w podklasie o takiej samej nazwie jak pole w nadklasie przesłania je (niezależnie od typu danych)
  - Dostęp do przesłoniętych zmiennych instancji przez super, np. `super.weight`
  - Dostęp do zmiennych statycznych przez prefiksowanie nazwą klasy, np. `Animal.maxWeight`
  - Przesłanianie pól nie jest zalecane

Referencja `super`

- W metodach klasy można używać referencji **`super`**
  - Na podobnej zasadzie jak **`this`**
- Za pomocą **`super`** można uzyskać dostęp do przesłoniętych składowych instancji (metod i zmiennych niestatycznych) nadklasy (`super.member`)
- Z konstruktora klasy można wywołać konkretny konstruktor nadklasy (`super(...)`), w pierwszej instrukcji ciała konstruktora
  - Gdy konstruktor nadklasy nie zostanie wywołany jawnie, to zostanie wywołany konstruktor bezargumentowy
    - Przy jego braku w nadklasie – będzie błąd kompilacji

Hierarchia dziedziczenia

- Każda klasa może posiadać jedną nadklasę
- Korzeniem hierarchii dziedziczenia jest klasa `Object`
- Gdy dla danej klasy nie zostanie podana jej nadklasa, domyślnie przyjmowana jest klasa `Object`
- Konwersja typów (jawna lub niejawna) możliwa jest w ramach gałęzi dziedziczenia

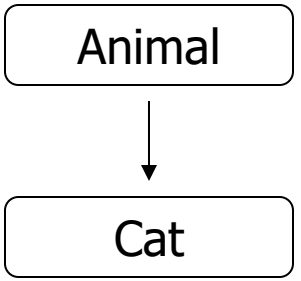



```
Cat filemon = new Cat();
Animal a = filemon; // OK
filemon = a; // error
filemon = (Cat) a; // OK
```

Zastępowanie referencji do obiektów

- Obiekt podklasy może być użyty tam, gdzie spodziewany jest obiekt nadklasy
- Prawdziwy typ obiektu może być określony przy użyciu operatora `instanceof`
- Powyższe stwierdzenia odnoszą się również do interfejsów implementowanych przez klasę obiektu

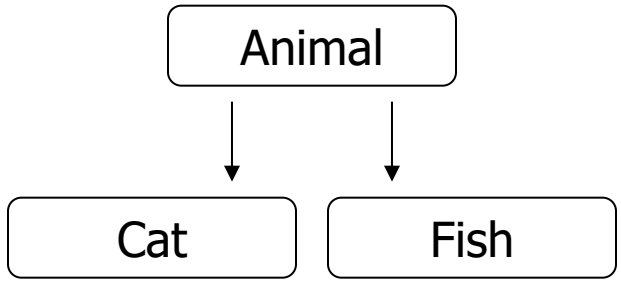
```
public void rename (Animal a) {
   if (a instanceof Cat)
   {
      ((Cat) a).catMethod();
   }
...
}
```



Polimorfizm

- Polega na dynamicznym (późnym) wiązaniu metod
- W przypadku wywołania metody obiektu podklasy poprzez referencję typu nadklasy, wywoływana jest metoda z jego klasy (na podstawie jego faktycznego typu)
  - W Javie wszystkie metody instancji (niestatyczne) są wirtualne

```
Animal [] zoo = {
                  new Cat(...),
                  new Fish(...),
                  ...
                  };
for (int i=0; i < zoo.length; i++)
  zoo[i].feed();
```



Modyfikator `final` (podsumowanie)

- Dostępny dla zmiennych, metod i klas
- Klasa **`final`** nie może posiadać klas pochodnych (podklas)
  - Nie można po niej dziedziczyć
- Metody **`final`** nie można nadpisać w klasie pochodnej
- Zmienna **`final`** jest stałą
  - Musi mieć nadaną wartość przed wykorzystaniem

Klasy i metody abstrakcyjne

- Klasa abstrakcyjna to klasa, której nie można wykorzystać do tworzenia instancji
  - Deklarowana z modyfikatorem **`abstract`**
- Metoda abstrakcyjna to metoda nieposiadająca implementacji
  - Deklarowana z modyfikatorem **`abstract`** i nieposiadająca ciała
- Klasa posiadająca chociaż jedną metodę abstrakcyjną musi być zadeklarowana jako abstrakcyjna

```
abstract class Shape
{
  abstract double perimeter();
  abstract double area();
}
```

```
class Square extends Shape
{
  double side;
  double perimeter() {
    return 4 * side;
  }
  double area()  {
    return Math.pow(side, 2);
  }
}
```

Dziedziczenie i polimorfizm – Przykład (1/3)

```
public class Person {
  private String lastName;
  public Person(String ln) {
    lastName = ln;
  }
  public String toString() {
    return "Name: " + lastName;
  }
}
```

Dziedziczenie i polimorfizm – Przykład (2/3)

```
public class Employee extends Person {
  private int salary;
  public Employee(String ln, int sal) {
    super(ln);
    salary = sal;
  }
  public String toString() {
    return super.toString()+ " salary: " + salary;
  }
}
```

Dziedziczenie i polimorfizm – Przykład (3/3)

```
Person[] arr = new Person[2];
arr[0] = new Person("Kowalski");
arr[1] = new Employee("Smith", 6000);

for (Person p : arr) System.out.println(p);
```

Tu będzie wywołana metoda toString()

```
Name: Kowalski
Name: Smith salary: 6000
```

Interfejsy

- Interfejs przypomina w pełni abstrakcyjną klasę
- Wszystkie jego metody niestatyczne muszą być abstrakcyjne (domyślnie `public abstract`)
  - Od Javy 8 możliwe domyślne implementacje metod niestatycznych (`default`)
  - Od Javy 8 interfejs może zawierać metody statyczne (z ciałami)
  - Od Javy 9 interfejs może zawierać metody prywatne (z ciałami)
- Nie może posiadać zmiennych wystąpień (wszystkie pola są domyślnie `public static final`)
- W pewnym stopniu kompensują brak dziedziczenia wielobazowego (wielokrotnego)
- Korzystanie z interfejsów zwiększa elastyczność kodu aplikacji i jest generalnie zalecanym stylem programowania

```
[public] interface Name [extends Interface1, ...] {
  // methods and static fields
}
```

