Andrzej Pająk

Andrzej.Pajak@pw.edu.pl

Podstawy Programowania w języku Java

Instytut Informatyki
Studia Podyplomowe
Java EE — produkcja oprogramowania

Edycja JA22Z - (grupa JA1-A)
Warszawa 2022

Plan

- Cel przedmiotu
- Rodowód języka Java
- Ogólna charakterystyka Javy
- Przykład "Hello …"
- Konwersacje tekstowe
- Struktura programu źródłowego
- Struktura leksykalna
- System typów
- Typy pierwotne
- Opakowania typów pierwotnych
- Instrukcje
- Typy referencyjne
- Tablice; przykład
- Definiowanie klas i metod
- Definiowanie klas generycznych
- Projektowanie klas; klasa Ulamek
- Praca ze strumieniami out i err
- Uwagi o stylu kodowania
- Dziedziczenie i polimorfizm
- Klasa Object
- Dziedziczenie vs zawieranie
- Referencje w hierarchii dziedziczenia
- Składowe statyczne i instancyjne
- Funkcje wirtualne i polimorfizm
- Klasy i metody abstrakcyjne. Przykład
- Hierarchia klas abstrakcyjnych
- Enumeracje
- Interfejsy
- Interfejsy i klasy abstrakcyjne
- Wyjątki
- Schemat składniowy obsługi wyjątków
- O kontenerach generycznych
- Typy wieloznaczne (wildcard)
- JCF – Java Collections Framework

Cel przedmiotu

- Przedstawić podstawy programowania w języku Java w stopniu umożliwiającym pisanie prostych programów konwersacyjnych.
  - **Struktura programu źródłowego**, rola klas i obiektów, scenariusz pracy nad programem, proste eksperymenty z językiem i środowiskiem.
  - **System typów**, deklaracje / definicje obiektów i zmiennych; tablice.
  - Współpraca programu z **WE/WY**.
  - Przetwarzanie danych; **repertuar operatorów** i ich interpretacja.
  - **Repertuar instrukcji**; semantyka, przykłady użycia.
  - Przetwarzanie tekstu, klasa String.
  - **Definiowanie klas**, mechanizm dziedziczenia. Klasy abstrakcyjne.
  - **Interfejsy** - specyfikowanie i implementacja
  - Usługi biblioteki JCF (**Java Collections Framework**).
- **Będzie końcowy (friendly) sprawdzian / test**

Do studiowania / ćwiczenia

Literatura (mini wybór, Helion)

- C. Horstmann: *Java. Podstawy*. Wyd.X, 2016. 872s.
- C. Horstmann: *Java 8. Przewodnik doświadczonego programisty, 2018, 462s*
- C. Horstmann: *Java. Techniki zaawansowane. Wyd. XI, Helion, 2020, 808s*
- B.J. Evans, D. Flanagan: *Java w pigułce.* Wyd.6, Helion 2015, 351s.
- i 100+ innych

Internet (mikro wybór)

8, 11, ...

- Oficjalna strona z dokumentacją: **https://docs.oracle.com/en/java/javase/**
- Specyfikacja języka i VM: **https://docs.oracle.com/javase/specs/**
- Specyfikacja API: **http://docs.oracle.com/javase/8/docs/api/**

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

- Java Code Geeks: **http://www.javacodegeeks.com/**

Środowiska zintegrowane (bezpłatne - wybór)

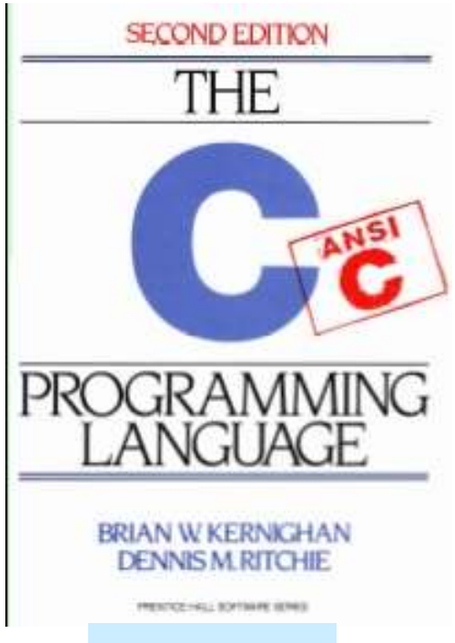
- Eclipse: **https://eclipse.org/** **NASZ WYBÓR**
- NetBeans: **https://netbeans.org/** OpenJDK: **https://adoptopenjdk.net/**
- IntelliJ IDEA: **https://www.jetbrains.com/idea/**
- DrJava ("lekkie" środowisko do ćwiczeń): **http://www.drjava.org/**