Implementacja interfejsu przez klasę

- Klasa może implementować zero, jeden lub więcej interfejsów

```
interface Printable {
   void print();
}

class Document implements Printable {
   public void print() {
   // specific implementation
   }
}
```

```
...
Printable p = new Document();
p.print();
...
```

Domyślne implementacje metod interfejsu (Java 8)

- Umożliwiają dodanie nowych metod do interfejsu bez konieczności aktualizacji kodu istniejących klas implementujących interfejs

```
interface InterfaceA {
    default void newMethod() {
        System.out.println("Default method of Interface A");
    }
}

public class Impl implements InterfaceA {  // OK
   // inherits default implementation od newMethod()
}
```

Konflikt domyślnych implementacji metod interfejsu (Java 8)

```
interface InterfaceA {
    void oldMethod();

    default void newMethod() {
        System.out.println("Default method of InterfaceA");
    }
}

interface InterfaceB {
    void oldMethod();

    default void newMethod() {
        System.out.println("Default method of InterfaceB");
    }
}

public class Impl implements InterfaceA, InterfaceB {
    public void oldMethod() {
        System.out.println("Method implementing both interfaces"); // OK
    }

    // compilation error - conflicting default implementations of newMethod()
    // inherited from interfaces
}
```

Konflikt domyślnych implementacji metod interfejsu (Java 8) - Rozwiązanie

```
interface InterfaceA {
    default void newMethod() {
        System.out.println("Default method of InterfaceA");
    }
}

interface InterfaceB {
    default void newMethod() {
        System.out.println("Default method of InterfaceB");
    }
}

public class Impl implements InterfaceA, InterfaceB {
    // explicit method implementation
    public void newMethod() {
        // may call inherited default implementations
        InterfaceA.super.newMethod();
        InterfaceB.super.newMethod();
    }
}
```

Klasy zagnieżdżone

- Klasy można definiować jako składowe innych klas
  - statyczne klasy zagnieżdżone (static nested classes)
    - mają dostęp jedynie do składowych statycznych klasy otaczającej
  - klasy wewnętrzne (inner classes)
    - ich obiekty „żyją wewnątrz” obiektów klasy otaczającej i mają swobodę dostępu do ich składowych
    - Nie mogą zawierać statycznych składowych
  - klasy lokalne (local classes) – zagnieżdżone w bloku kodu (najczęściej w metodzie)
    - Szczególny przypadek klasy wewnętrznej (dotyczą ich ograniczenia i możliwości klas wewnętrznych)
    - Mają dostęp do lokalnych zmiennych metody, które są final (lub efektywnie final) oraz jej parametrów (od Java 8)
    - Nie mogą mieć kwalifikatora widzialności
- Cele stosowania: enkapsulacja, grupowanie klas używanych w jednym miejscu, czytelność kodu

Klasy zagnieżdżone - Przykład

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

  void method() {
    class LocalClass { ...
    }
  }
}
```

Statyczna klasa zagnieżdżona

Klasa wewnętrzna

Klasa lokalna

Klasy zagnieżdżone – specyficzne konstrukcje

- Korzystanie z klasy zagnieżdżonej (statycznej) poza jej klasą otaczającą (jeśli widzialność pozwala)

```
OuterClass.NestedClass no = new OuterClass.NestedClass()
```

- Korzystanie z klasy wewnętrznej poza jej klasą otaczającą (jeśli widzialność pozwala)

```
OuterClass.InnerClass io = outerObj.new InnerClass()
```

- Odsłonięcie w klasie zagnieżdżonej składowej klasy otaczającej zasłoniętej identyczną nazwą w klasie zagnieżdżonej

```
OuterClass.this.x
```

Klasy anonimowe

- Specjalny rodzaj klas lokalnych
- Klasy nieposiadające nazwy
- Używane gdy klasa ma być użyta tylko raz
- Instancja tworzona na bazie rozszerzanej nadklasy / implementowanego interfejsu
- Nie mogą posiadać konstruktora (mogą blok inicjalizacyjny)

```
SuperClass anon = new SuperClass() {
            String field = "?";
            public void method() {
                ...
            }
            {
                ...
            }
        };
```

Dodane pola i metody, nadpisane metody

Blok inicjalizacyjny

Średnik!

Interfejsy zagnieżdżone

- Interfejsy można zagnieżdżać w klasach i interfejsach
- Interfejsy zagnieżdżone są automatycznie statyczne
- Interfejsy nie mogą być lokalne
  (tzn. zagnieżdżone w metodach)

```
class MyClass {
    interface NestedInterface {
        interface NestedNestedInterface {
            ...
        }
        ...
    }
    ...
}
```

Wyjątki

Wyjątki

- Stanowią wydajny i przejrzysty mechanizm obsługi błędów
- Przykłady (podklasy klasy `Throwable`)
  - błędy (podklasy `Error`) np.:
    - `OutOfMemoryError`, `InternalError`
  - wyjątki kontrolowane (podklasy `Exception`), np.:
    - `MalformedURLException`, `IOException`
  - wyjątki czasu wykonania (podklasy `RuntimeException`), np.:
    - `ArrayIndexOutOfBoundsException`, `ArithmeticException`
- Wszystkie wyjątki kontrolowane (poza wyjątkami czasu wykonania) muszą być obsłużone lub wymienione w klauzuli `throws` w deklaracji metody

```
public void insertRow() throws SQLException {...}
```

Przechwytywanie i obsługa wyjątków

```
try {
  // statements that may throw exceptions
}
catch (ExceptionClass1 e)
{ ... }
catch (ExceptionClass2 e)
{ ... }
...
finally
{
  // executed always (!), optional section
}
```

Zgrupowana obsługa wyjątków (od 1.7)

```
try {
  // statements that may throw exceptions
}
catch (ExceptionClass1 | ExceptionClass2 e)
{ ... }
...
finally
{
  ...
}
```

Obsługa zdarzeń wyjątkowych - Przykład

```
File f = new File("data.txt");

InputStream is = null;

try {
  is = new FileInputStream(f);
  ...
}
catch (FileNotFoundException e) {
  System.err.println(e);
}
finally {
  if (is != null) {
    is.close();
  }
}
```

Instrukcja `try` z zasobami (od 1.7)

- Ułatwia zagwarantowanie, że zasób zostanie zwolniony po jego użyciu niezależnie od tego czy wystąpi wyjątek czy nie
  - Przed catch / finally
- Klasa zasobu musi implementować interfejs **`AutoCloseable`**
  - Interfejs obejmuje jedną metodę: **`close()`**
  - W Java 7 wiele klas bibliotecznych zmodyfikowano o implementację tego interfejsu (np. związane z plikami, bazą danych, itp.)

```
File f = new File("data.txt");
try (InputStream is = new FileInputStream(f)) {
   ...
}
catch (FileNotFoundException e) {
  System.err.println(e);
}
```

Rzucanie wyjątków

- Wyjątek można rzucić jawnie

```
throw new IOException();
```

- Metody, które mogą zgłaszać wyjątki, muszą zawierać informację o tym w deklaracji

```
public void method1() throws IOException {
  // ...
}
```

Przykład definicji wyjątku użytkownika

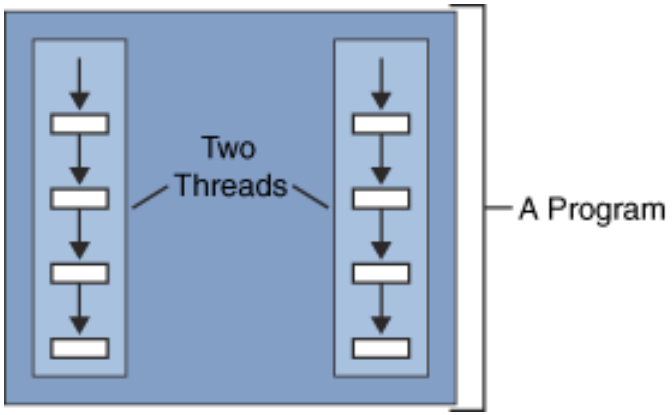
```
public class TestException extends Exception {
   public TestException() {} ;

   public TestException (Object o, String method, Exception e)
   { this(e.getClass() + " in <" + o.getClass() + "> ( " +
       method + ") :: " + e.getMessage());
   e.printStackTrace();
   }

   public TestException(String msg) { super(msg); }
}
```

Wątki

Wątki



- Wątek to pojedynczy sekwencyjny przepływ sterowania w programie
- Java umożliwia tworzenie programów wielowątkowych
- Wsparcie dla wątków na poziomie języka:
  - Klasa **`Thread`** implementująca wątek
  - Interfejs **`Runnable`**, pozwalający na uruchomienie klasy, która go implementuje w nowym wątku (zawiera jedną metodę **`public void run`**())
  - Klasa **`Object`**, zawierająca metody **`wait`**, **`notify`** i **`notifyAll`** do synchronizacji wątków
  - Słowo kluczowe **`synchronized`** do oznaczania metody lub bloku kodu jako sekcji krytycznej
  - Słowo kluczowe **`volatile`**
- Sposoby implementacji wątków
  - Dziedziczenie z klasy **`Thread`**
  - Wykorzystanie oryginalnej klasy **`Thread`**, przekazując konstruktorowi wątku obiekt klasy implementującej **`Runnable`**

Wątki - Przykład

```
class Task implements Runnable
{
  public void run()
  {
    for (int i=0; i<5; i++) {
      try { Thread.sleep(1000);
          } catch (InterruptedException e){ }
      System.out.println(i);
    }
    System.out.println("Done!");
  }
}
```

```
0
0
1
1
2
2
3
3
4
Done!
4
Done!
```

```
Thread t1 = new Thread(new Task());
Thread t2 = new Thread(new Task());
t1.start();
t2.start();
```

Refleksja

Mechanizm Java Reflection

- Umożliwia odczyt informacji o klasach i obiektach (konstruktory, metody, pola) w trakcie pracy programu
- Przydatny dla dynamicznie ładowanych klas
- Pozwala obejść enkapsulację!
- Realizowany poprzez klasy z pakietu **`java.lang.reflect`** (np. **`Field`**, **`Method`**, **`Constructor`**) i klasę **`java.util.Class`** (jej przykładowe metody: **`getFields()`**, **`getMethods()`**, **`getConstructors()`**, **`getField()`**, **`getMethod()`**, **`getConstructor()`**)

```
    Class cl = ob1.getClass();
    Field f1 = cl.getField("lastName");
    String ln = (String) f1.get(ob1);
```

Tworzenie instancji klas wskazanych w trakcie pracy programu

- Java jest językiem dynamicznym
  - Umożliwia wykorzystanie klas, których nazwy nie były znane podczas tworzenia i kompilacji programu

```
String className = "test.Person";
Object ob1 = null;
try {
  Class c = Class.forName(className);
  ob1 = c.newInstance();
} catch (InstantiationException e) {
    System.out.println(e);
} catch (IllegalAccessException e) {
    System.out.println(e);
} catch (ClassNotFoundException e) {
    System.out.println(e);
}
```

Generics

Generics

- Umożliwiają używanie typów (klas i interfejsów) do parametryzacji:
  - klas
  - interfejsów
  - metod
- Cele stosowania:
  - Ściślejsza kontrola typów, uniknięcie rzutowania
  - Implementacja ogólnych algorytmów (raz dla różnych typów danych)
- Przykłady:
  - Collection<E>, Map<K,V>, Comparable<T>, Optional<T>, Future<T>
- Uwaga: Informacja o typie użytym faktycznie jako parametr jest w Javie usuwana na etapie kompilacji (type erasure)!

Klasa generyczna - Przykład

```
class Pair<T> {
  public T left;
  public T right;

  public Pair(T l, T r) {
    left = l; right = r;
  }

  public void swap() {
    T temp;
    temp = left; left = right; right = temp;
  }

  public String toString() {
    return "[" + left + ";" + right + "]";
  }
}
```

Klasa generyczna – Przykład użycia

```
Pair<Cat> pc = new Pair<Cat>(new Cat("Tom"),
                             new Cat("Jinx"));

System.out.println(pc.toString());
pc.swap();
System.out.println(pc.toString());
```

Klasa generyczna - Problem

- Hierarchia klas

```
abstract class Animal { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
```

- Metoda działająca na parach zwierząt

```
static void swapAnimalPair(Pair<Animal> p) {
  System.out.println(p.toString());
  p.swap();
  System.out.println(p.toString());
}
```

- Próba wywołania dla pary kotów…