Rodowód języka Java

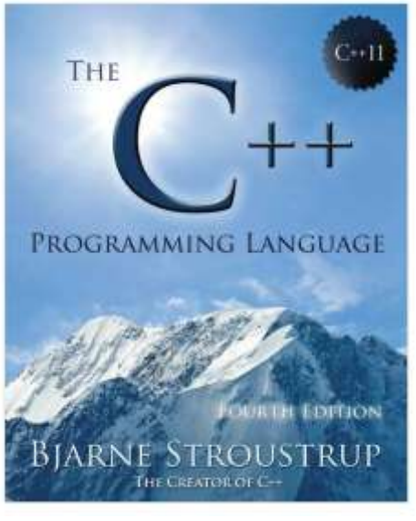
Dennis Ritchie
Język C ~1971

Bjarne Stroustrup
Język C++ ~1983

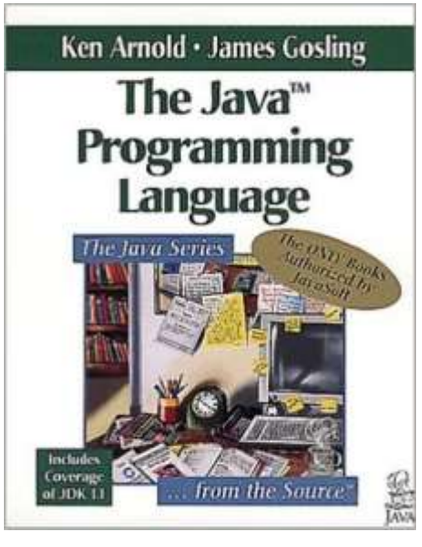
James Gosling
Język Java ~1995



~280 str



>1300 str



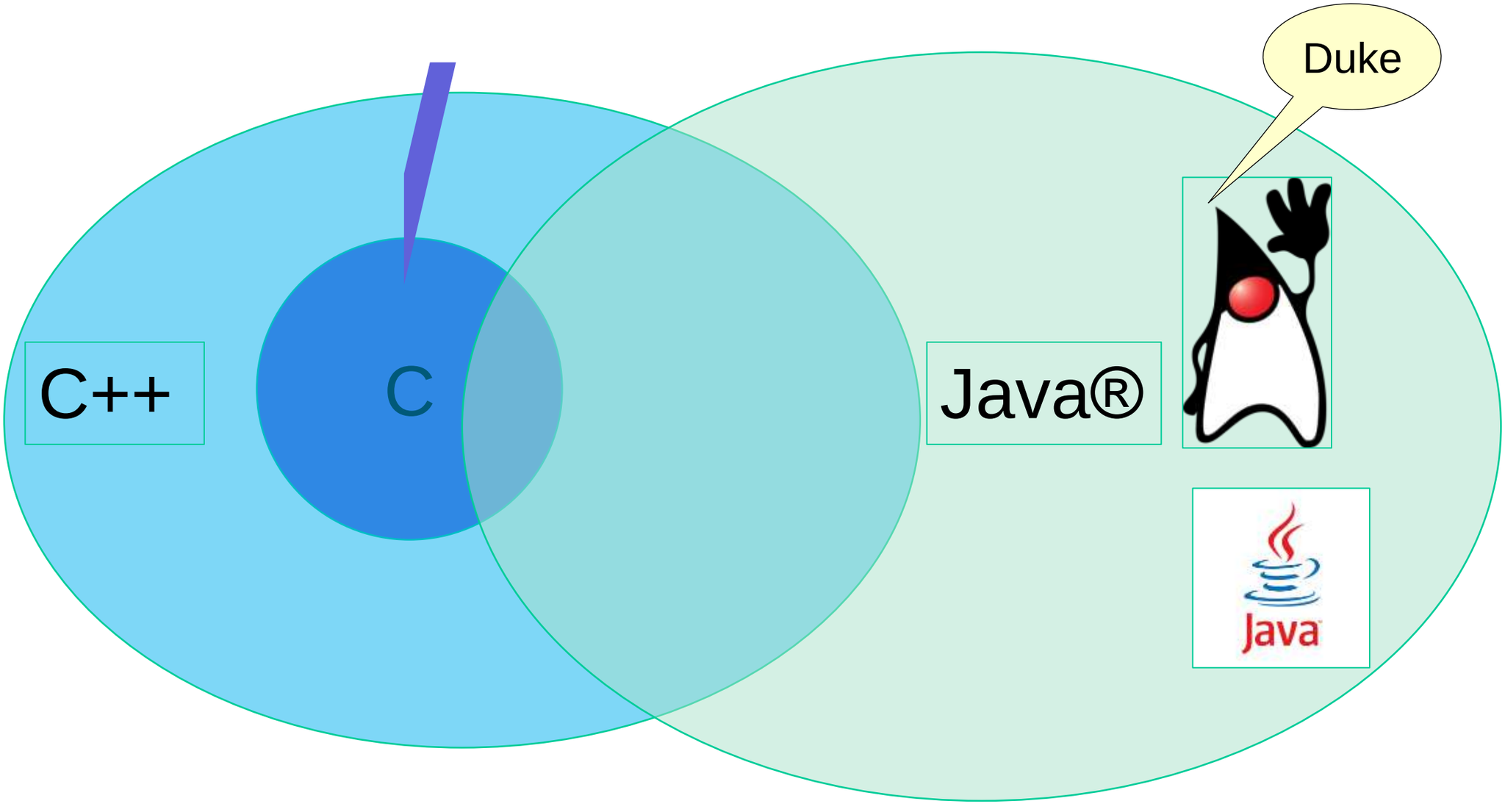
>900 str

Rodowód języka Java (cd1)

Trzy cytaty:

1. "**C** is a general-purpose programming language with features: economy of expression, modern flow control and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application." [Kernighan, Ritchie]
2. "**C++** is a general-purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types." [Stroustrup]
3. "**The Java®** programming language is a general-purpose, concurrent, class based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, …" [Oracle, jls8]

Rodowód języka Java (cd2)



Relacja ogólna między językami

Java - historia popularności języków

Na podstawie TIOBE: www.tiobe.com/tiobe-index/

(w tabeli są języki, które były lub są w pierwszej dziesiątce)

Język	2020	2015	2010	2005	2000	1995	1990	1985
C	1	2	2	1	1	2	1	1
Java	2	1	1	2	3	29	-	-
Python	3	6	6	6	21	15	-	-
C++	4	3	3	3	2	1	2	9
C#	5	4	5	7	9	-	-	-
JavaScript	6	8	8	10	7	-	-	-
PHP	7	7	4	5	19	-	-	-
SQL	8	-	-	-	-	-	-	-
Swift	9	16	-	-	-	-	-	-
R	10	13	49	-	-	-	-	-
Lisp	29	25	15	13	8	5	6	2
Fortran	31	24	24	15	15	17	3	5
Ada	33	27	22	17	17	4	7	3
Pascal	242	15	14	16	16	3	10	6

Ogólna charakterystyka Javy

- "Zaklęcia" wysokiej rangi (por. wybory 5-przymiotnikowe)
  - język wysokiego poziomu, ogólnego przeznaczenia
  - język obiektowy (mechanizmy hermetyzacji obiektów, klasy, dziedziczenie, polimorfizm, przeciążanie funkcji, …)
  - wsparcie dla programowania generycznego; modularność (od Java 9)
  - wsparcie dla programowania współbieżnego (wielowątkowego, rozproszonego) i programowania funkcyjnego (od Java 8)
  - silny, statyczny system typów; automatyzm zarządzania pamięcią
  - przenośność poprzez standaryzację środowiska wykonawczego (maszyna wirtualna)
  - zgodność wstecz z wcześniejszymi wersjami platformy
- "Zaklęcia" niższej rangi
  - język "klamroluby" – składnia wg konwencji wziętych z C
  - nazwa języka jest znakiem towarowym (**Java®**)
  - standaryzacja poza ISO - zarządzana przez korporację (obecnie Oracle) z wykorzystaniem procedury JCP (**Java Community Process**)

Ewolucja Javy

- **JDK 1.0 (1996)** 8 pakietów, 212 klas
- **JDK 1.1 (1997)** Klasy lokalne, API Reflection; rozmiar >2x ↑
- **J2SE 1.2 (1998)** **Java 2; API Collection; rozmiar >3x** ↑↑
- **J2SE 1.3 (2000)** Maszyna wirtualna HotSpot; konserwacja
- **J2SE 1.4 (2002)** **I/O niskiego poziomu; regex, XML, SSL** ↑↑

Sun

- **J2SE 5.0 (2004)** **Język: typy rodzajowe, enums, varargs adnotacje; 166 pakietów, 3562 klasy** ↑↑↑
- **Java SE 6 (2006)** API Compiler; usprawnienia, **konserwacja**
- **Java SE 7 (2011)** (pierwsza aktualizacja w Oracle) NIO2 API ↑↑
- **Java SE 8 (2014)** **wyrażenia lambda, zmiany w kolekcjach, JavaScript wykonywany w JVM (217 pakietów, 4240 klas i interfejsów)** ↑↑↑
- **Java SE 9 (9.17)** **System modułów JPMS (Jigsaw), Java Shell, Linker, ... (2 aktualizacje)** ↑↑↑
- **Java SE 10 (03.18)** **12 uzupełnień / rozszerzeń (2 aktualizacje)** ↑
- **Java SE 11 (9.18)** **nowy GC, Unicode 11; JavaFX, Java EE,** ↑↑
- **Java SE 17 (9.21)** **enhancements, bug-fixes, ...** ↑↑

Oracle

Ewolucja Javy (cd) - klasy, pakiety, moduły

Wersja	Moduły	Pakiety	Klasy
Java SE 6		203	3793
Java SE 7		209	4024
Java SE 8 (LTS)		217	4240
Java SE 9	78 (SE35+JDK43)	315	6005
Java SE 10	77 (SE35+JDK42)	314	6002
Java SE 11 (LTS)	59 (SE22+JDK37)	223	4410
Java SE 13	59 (SE22+JDK37)	223	4403
Java SE 14	61 (SE22+JDK39)	225	4420
Java SE 17 (LTS)	60 (SE22+JDK38)	223	4421

Porównanie z "przodkami": C/C++

Cecha / wsparcie	Java	C/C++
• Paradygmat programowania proceduralnego	-+	+
• Paradygmat programowania obiektowego	+	+
• Paradygmat programowania rodzajowego	+	+
• Przenośność na poziomie języka pośredniego (bajtkodu)	+	-
• Wskazania i arytmetyka adresowa	-	+
• Automatyczne zarządzanie pamięcią	+	-
• Preprocesor (makrogenerator)	-	+
• Dziedziczenie wielobazowe	-	+
• Funkcje składowe (metody) domyślnie wirtualne	+	-
• Parametry przekazywane przez: wartość | referencję	+ | -	+ | +
• Silny mechanizm szablonów / generyków	+-	++
• Przeciążanie metod (i operatorów)	+(-)	+(+)
• Obsługa sytuacji wyjątkowych	+	+
• Efektywność kodu	+	++