```
Pair<Cat> pc = new Pair<Cat>(new Cat("Tom"),
                             new Cat("Jinx"));
swapAnimalPair(pc); // compilation err - incompatible types
```

Rozwiązanie: wildcards

- Unbounded: `?`
- Bounded:
  - **`? extends Type`**
  - **`? super Type`**

```
// OK, but pairs of arbitrary objects allowed
static void swapAnimalPair(Pair<?> p) {
    ...
}
```

```
// Only pairs of Animal objects allowed
static void swapAnimalPair(Pair<? extends Animal> p) {
    ...
}
```

Generyki a dziedziczenie

- Klasa generyczna jako podtyp niegenerycznej

```
class NonGeneric {}

class Generic<T> extends NonGeneric {}
```

- Klasa generyczna jako podtyp generycznej

```
class SubGeneric<T> extends Generic<T> {}

class SubExtraGeneric<T, U> extends Generic<T> {}

class SubNonGeneric extends Generic<String> {}

class SubtypedGeneric<U> extends Generic<String> {}
```

Rekursja w generykach (wersja „lite”)

- Przykład generycznego interfejsu: Comparable<T>

```
public interface Comparable<T> { int compareTo(T o); }
```

- Przykład zalecanej implementacji Comparable<T>

```
class User implements Comparable<User> {
  public int compareTo(User user) { ... }
 }
```

- Przykład implementacji Comparable jako „raw type”

```
class User implements Comparable {
  public int compareTo(Object user) {
     User castedUser = (User) user;
     ...                  // risk of ClassCastException
   }
 }
```

Rekursja w generykach (wersja „hard”)

```
abstract class Sequence<T extends Sequence<T>> {
    public abstract T merge(T seq);
}

class DoubleSequence extends Sequence<DoubleSequence> {
    ...
    @Override
    public DoubleSequence merge(DoubleSequence ts) {
        return ...;
    }
}
```

- Wyjaśnienie:
  - Definiujemy ogólne pojęcie sekwencji, przewidując specjalizowane implementacje
  - Chcemy zapewnić polimorfizm dla metody merge, jednocześnie wymagając, aby sekwencje danego podtypu mogły być łączone tylko z sekwencjami tego samego podtypu

Organizacja kodu

Organizacja kodu w języku Java

- W jednym pliku źródłowym Java może znajdować się tylko jedna klasa publiczna
  - Plik może zawierać wiele klas niepublicznych lub zagnieżdżonych
  - Po kompilacji KAŻDA klasa jest umieszczana w osobnym pliku *.class
- Nazwa pliku MUSI być taka sama jak nazwa klasy publicznej w nim definiowanej
- Nie ma pojęcia programu głównego, nie ma zmiennych globalnych, cały kod jest zawarty w klasach

Account.java

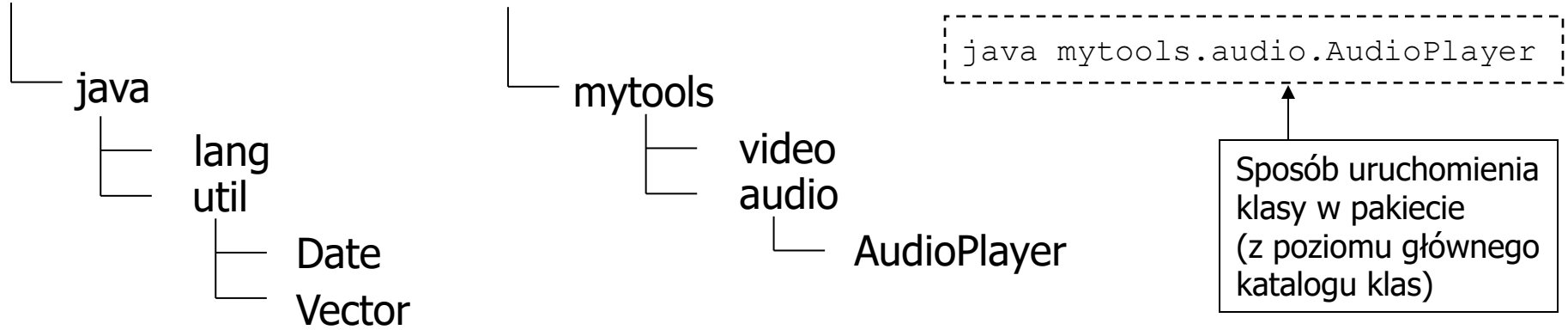
```
public class Account {
   float balance;
   public void deposit(float amount){
     balance = balance + amount;
   }
   public float getBalance() {
     return balance ;
   }
}
```

Grupowanie klas w pakiety

- Informacja o przynależności do pakietu w pierwszej linii jednostki kompilacji (przy braku - pakiet domyślny)

```
package mytools.audio;
class AudioPlayer { ... }
```

- Lokalizacja pliku `.class` w drzewie katalogów musi odpowiadać zadeklarowanej nazwie pakietu



- W praktyce, biblioteki klas i aplikacje są pakowane do plików JAR (archiwa w formacie ZIP, niekoniecznie skompresowane)

Importowanie klas

- Odwołanie do klasy z innego pakietu wymaga prefiksu:

```
mytools.audio.AudioPlayer ap;
```

- Dzięki zastosowaniu polecenia `import`, możliwe jest odwoływanie się do klas z innych pakietów bez prefiksu
  - domyślnie importowany jest pakiet java.lang

```
import mytools.audio.AudioPlayer;
```

```
import mytools.audio.*;
```

- W celu załadowania wskazywanej klasy, JVM przeszukuje wszystkie lokalizacje zapisane w zmiennej środowiskowej CLASSPATH
  - Aby import był możliwy, klasa (też interfejs, enum) musi być fizycznie dostępna (w kodzie aplikacji lub dołączonych bibliotekach)

Statyczne importowanie klas (od 1.5)

- Umożliwia odwołania do statycznych stałych i metod innej klasy bez konieczności prefiksowania ich nazwą klasy

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;

public class Test {
  public static void main(String[] args) {
    System.out.println(Math.PI);
    System.out.println(PI);
    System.out.println(Math.sin(0));
    System.out.println(sin(PI/2));
  }
}
```

Klasa biblioteczna z funkcjami matematycznymi oraz stałymi E i PI

```
public final class java.lang.Math {
  public static final double PI = 3.141592653589793;
  public static double sin(double arg) {...}
  ...
}
```

Konwencje dotyczące nazw

- Konwencje nie oznaczają wymagań!
  - Nazwy niezgodne z przyjętymi konwencjami nie spowodują błędów kompilacji
  - Stosowanie się do konwencji poprawia czytelność kodu
- Konwencje powszechnie przyjęte w języku Java:
  - Nazwy klas z dużej litery, bez podkreśleń, kolejne człony nazwy z dużych liter
    - Vehicle, Car, SportsCar
  - Nazwy metod i zmiennych z małej litery, bez podkreśleń, kolejne człony nazwy z dużych liter
    - speed, numberOfPassengers, stepOnTheBrake()
  - Nazwy pakietów małymi literami
    - mycompany.data, entities, java.lang
  - Nazwy stałych dużymi literami, kolejne człony oddzielane podkreśleniami
    - MIN_LENGTH, MAX_NUM_PERSONS

Java Beans

- JavaBeans to komponenty posiadające właściwości, które można ustawiać i odczytywać
- Właściwość definiowana przez parę metod get/set (accessor/mutator)
- Implementują ideę hermetyczności obiektów (encapsulation)
- Początkowo koncepcja opracowana z myślą o komponentach graficznych, które miały mieć własności ustawiane przez paletę właściwości w środowisku IDE

```
public class Bean
{
  private String type;
  private boolean edible;
  public Bean() {}
  public String getType() {return type;}
  public void setType(String type)
  { this.type = type; }
  public boolean isEdible() {return edible;}
  public void setEdible(boolean edible)
  { this.edible = edible; }
}
```

Adnotacje (od 1.5)

- Oznaczenia kodu umożliwiające automatyczną generację kodu towarzyszącego i konfigurację aplikacji
- Nie wpływają bezpośrednio na semantykę programu, ale wpływają na sposób traktowania programu przez narzędzia i biblioteki
- Koncepcja podobna do komentarzy Javadoc, ale Javadoc służy tylko do dokumentowania kodu
- Predefiniowane adnotacje używane przez kompilator języka Java:
  - @Deprecated, @Override, @SuppressWarnings
- Adnotację definiuje się jako specyficzną formę interfejsu
  - W praktyce programiści aplikacji nie definiują nowych typów adnotacji, a jedynie wykorzystują adnotacje oferowane przez poszczególne biblioteki i narzędzia

```
public @interface Copyright {
String value(); }
```

```
@Copyright("2018 Code Masters")
public class ProblemSolver { ... }
```

Przykłady popularnych adnotacji

- Przykładowe wbudowane adnotacje JDK
  - @Override
  - @Deprecated
- Przykładowe adnotacje EJB
  - @Stateless
  - @Stateful
- Przykładowe adnotacje JPA
  - @Entity
  - @Table
  - @Id
  - @ManyToOne
- Przykładowe adnotacje Web Services
  - @WebService
  - @WebMethod

Adnotacja @Override (od 1.5)

- Informuje kompilator, że intencją programisty jest nadpisanie metody z nadklasy
  - Dotyczy to również implementacji metody interfejsu
  - Błąd kompilacji gdy tak nie jest ze względu na niezgodne nagłówki metod
- Zalecane jest jej używanie zawsze przy nadpisywaniu metod
  - Gdy nadpisywana jest metoda, a nie ma adnotacji @Override, kompilator zgłasza ostrzeżenie (nie błąd, bo adnotacja jest nieobowiązkowa)
  - Ostrzeżenie kompilatora wynika z faktu, że nieświadome nadpisanie metody może prowadzić do nieoczekiwanego zachowania się aplikacji

```
class MyClass {                          // extends Object
  ...
  @Override
  public String toString() {
            return ...;
  }
}
```

Klasy biblioteczne (Przykłady)

Klasa java.lang.String

- Służy do reprezentowania niemodyfikowalnych ciągów znaków Unicode
- Metody klasy `String`
  - `length(), charAt(int), equals(String), compareTo(String), substring(int beginIndex, int endIndex),...`
- Każda klasa posiada metodę `toString()`, konwertującą jej obiekty do tekstów
- Tworzenie obiektów String
  - `new` wymusza utworzenie nowej instancji
  - Tworzenie przez przypisanie literału pozwala JVM na optymalizację poprzez współdzielenie reprezentacji obiektu w pamięci

```
String s1 = "To be";
String s2 = new String (" or not to be"); // not recommended
String s3 = s1 + " " + s2;
```

Porównywanie obiektów String

- Operator == nie został przeciążony w klasie String
  - Sprawdza czy referencje są identyczne
- Została nadpisana metoda `equals()` i jest ona bezpiecznym mechanizmem sprawdzania czy teksty są sobie równe

```
String s1 = "Str";
String s2 = "Str";
if (s1 == s2) // true (!)
{  ... }
```

```
String s1 = new String("Str");
String s2 = new String("Str");

if (s1 == s2) // false (!)
{  ... }
```

```
String s1 = "Str";
String s2 = "Str"
if (s1.equals(s2)) // true
{  ... }
```

```
String s1 = new String("Str");
String s2 = new String("Str");

if (s1.equals(s2)) // true
{  ... }
```

Klasa java.lang.StringBuilder

- Zachowuje większość funkcjonalności `String`: `length(),charAt(int),...`
- Reprezentuje modyfikowalne ciągi znaków:
  - dodatkowe metody: `append (String s), delete(int st, int end), insert(int offset, String s), reverse(), append(StringBuilder sb), delete(int st, int end), insert(int offset, StringBuilder sb), reverse(),...`
- Konwersja do `String`:
  - wywołanie metody `toString()`
  - przekazanie jako parametr konstruktora `String`
- Porównanie wydajności `StringBuilder` i `String`:
  - kosztowniejsza alokacja `StringBuilder`
  - operacje na `StringBuilder` wydajniejsze od konkatenacji `String`

```
String x = "a" + 4;
String y = new StringBuilder().append("a").append(4).toString();
```

Istnieje również starsza klasa `StringBuffer`, mniej wydajna przez synchronizację

Klasy opakowujące (ang. *wrapper*)