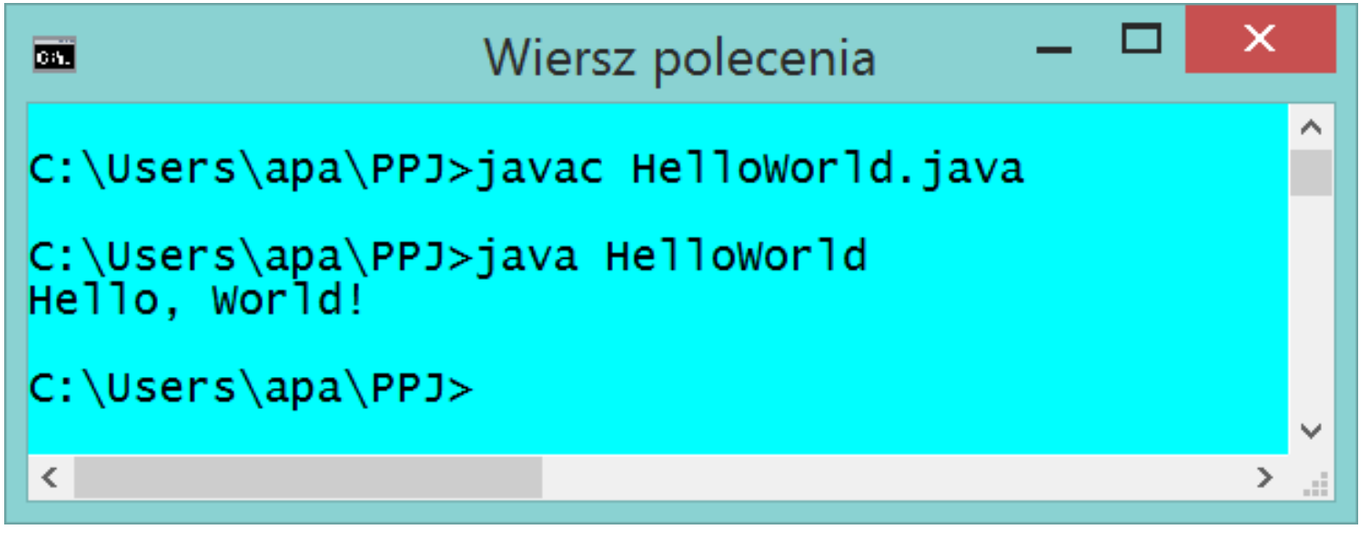
Przykład 0 , Hello, World! – z linii poleceń

```
// Program 0, plik HelloWorld.java
// import java.lang.*;

public class HelloWorld {

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

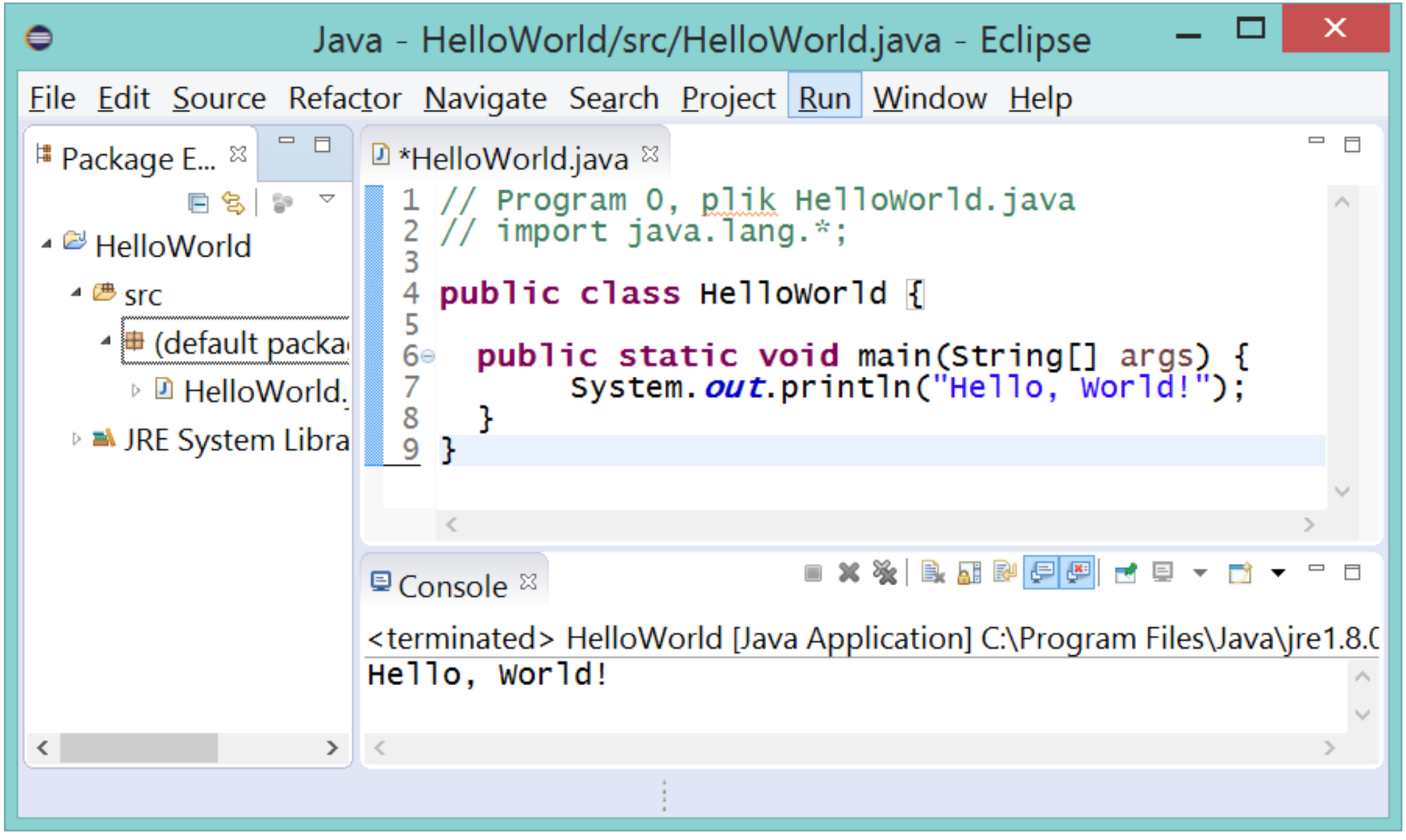
Pakiet dołączany implicite

Obowiązkowo w każdym programie

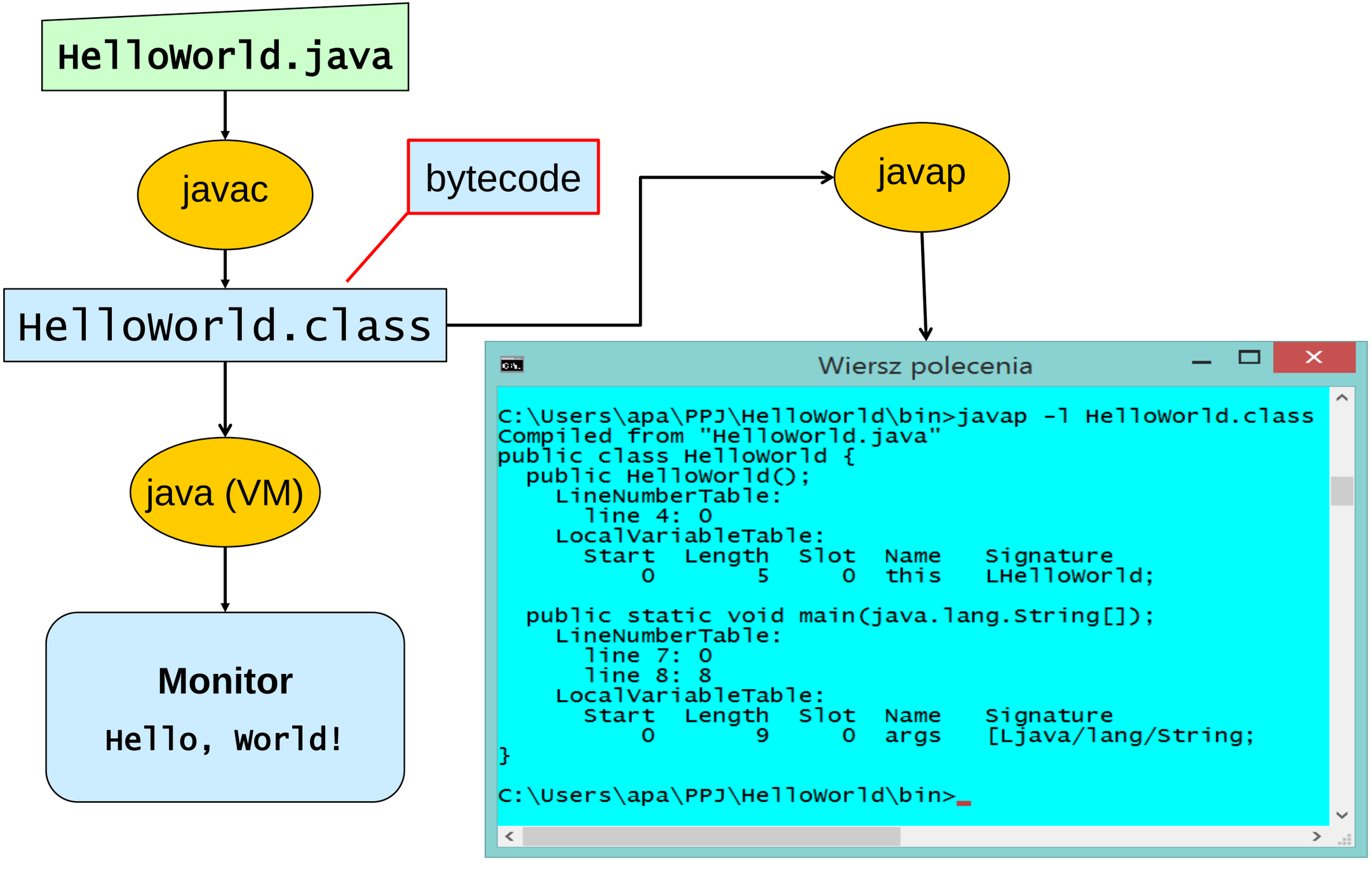
Znakowy strumień wyjściowy (ekran lub plik)



Przykład 0 , Hello, World! – w Eclipse



Przykład 0 (cd1) – kompilacja i wykonanie



Proste konwersacje tekstowe [prompt ==> input]

- **Program konwersacyjny (PK)** to program realizujący scenariusz (najczęściej powtarzalny aż do wymuszonego zakończenia):
  - program zgłasza zachętę do wprowadzenia pewnych danych (zlecenia)
  - użytkownik odpowiada
  - program sprawdza, czy odpowiedź implikuje zakończenie przetwarzania; jeśli nie, to przetwarza zlecenie i generuje wyniki
- Podobny scenariusz realizują interpretery linii poleceń **(CLI)**; jednak zamiast konkretnych zachęt pokazują tylko sygnał gotowości przyjęcia następnego polecenia z repertuaru powłoki
- W **PK** jak wyżej, **inicjatywa jest po stronie programu**; użytkownik jest tylko dostarczycielem danych

Uwaga: Nie należy prostych **PK** mylić z wyrafinowanymi systemami konwersacji usiłującymi symulować kompetentną rozmowę z użytkownikiem (systemy typu chatbot). [por. test Turinga]

Proste konwersacje (cd1)

- **Asymetria** pomiędzy programową obsługą wyjścia i wejścia
- **Wyjście** jest generowane pod kontrolą programu ==> poprawny program nie generuje "śmieci"
- **Wejście** pochodzi od użytkownika i może być (celowo lub niechcący) błędne, złośliwe, bezsensowne, ...
- Ergo: program musi kontrolować **formę i treść** danych WE
- Dodatkowa trudność: obsługa Unikodu

Formatowane wyjście znakowe:

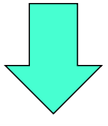
- `System.out.print(`*`string`*`);`
- `System.out.println(`*`string`*`);`
- `System.out.printf(`*`formatstring`*`, `*`args`*`);`
- `System.out.format(`*`formatstring`*`, `*`args`*`);`

`formatstring` steruje (jak w C) oczekiwaniami względem *args*

Proste konwersacje (cd2)

Przykład

```
String t = "Przyśpieszenie ziemskie";
double g = 9.81;
System.out.println(t + " g=" + g + " [m/s^2]");
System.out.printf("%s g=%.3f %s\n", t, g, "[m/s^2]");
System.out.format("%s g=%+8.2f %s\n", t, g, "[m/s^2]");
```



```
Przyśpieszenie ziemskie g=9.81 [m/s^2]
Przyśpieszenie ziemskie g=9,810 [m/s^2]
Przyśpieszenie ziemskie g=   +9,81 [m/s^2]
```

Metody `printf` i `format` są wrażliwe na ustawienia regionalne (`Locale`); w przykładzie obowiązuje lokalizacja polska:

```
System.out.println(Locale.getDefault()); // pl_PL
```

PPJ00_InputOutputStrings

Proste konwersacje (cd3)

Budowa łańcucha formatującego *formatstring*

- Zanurzony w zwykłym tekście ciąg **deskryptorów konwersji**
- Każdy **deskryptor** konwersji jest kojarzony z odpowiednim **argumentem**
- Typ argumentu musi być **zgodny** z deskryptorem konwersji
- Efektem formatowania jest tekst wynikowy, w którym w miejsce deskryptorów wstawiono konwersje znakowe argumentów

Postać deskryptora (w nawiasach [ ] elementy opcjonalne)

%[argnum$][znaczniki][pole][.precyzja]TypKonw

```
argnum     Numer argumentu: 1$, 2$, ...
znaczniki  '-' wyrównanie lewostronne; '+' pokaż znak
           '(' liczby ujemne w nawiasach, ...
pole       minimalna liczba znaków w konwersji
precyzja   tyle cyfr po kropce (przecinku)
TypKonw    jedna litera z {aA bB cC d eE f gG hH o sS xX}
```

Proste konwersacje (cd4)

Przykład

```
// Numery argumentów:                          1 2  3
System.out.format("%3$s g=%2$-8.3f %1$s\n",t,g,"[m/s^2]");
System.out.printf("%3$S g=%2$.3f %1$S\n", t, g, "[m/s^2]");

[m/s^2] g=9,810    Przyśpieszenie ziemskie
[M/S^2] g=9,810 PRZYŚPIESZENIE ZIEMSKIE
```

Wejście znakowe (najprostsze rozwiązanie)

- Skojarzyć ze standardowym strumieniem WE `System.in` (albo z plikiem tekstowym) obiekt do analizy ciągów znaków **(skaner)**:
  `Scanner scn = new Scanner(System.in); // java.util`
- Po wysłaniu zachęty można czytać ze skanera dane różnych typów