- Dla każdego typu prostego w Javie istnieje odpowiadająca mu klasa "wrapper" (`Integer`, `Character`, `Float`, ...)
- Klasy "wrapper" zastępują zmienne proste w miejscach gdzie spodziewane są obiekty
  - Możliwość przypisania null
  - Przechowywanie w kolekcjach
- Zawierają metody zwracające proste wartości (`intValue()`, `floatValue()`, ...)
- Ich instancje są niemodyfikowalne
- Zawierają przydatne metody konwersji typów np.
  - `Integer.parseInt(String)` - String -> int
  - `Integer.toString(int)` - int -> String

Automatyczna konwersja miedzy typami prostymi i klasami opakowującymi (od 1.5)

- Dla wygody programistów od wersji 1.5 języka Java dokonywane są automatyczne konwersje między typem prostym i odpowiadającym mu typem opakowującym
  - Mechanizm „auto-boxing” / „auto-unboxing”
  - Konwersji nie widać w kodzie, ale w dalszym ciągu wiąże się z narzutem czasowym
  - Mechanizm ten nie oznacza, że należy zaniechać używania typów prostych

```
int x = 5; int y = 4;
Integer [] arr = new Integer[3];
arr[0] = new Integer(x);
arr[1] = new Integer(y);
arr[2] = null;
System.out.println(arr[0].intValue()
                + arr[1].intValue());
```

```
int x = 5; int y = 4;
Integer [] arr = new Integer[3];
arr[0] = x;
arr[1] = y;
arr[2] = null;
System.out.println(arr[0] + arr[1]);
```

Klasa java.lang.Math

- Klasa `java.lang.Math` pełni rolę biblioteki matematycznej
  - wszystkie jej metody są statyczne (nie jest konieczne tworzenie obiektów)

`Math.abs(x)`	`Math.acos(x)`	`Math.asin(x)`
`Math.atan(x)`	`Math.ceil(x)`	`Math.cos(x)`
`Math.exp(x)`	`Math.floor(x)`	`Math.log(x)`
`Math.max(x,y)`	`Math.min(x,y)`	`Math.pow(x,y)`
`Math.random()`	`Math.round(x)`	`Math.sin(x)`
`Math.sqrt(x)`	`Math.tan(x)`	

- Klasa ta zawiera również stałe reprezentujące liczby π i e

`Math.PI`	`Math.E`

```
double sqRoot;
sqRoot = Math.sqrt(2.0);
```

```
double twoPi;
twoPi = 2 * Math.PI;
```

Obsługa dat i czasu

- Klasa `java.util.Date` służy do tworzenia obiektów reprezentujących datę i czas
  - z założenia UTC
  - z dokładnością do milisekund
  - Wiele metod (formatowanie, składniki daty i czasu) zdeprecjonowanych (problemy roku 2000 i z internacjonalizacją)
- Klasa `java.util.Calendar` obsługuje składniki daty i czasu
- Klasa `java.util.TimeZone` reprezentuje strefę czasową
- Nowe typy w Java 8: pakiet `java.time.*`, niemodyfikowalne obiekty, odrębne typy dla daty i czasu, czas z uwzględnieniem lub bez strefy czasowej
  - `LocalDate`, `LocalDateTime`, `LocalTime`, `Instant`, `Duration`, `ZoneId`, `ZonedDataTime`, …

Data i czas – Przykłady (stary i nowy)

```
Date now = new Date();
Thread.sleep(2000);
Date later = new Date();

System.out.println(now);
if (later.after(now)) System.out.println("Time flies");
System.out.println(later);
```

```
LocalDateTime now = LocalDateTime.now();
Thread.sleep(2000);
LocalDateTime later = LocalDateTime.now();

System.out.println(now);
if (later.isAfter(now)) System.out.println("Time flies");
System.out.println(later);
```

Kolekcje

- Kolekcja to obiekt grupujący wiele elementów
- Przykłady:
  - Numery telefonów klienta
  - Pracownicy danego departamentu
- Java Collections Framework: zunifikowana architektura do reprezentowania kolekcji i operowania na nich
  - Klasy i interfejsy w pakiecie **`java.util`**

Składniki Java Collection Framework

- **Interfejsy**
  - Abstrakcyjne typy danych reprezentujące kolekcje
  - Opisują sposób korzystania z kolekcji niezależny od implementacji
  - Np. **`List`**, **`Set`**
- **Implementacje**
  - Konkretne implementacje interfejsów (klasy użytkowe)
  - Np. **`ArrayList`**, **`HashSet`**
- **Algorytmy**
  - Metody realizacji przydatnych operacji na kolekcjach:
    - Wyszukiwanie, sortowanie
  - Algorytmy są polimorficzne, działają dla różnych implementacjach danego interfejsu kolekcji

Interfejsy kolekcji

Źródło:
docs.oracle.com

- **Collection** – ogólna koncepcja kolekcji elementów
  - Nie posiada standardowych implementacji
- **Set** – zbiór w sensie matematycznym (bez duplikatów)
- **List** – uporządkowana kolekcja, dopuszczająca duplikaty, możliwy dostęp do elementu o danym indeksie
- **Queue** – kolejka, typowo FIFO
- **Deque** – dwukierunkowa kolejka (FIFO / LIFO)
- **Map** – obiekt mapujący klucze (unikalne) na wartości
- **SortedMap**, **SortedSet** – posortowane wersje Map i Set

Interfejs Collection

```
public interface Collection<E> extends Iterable<E> { //
Basic Operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); // Optional
boolean remove(Object element); // Optional
Iterator iterator();
// Bulk Operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c); // Optional
boolean removeAll(Collection<?> c); // Optional
boolean retainAll(Collection<?> c); // Optional
void clear(); // Optional
// Array Operations
Object[] toArray();
<T> T[] toArray(T[] a); }
```

Dana implementacja kolekcji może nie obsługiwać metod opcjonalnych poprzez rzucanie wyjątku **`UnsupportedOperationException`**

Sposoby nawigacji po kolekcji

- Pętla for-each

```
for (Object o : kolekcja) System.out.println(o);
```

- Iterator
  - Jeśli chcemy mieć możliwość usuwania lub podmiany bieżącego elementu

Interfejs **`Iterator`**

```
public interface Iterator<E>
{
  boolean hasNext();
  E next();
  void remove(); // Optional
}
```