```
String s = scn.next();// czytaj do znaku "białego"
double d = scn.nextDouble();
int i = scn.nextInt(); // itp. ...; p. dokumentacja
```

Wybrane metody klasy Scanner

Typ	Metoda
void	close()
Pattern	delimiter()
boolean	hasNext() hasNextBoolean() hasNextByte(); hasNextByte(int radix) hasNextShort(); hasNextShort(int radix) hasNextInt(); hasNextInt(int radix) hasNextLong(); hasNextLong(int radix) hasNextFloat(); hasNextDouble() hasNextLine()
String typ	next(); nextLine() next*Typ*() [*Typ* = Boolean|Byte|Short|Int| . . .

Proste konwersacje (cd6)

Wybrane metody klasy String [67 wszystkich]
`char charAt(int index)`
`int compareTo(String other)`
`int compareToIgnoreCase(String str)`
`void getChars(int srcBg, int srcEd, char[] dst, int dstBg)`
`int indexOf(int ch) // pierwsze wystąpienie ch`
`int indexOf(int ch, int fromIndex)`
`int indexOf(String str)`
`int lastIndexOf(int ch)`
`int lastIndexOf(String str)`
`int length()`
`String replace(char oldChar, char newChar)`
`String substring(int beginIndex)`
`String substring(int beginIndex, int endIndex)`
`String toLowerCase(); toUpperCase()`
`static String valueOf(xxxx x) [xxxx=int|long|double, ...]`
`static String format(String format, Object... args)`

PPJ00_InputOutput
Strings

Program z lotu ptaka

- Program w języku Java składa się z jednego lub większej liczby plików źródłowych (`*.java`) – **jednostek kompilacji (JK)**
- Każdy plik źródłowy rozpoczyna się od:
  - (**opcjonalnie**) **deklaracji pakietu,** do którego przynależy
  - ciągu **deklaracji import** wg potrzeb (domyślnie: **import java.lang;**)

  Łącznie deklaracje te określają przestrzenie nazw pobieranych (import) oraz przestrzeń, do której wejdą definicje. **Wszystkie nazwy wykorzystywane przez JK muszą być jednoznaczne**
- Zasadnicza część JK to **definicja(e) typów** referencyjnych, najczęściej klasy lub interfejsu; wewnątrz klasy definiowane są parametryzowane metody, konstruktory, zmienne, stałe…
- **Metody** zawierają ciągi deklaracji lokalnych, wyrażeń, instrukcji
- Wszystkie **nazwy** zdefiniowane w JK mają określoną widzialność (zasięg) i określony stopień ochrony dostępu (**domyślny w pakiecie, public, protected, private**).

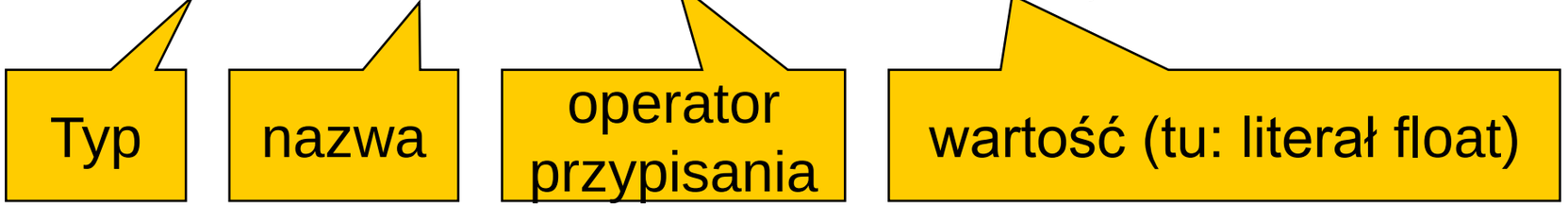
Struktura programu źródłowego w Javie

- Program składa **z jednej lub więcej klas** zorganizowanych w **pakiety** (nie ma samodzielnych funkcji, tylko metody w klasach)
- Warstwa informacyjna (**dane**) i warstwa "czynnościowa" (**operacje, metody**) są zintegrowane na poziomie klas i obiektów
- Dane są reprezentowane przez stałe i zmienne (obiekty)
- Stałe są desygnowane przez napisy zwane literałami albo "OBIEKTY ZAMROŻONE" (**final**)
- Zmienna jest desygnowana przez wyrażenie, w najprostszym przypadku przez zwykłą nazwę
- **Operacje** odnoszą się do:
  - otoczenia (WE/WY)
  - manipulacji arytmetycznych, numerycznych, logicznych, …
  - zorganizowanych agregacji czynności (sekwencje, powtórzenia, wybory)
  - abstrakcji nowych operacji (**definicje metod w klasach**)

Struktura programu źródłowego w Javie (cd1)

- Zmienna jest pojemnikiem przystosowanym do przechowywania danych **określonego typu**;
- Zmienne (**obiekty**) mają odpowiednie umiejscowienie w pamięci i podlegają cyklowi: tworzenie, używanie, likwidacja
- Podstawę **systemu typów** Javy stanowią wbudowane typy liczbowe, typ znakowy, typ logiczny i typy referencyjne
- Zmienne muszą być deklarowane przed pierwszym użyciem, np. `float cena;`
- Deklaracja nie oznacza nadania wartości; niektóre konteksty powodują jednak nadanie zmiennym **wartości domyślnych**
- W deklaracji można jawne zainicjować wartość zmiennej, np.:

```
float cena = 13.99f;
```



Struktura leksykalna

- **Program źródłowy** w Javie wykorzystuje unikod (Unicode 10.0)
- Tekst programu analizowany jest w 3 logicznych fazach:
  1. Zastępowanie znaczników unikodu (**\uxxxx**) odpowiednimi znakami (można zatem napisać program posługując się tylko kodem ASCII)
  2. Filtracja komentarzy i rozbicie strumienia znaków z fazy 1 na **atomy leksykalne**
  3. Analiza składniowa strumienia atomów leksykalnych wg formalnych reguł **gramatyki bezkontekstowej**.
- **Atom leksykalny** to niepodzielna jednostka tekstu posiadająca interpretację; ciąg znaków reprezentujący atom to **leksem**
- Jest 5 rodzajów atomów leksykalnych:
  - słowa kluczowe
  - identyfikatory
  - literały
  - operatory
  - ograniczniki

Struktura leksykalna - komentarze, słowa kluczowe

Komentarze 3 postaci:

- `// Komentarz do końca wiersza`
- `/* komentarz wielowierszowy do zamknięcia parą */`
- `/** Komentarz dokumentujący dla javadoc */`

Słowa kluczowe (zarezerwowane)

abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp (--)	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	_ (podkr.)

Typy pierwotne | Literały | Operatory | Modyfikatory | Nieużywane

Słowa kontekstowe: exports module non-sealed open opens permits provides requires sealed record to transitive uses var with yield

Struktura leksykalna - identyfikatory

Identyfikatory

- Duże i małe litery są różne
- Dopuszczalne znaki: litery, cyfry, podkreślenia '_', znak waluty (**'$', £, ...**); symbole waluty są przewidziane do wyróżniania kodu generowanego przez narzędzia - lepiej ich nie używać w własnym kodzie
- Pierwszy znak nie może być cyfrą
- Pojedyncze podkreślenie '_' nie może być identyfikatorem
- Nie ma ograniczenia na długość identyfikatora
- Wyrażenie regularne (uproszczone): `[a-zA-Z_$][a-zA-Z\d_$]*`
- Poprawność identyfikatora może być sprawdzona formalnie przez metody logiczne klasy `java.lang.Character`:
  `isJavaIdentifierStart()` oraz
  `isJavaIdentifierPart()`

mogą być litery narodowe

Przykłady

- `zmienna STATUS toJestNazwaGarbata MojaKlasa isBlue ...`

Struktura leksykalna - literały

Literały

- `Object ob = null; // Uniwersalny literał referencyjny`
- `boolean b1= false, b2=true;`
- `char c='C', tab='\t', cr='\u00A9'; // \u00A9 == ©`
- `int a=1, b=255,  c=0377,  d=0xff, e=0b111_111;`
  `//  dziesiętne  oktalna  hex  binarnie`
- `long a=1L, b=255L, c=0377L, d=0xffL, e=0b111_111L;`
- `long l=1l;  // Lepiej nie używać malej litery`
- `byte x = (byte)44;  // byte i short z rzutowaniem`
- `float  fec = 1.602_176_621E-19F;// ładunek elektronu`
- `double dec = 1.602_176_621E-19; // ładunek elektronu`
- `double dec = 1.602_176_621E-19D;// to samo co powyżej`
- `float  x = 2F;  // wartość 2.0F`
- `float  y = 0xFF.0p12F; // Zapis heks.: 255.0*2^12`
- `double z = 0xFF.0p12;  // jak wyżej, wartość double`
- `String name = "Scarlett O\'Hara";// niemodyfikowalny`

Przykład

```
// Program 1: nazwa zmiennej;
// znaki spec. wtrącone w tekst.

public class Wierszyk {

  public static void main(String[] args) {
  String treść =
    "\nKiedy Puchacz i Kicia wyruszyli w rejs życia" +
    "\nW zgrabnej łódce groszkowozielonej" +
    "\nWzięli z sobą parówki, duży zapas gotówki" +
    "\nI słój miodu z nalepką \"The Honey\"." +
    "\n  Edward Lear (tł. S.Barańczak) (\u00B1 \u00AE)";
                                        //   ±     ®
  System.out.println(treść);
  }
}
```

Znaki specjalne

```
\n nowy wiersz
\t tabulacja
\b cofacz
\r powrót karetki
\f nowa strona
\\ lewy ukośnik \
\' apostrof '
\" cudzysłów "
\ooo   znak oktalnie
\uxxxx unikod
```

==> PPJ01_Wierszyk

Przykład (cd1)

```
// Można uprościć zapis tekstów korzystając z nowego
// mechanizmu: bloku tekstowego """ ..... """
// znaki spec. wtrącone w tekst.
public class Wierszyk {
  public static void main(String[] args) {
    String textBlock =
    """

    Kiedy Puchacz i Kicia wyruszyli w rejs życia
    W zgrabnej łódce groszkowozielonej
    Wzięli z sobą parówki, duży zapas gotówki
    I słój miodu z nalepką "The Honey".
        Edward Lear (tł. S.Barańczak) (\u00B1 \u00AE)
    """;                                  //   ±     ®
    System.out.println(treść);
  }
}
```

==> PPJ01_Wierszyk

Struktura leksykalna - operatory

Operatory (38 atomów w tej roli; niektóre mają parę znaczeń)

Pr	Ł	Operator	Argumenty	Znaczenie

16	→	`.` (separator)
		`(args)`
		`[ ]`
		`++ --`
			Obiekt.składowa
			metoda(args)
			tablica, indeks
			zmienna
				Wybór składowej w obiekcie
				wywołanie metody z argumentami
				dostęp do elementu wg indeksu
				post- inkrementacja i dekrementacja

15	←	`++ --`
		`+ -`
		`~`
		`!`
			zmienna
			liczba
			integer
			`boolean`
				pre- inkrementacja i dekrementacja
				unarny plus; unarny minus
				inwersja bitów
				negacja logiczna (NOT)

14	←	`(`*`Typ`*`)` `new`	typ klasa(args)	operator konwersji (do typu *Typ*) utworzenie nowego obiektu klasy
13	→	`* / %`	liczba, liczba	mnożenie, dzielenie, reszta
12	→	`+ -` `+`	liczba, liczba `String, String`	dodawanie, odejmowanie konkatenacja

==> PPJ02_OperatoryBinarne

Operatory (cd1)

Pr	Ł	Operator	Argumenty	Znaczenie

11	→	`<<`
		`>>`
		`>>>`
			integer, integer	przesunięcie bitowe w lewo
				przesunięcie bitowe w prawo
				jak wyżej - wsuwane zera,

10	→	`<  <=  >  >=` `instanceof` *Typ*	liczba, liczba referencja, typ	relacje porządku przynależność do typu *Typ*
9	→	`==  !=`	pierwotny, pierwotny referencja, referencja	porównanie wartości czy refs do tego samego obiektu
8	→	&	integer, integer boolean, boolean	bitowa operacja AND logiczna operacja AND
7	→	^	integer, integer boolean, boolean	bitowa operacja XOR logiczna operacja XOR
6	→	|	integer, integer boolean, boolean	bitowa operacja OR logiczna operacja OR

Struktura leksykalna - operatory (cd2); ograniczniki

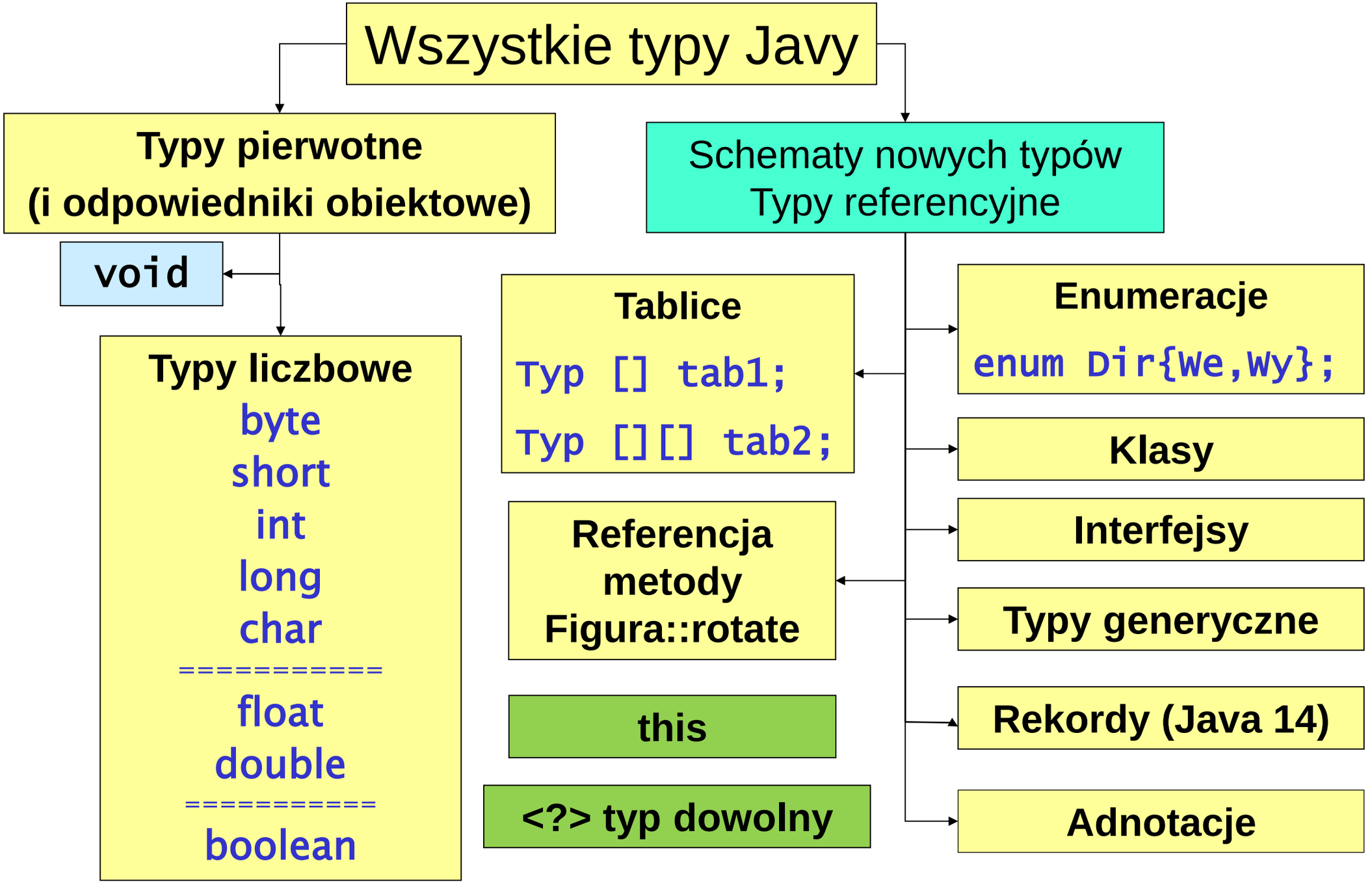
Operatory (cd2)

Pr	Ł	Operator	Argumenty	Znaczenie
5	→	&&	boolean, boolean	warunkowa operacja AND
4	→	||	boolean, boolean	warunkowa operacja OR
3	←	? :	boolean, dwa dowolne równych typów	operator warunkowy 3 argumentowy
2	←	= *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	zmienna, wyrażenie	przypisanie "zwykłe" przypisania złożone v #= exp; ⇒ v = v # exp;
1	←	->	(args) -> treść metody case stała -> akcja	wyrażenia lambda: (*params*) -> { *instrukcje* } switch(x) { case A-> a(); case B-> b(); };

Ograniczniki/separatory (12): `( ) { } [ ] ; , . ... @ ::`