Korzystanie z iteratora

```
for (Iterator i = kolekcja.iterator(); i.hasNext(); )
  if (!cond(i.next())) i.remove();
```

Sposoby nawigacji po kolekcji (Java 8)

- Przetwarzanie strumieniowe
- Możliwość zrównoleglenia przetwarzania
  - Wspólna pula wątków dla wszystkich strumieni
  - Wykorzystanie wszystkich dostępnych procesorów
  - Niezalecane w aplikacjach Java EE
- Elementy programowania funkcyjnego
  - „Funkcja” do wykonania na elementach kolekcji przekazana typowo jako wyrażenie lambda

```
coll.stream()
        .forEach(o -> {System.out.println(o);});
```

```
coll.parallelStream()
        .forEach(o -> {System.out.println(o);});
```

Uwagi o elementach kolekcji JCF

- Kolekcje nie mogą przechowywać wartości typów prostych
  - można wykorzystać typy opakowujące
  - istnieją biblioteki od firm trzecich z kolekcjami dla typów prostych
- Wiele metod kolekcji wykorzystuje metodę equals() do porównania elementów
  - np. sprawdzanie czy dany element już znajduje się w kolekcji
- Implementacje algorytmów dla kolekcji mogą optymalizować operację porównania poprzedzając wywołanie equals() porównaniem skrótów zwracanych przez metodę hashCode()
  - dlatego obiekty równe w sensie equals() muszą mieć równe wartości skrótu zwracane przez hashCode()
- Elementy przechowywane w kolekcjach uporządkowanych (SortedSet, SortedMap) muszą być typu posiadającego naturalny porządek (interfejs Comparable) lub przy tworzeniu kolekcji musi być przekazany obiekt interfejsu Comparator

Interfejs Set

```
public interface Set<E> extends Collection<E> {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();
    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear();                              // Optional
    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Implementujące klasy biblioteczne: **`HashSet`**, **`TreeSet`**, **`LinkedHashSet`**

Interfejs List

```
public interface List<E> extends Collection<E> {
    // Positional Access
    E get(int index);
    E set(int index, E element);    // Optional
    boolean add(E element);         // Optional
    void add(int index, E element); // Optional
    E remove(int index);            // Optional
    abstract boolean addAll(int index,
        Collection<? extends E> c);  //Optional
    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    // Range-view
    List<E> subList(int from, int to);
}
```

Implementujące klasy biblioteczne: **`ArrayList`**, **`LinkedList`**, **`Vector`**

Interfejs Queue

```
public interface Queue<E> extends Collection<E> {
    boolean add(E o);
    boolean offer(E o);
    E element();
    E peek();
    E remove();
    E poll();
}
```

Implementujące klasy biblioteczne: **`PriorityQueue`**, **`BlockingQueue`**

Interfejs Map

```
public interface Map<K,V> {       // Basic Operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk Operations
    void putAll(Map<? extends K,? extends V> t);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Implementujące klasy biblioteczne: **`HashMap`**, **`TreeMap`**, **`LinkedHashMap`**

List – Przykład („raw type”)

```
String me  = "Alice";
String you = "Carol";
String he  = "Bob";

List people = new ArrayList();

people.add(me); people.add(you); people.add(he);

Collections.sort(people);

System.out.println((String) people.get(0));
System.out.println((String) people.get(1));
System.out.println((String) people.get(2));

for (Object p : people)
  System.out.println((String) p);

for (Iterator i = people.iterator(); i.hasNext(); )
  System.out.println((String)i.next());
```

```
Alice
Bob
Carol
```

```
Alice
Bob
Carol
```

```
Alice
Bob
Carol
```

Kolekcje generyczne (Generics) (od 1.5)

- Pozwalają na etapie kompilacji sprawdzić czy obiekt umieszczany w kolekcji jest odpowiedniej klasy (lub jej podklasy)
- Upraszczają kod, gdyż nie jest konieczne rzutowanie po pobraniu obiektu z kolekcji
- (!) Informacja o typie jest tracona na etapie kompilacji (type erasure)

```
String me="Alice"; String you="Carol"; String he="Bob";

List<String> people = new ArrayList<String>();

people.add(me); people.add(you); people.add(he);

Collections.sort(people);

System.out.println(people.get(0));

for (String p : people)
  System.out.println(p);

for (Iterator<String> i = people.iterator(); i.hasNext(); )
  System.out.println(i.next());
```

Optional<T>

- Kontener, który może zawierać wartość różną od null lub jej nie zawierać
- Gdy wartość jest dostępna, isPresent() zwraca true, a get() zwraca tę wartość
- Dla typów prostych dostępne są dedykowane warianty:
  - OptionalDouble
  - OptionalInt
  - OptionalLong
- Motywacje:
  - dotychczas brak możliwości wskazania, że metoda może zwrócić null
  - uciążliwe testowanie (referencja != null), aby uniknąć wyjątku NullPointerException

Tworzenie instancji Optional

- Utworzenie pustego obiektu Optional:

```
Optional<String> empty = Optional.empty();
```

- Utworzenie obiektu Optional z wartości różnej od null:

```
String str = "not null for sure";
Optional<String> opt = Optional.of(str);
```

  - rzuca NPE gdy parametr jest równy null

- Utworzenie obiektu Optional z wartości, która może być null:

```
String str = someOldMethod();
Optional<String> opt = Optional.ofNullable(str);
```

  - gdy przekazana wartość != null, tworzy obiekt Optional reprezentujący tę wartość
  - gdy przekazana wartość == null, tworzy pusty obiekt Optional

Praca z obiektami Optional

- Sprawdzenie czy wartość obecna i pobranie jej:

```
if (opt.isPresent()){str = opt.get(); }
```

  - get() rzuca NoSuchElementException gdy brak wartości
  - obecnie zamiast get() preferowane równoważne orElseThrow()
- Sprawdzenie czy obiekt Optional jest pusty:

```
if (opt.isEmpty()){...}
```

- Zwrócenie wartości domyślnej gdy pusty
  - zawartej wartości gdy niepusty

```
str = opt.orElse("default text");
```

- Wykonanie operacji na wartości, gdy dostępna:

```
opt.ifPresent(str -> System.out.println(str));
```

- Przetwarzanie w stylu strumieni: filter(), map(), flatMap()
- Konwersja do strumienia: stream()