```
public static void main(String[] args) {  // 11 atomów
```

System typów Javy



Typy pierwotne

Typ	L. bitów	Rodzaj danych	Wartość domyślna	Zakres
boolean	1b	logiczne (false, true)	false	---
byte	8b	Liczba całkowita	0	-128 .. 127
char	16b	Znak Unicode	\u0000	\u0000 .. \uffff (0 .. 2^{16}-1)
short	16b	Liczba całkowita	0	-32768 .. +32767 (-2^{15} .. 2^{15} – 1)
int	32b	Liczba całkowita	0	-2147483648 .. +2147483647
long	64b	Liczba całkowita	0	-9223372036854775808 +9223372036854775807
float	32b	Zm.przecinek (IEEE 754)	0.0F	1.4E-45 .. 3.4028235E+38
double	64b	Zm. przecinek (IEEE 754)	0.0	4.9E-324 .. 1.7976931348623157E+308

Użycie typów pierwotnych

- Typ **boolean** ściśle odpowiada roli wartości logicznych true, false
  - nie można żadnego typu pierwotnego ani żadnego typu referencyjnego przekształcić do typu boolean
  - w kontekstach gdzie spodziewana jest wartość logiczna (np. w operatorze `war? exp1: exp2`) war musi generować wartość logiczną
  - nie ma znanego z innych języków (np. C/C++, Python) traktowania wartości liczbowych różnych od 0 jako wartości logicznej true.
- Typ **char** jest jedynym pierwotnym typem liczbowym z zakresem nieujemnym (do reprezentacji numerów kodowych unikodu)
- Między typami liczbowymi (w tym char) są dopuszczalne następujące konwersje:
  - konwersja **poszerzająca**, np.: `double x = 12345;// int⇒double`
  - konwersja **zwężająca** wymaga na ogół jawnego wymuszenia:

```
int i = 44; byte b = (byte)i;
short s = 999; // OK, literał 999 w zakresie short
```

Użycie typów pierwotnych (cd1)

- Dla wszystkich typów pierwotnych interpreter Javy gwarantuje konwersję znakową; można pisać:

```
System.out.println("licznik = " + licznik);
System.out.println("S = " + 3.1415*r*r);
System.out.println("a>b: " + (a>b)); // np. true
```

- Dla typów referencyjnych dostępność metody `toString()` daje podobny efekt (klasa `Object` ma tę metodę – można ją podmieniać w klasach potomnych):

```
Object o = new Object();
System.out.println(""+o.toString());
                        // java.lang.Object@2a139a55
```

- Parametry przekazywane są do metod zawsze **przez wartość (dotyczy to typów pierwotnych i referencyjnych)**
- Nie można napisać metody odpowiadającej w C/C++ np. funkcji swap(int& a, int& b); podobnie: swap(Object x, Object y);

==> PPJ02_SwapNieDziala

Użycie typów pierwotnych (cd2)

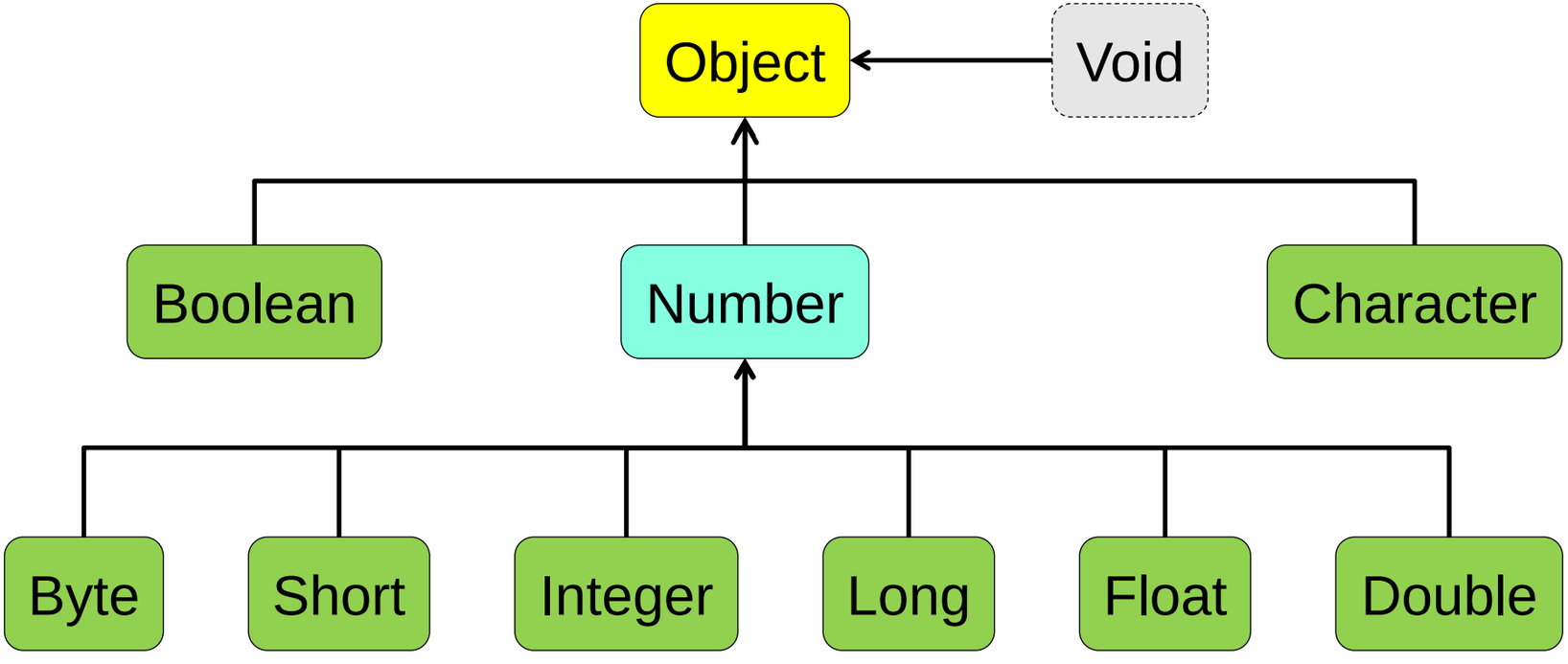
```
public class SwapNieDziala {
  public static void main(String[] args) {
    int x1=111, x2=999;
    Object o1 = new Object(), o2 = new Object();