Przykład motywujący (bez Optional)

- Typowy scenariusz:

```
Car car = ... // find a car using some API method
              // returning Car

String name = car.getOwner().getName();
```

  - NullPointerException jeśli car (lub owner) jest równy null

- Tradycyjne rozwiązanie:

```
String name = "Unknown";
if(car != null){
  Owner owner = car.getOwner();
  if(owner != null){
    name = owner.getName();
  }
}
```

Przykład motywujący (z Optional)

```
class Owner {
    private String name;
    ...
    public Optional<String> getName() {
        return Optional.ofNullable(name);
    }
}
```

```
class Car {
    private Owner owner;
    ...
    public Optional<Owner> getOwner() {
        return Optional.ofNullable(owner);
    }
}
```

```
Optional<Car> car = ... // find a car using some API method
                        // returning Optional<Car>

String name = car.flatMap(Car::getOwner)
                 .flatMap(Owner::getName)
                 .orElse("Unknown");
```

Kiedy używać typów opcjonalnych?

- PLANOWANE zastosowanie: typ zwrotny (czytelne API)
- NIEPOPRAWNE zastosowania:
  - pola (nieserializowalne)
  - parametry (komplikuje kod, mogą same być nullami)
  - puste collections (należy użyć pustych kolekcji…)
- Czy używać Optional w getterach w POJO? Raczej NIE
  - "Do not overuse."
  - problemy przy serializacji do formatu JSON
  - problemy z JPA (encje ORM)
  - problemy z beanami w aplikacjach internetowych (wyrażenia Expression Language na stronach)

Value-based classes

- Java ciągle (Java 13) nie ma prawdziwych value types
  - obiektów nie posiadających tożsamości, a jedynie wartość
  - jak np. struktury (typy struct) w C#
- Od Java 8 niektóre nowe klasy biblioteczne są definiowane jako "value-based classes"
  - Optional<T>, OptionalDouble, OptionalInt, OptionalLong
  - java.time classes (Duration, Instant, LocalDate, LocalDateTime, …)
- Charakterystyka:
  - final i niemodyfikowalne (ang. immutable)
  - equals(), hashCode() i toString() bazują tylko na stanie obiektu
  - nie powinno się na nich używać operacji polegających na tożsamości np. testowania równości referencji (==)
  - uważane za równe (i zastępowalne) na podstawie equals()
  - brak dostępnych (widzialnych) konstruktorów
  - instancje tworzone wzorcem fabryki (factory methods)

Pakiet java.io

- Pakiet `java.io` grupuje klasy służące do obsługi plikowego wejścia/wyjścia:
  - `java.io.FileInputStream`

    Służy do odczytywania plików binarnych
  - `java.io.InputStreamReader`

    Służy do odczytywania plików tekstowych; odpowiednio konwertuje bajty odczytywane przez `FileInputStream` na znaki zgodnie ze wskazanym kodowaniem znaków
  - `java.io.FileOutputStream`

    Służy do zapisywania plików binarnych
  - `java.io.OutputStreamWriter`

    Służy do zapisywania plików tekstowych; odpowiednio konwertuje zapisywane znaki do bajtów dla `FileOutputStream` zgodnie ze wskazanym kodowaniem znaków
- Przykładowe metody: `int read()` i `write(int)`

Odczyt pliku - przykłady

binarnego

```
int value;
FileInputStream fStream;

fStream = new FileInputStream("/home/data.txt");
while ((value = fStream.read()) != -1)
  System.out.write((char)value);
fStream.close();
```

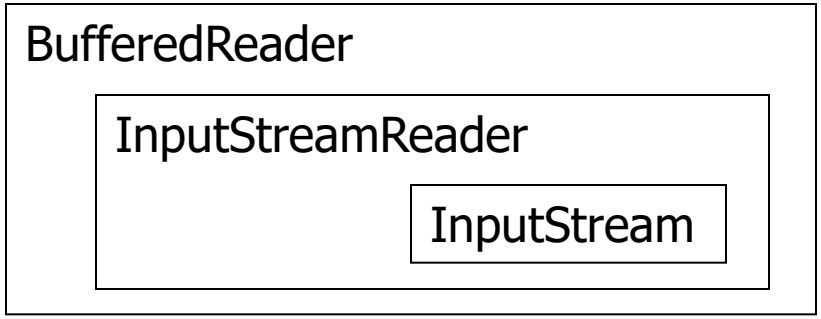
tekstowego

```
int value;
FileInputStream fStream;
InputStreamReader fReader;

fStream = new FileInputStream("/home/data.txt");
fReader = new InputStreamReader(fStream,"ISO-8859-2");
while ((value = fReader.read()) != -1)
  System.out.print((char)value);
fReader.close();
fStream.close();
```

Odczyt ze standardowego wejścia

- Standardowe wejście: obiekt `System.in` klasy `InputStream`
  - strumień bajtowy
  - wymaga „opakowania” dla odczytu danych znakowych




```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader reader = new BufferedReader(isr);
String line = null;
while ((line = reader.readLine()) != null)
{
  // process line
}
```

Dokumentacja

Javadoc

- Narzędzie do generowania dokumentacji kodu Java w formacie HTML w oparciu o specjalne komentarze poprzedzające klasy, metody i pola
  - część dystrybucji Java SE (JDK, Java SDK)
- Domyślnie uwzględnia tylko składowe public i protected
- Powszechny format dokumentacji API bibliotek języka Java
- Przykład uruchomienia (z domyślnymi opcjami):

```
javadoc package_name
```

Javadoc – przykład komentarzy

```
package shapes;
/**
 * Class to represent squares
 * @author Marek
 */
public class Square {
    /** Side length */
    private double side;
    /** Calculates area */
    public double area() { return side * side;     }
    /**
     * Scales the square
     * @param ratio Scale ratio
     */
    public void scale(double ratio) {
        side = side * ratio;
    }
}
```

Javadoc – przykład dokumentacji

PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package shapes

Class Square

java.lang.Object
shapes.Square

Constructor Summary

Constructors

Constructor	Description
`Square()`	

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`double`	`area()`	Calculates area
`void`	`scale(double ratio)`	Scales the square

Method Detail

area

```
public double area()
```

Calculates area

scale

```
public void scale(double ratio)
```

Scales the square

Parameters:

`ratio` - Scale ratio