    System.out.println("x1=" + x1 + " x2=" + x2);
    System.out.println("o1=" + o1 + " o2=" + o2)
       swap(x1, x2); swap(o1, o2);
    System.out.println("x1=" + x1 + " x2=" + x2);
    System.out.println("o1=" + o1 + " o2=" + o2);
  }

  static void swap(int a, int b) {
    int tmp = a; a = b; b = tmp;
  }

  static void swap(Object a, Object b) {
    Object tmp = a; a = b; b = tmp;
  }
}
```

```
x1=111 x2=999
o1=java.lang.Object@2a139a55 o2=java.lang.Object@15db9742
x1=111 x2=999
o1=java.lang.Object@2a139a55 o2=java.lang.Object@15db9742
```

Promocje typów

- Operatory są stosowane do wartości argumentów po wykonaniu jednej z dwu rodzajów **promocji typów**: promocji unarnej albo promocji binarnej
- **Promocja unarna**: argumenty typu **`byte, short, char`** ⇒ **`int`** (pozostałe typy pierwotne bez zmian); promocja dotyczy argumentów operatorów: +, - (unarne), ~, <<, >>, >>>, [arg]
- **Promocja binarna** (argumenty operatorów 2-argumentowych):
  - jeden z argumentów **`double`** ⇒ drugi promowany do **`double`**, **wpw**
    jeden z argumentów **`float`** ⇒ drugi promowany do **`float`**, **wpw**
  - jeden z argumentów **`long`** ⇒ drugi promowany do **`long`**, **wpw**
  - obydwa argumenty promowane do **`int`** (jeśli jeszcze nie są tego typu)
- Żadne operacje arytmetyczne nie są realizowane na wartościach z zakresu poniżej `int`; wynik obliczeń w podstawieniu może być poddany **rzutowaniu zwężającemu**.

Opakowania typów pierwotnych



- **Klasy "konfekcjonujące"** (*wrapper classes*) pozwalają posługiwać się typami pierwotnymi jak obiektami
- **Klasa Void** jest pseudotypem odpowiadającym słowu kluczowemu **void**
- **Klasa `Number`** jest klasą abstrakcyjną (dziedziczą z niej jeszcze inne klasy liczbowe: **BigDecimal, BigInteger**, …)
- Łącznie typy pierwotne i opakowujące je klasy tworzą zbiór tak zwanych typów podstawowych (**fundamental types**).

Opakowania typów pierwotnych (cd1)

- Klasy opakowujące są ***immutable***, zatem modyfikacja zawartej w obiekcie wartości jest niedopuszczalna
- Klasy te są niezbędne, gdyż nie można definiować kontenerów zawierających wartości pierwotne, np. `ArrayList<int>;` trzeba użyć `ArrayList<Integer>`
- Kompilator automatyzuje operacje opakowania / rozpakowania (**boxing / unboxing**). Np.

```
int a = 1;
Integer b = new Integer(2);
int c = a + b;
int c = a + b.intValue(); // wyłuskanie wartości
// Są dostępne konwersje (konstruktory) wg łańcucha
b = new Integer(“44”);
```

Opakowania typów pierwotnych (cd2)

- Klasa `Number` jest wspólną **abstrakcją** różnych typów liczbowych; klasa deklaruje abstrakcyjne metody rzutowania do typów konkretnych (`Integer`, ... `Double`) definiowane w podklasach:
- Z klasy **`Object`** dziedziczone są (miedzy innymi) metody: `clone, equals, getClass, hashCode, toString`

```
package java.lang;  // Wg jdk1.8.0_73
public abstract class Number implements java.io.Serializable
{ public abstract int intValue();
  public abstract long longValue();
  public abstract float floatValue();
  public abstract double doubleValue();
  public byte byteValue() { return (byte)intValue(); }
  public short shortValue() { return (short)intValue(); }
  private static final long serialVersionUID =
     -8742448824652078965L;
}
```

Opakowania typów pierwotnych (cd3)

- Klasy konkretne, np. `Integer`, definiują liczne metody użyteczne przy programowaniu obliczeń na liczbach. Przykład:

```
public static void main(String[] args) {
  // ...
  Number a1 = new Integer("43690"); // String ⇒ int
  System.out.println(a1);
  // Statyczna metoda konwersji znakowej wg podstawy zapisu
  System.out.println(Integer.toString(43690, 2));
  System.out.println(Integer.toString((int)a1, 4));
}
```

Wynik

```
43690
1010101010101010
22222222
```

==> PPJ03_PodstawyZapisuLiczb
Zmodyfikować tak, aby wynik pokazywał postawę zapisu w nawiasie, np. n = 123(4)

Instrukcje

- Repertuar instrukcji bliski C/C++ (drobne różnice)

Nazwa instrukcji	Schemat składniowy	Rola, przykłady
Instrukcja wyrażenia	*exp* ;	`a = b + f(c); ++j;`
Instrukcja sterująca	**return** *exp*; **return;** **break; break etyk;** **continue;** **continue etykieta;**	powrót z metody przerwanie iteracji lub wyboru przejście do warunku iteracji
Instrukcja deklaracji	`typ nazwa [= init], …`	`int a, b=5;` `Shape r=new Rect();`
Instrukcja złożona (blok)	{ $Inst_1$ .... $Inst_n$ } // n>=0	`{ ++i; --j; }`

Wybór: Instrukcja warunkowa	**if**(*warunek*) *Inst*
	if(*warunek*)
	Inst
	else
	Inst
		`if(a>b) f(b, a-b);`
		`if(p-- <n)`
		`x = find(p);`
		`else`
		`x = find(n);`

Instrukcje (cd1)

Nazwa instrukcji	Schemat składniowy	Rola, przykłady

Wybór:
Instrukcja switch
	switch(*warunek*)
	Inst
		switch(status) {
		case s1: op1(); **break**;
		default: err();
		}

Iteracja **while**	**while**(*warunek*) *Inst*	while(i<n) iter(i++);
Iteracja **for**	**for**(*dekl_exp*; *warunek*; *itr_exp*) *Inst*	**for**(**int** i=0; i<n; ++i) A[i] = fun(i);
Iteracja **foreach** ***(enhanced for)***	**for**(*zmienna*: *kontener*) *Instrukcja*	Pkt[] tab = **new** Pkt[10]; ... **for**(Pkt p: tab) move(p);
Iteracja **do**	**do** *Inst* **while**(*warunek*);	**do**{ fun(n); } **while** (n-->0);
Blok **try** z blokami catch i/lub blokiem **finally**	**try** { /* Blok */ } **catch**(par_1){/* $Obsługa_1$ */} .... **finally** {/* *Obsługa* */}	**try**{ f(); } **catch**(Exception e){/**/} **finally** {/* */}
Instrukcja z etykietą	*etykieta*: *Inst*	Lab: **while**(...){...}
Instrukcja pusta	; {}	{} {;;} **if**(**true**);
Asercja	**assert** *warunek* [: *exp*]$_{opt}$;	**assert** a>0 : a;

Ćwiczenie

Napisać program wyznaczający czynniki pierwsze liczb naturalnych wprowadzanych przez użytkownika. Konwersacja odbywa się w cyklu. Przyjąć, że liczba rozkładana należy do zakresu `long` ($<2^{63}$). Rozkład zaprezentować jako iloczyn czynników pierwszych (w kolejności niemalejącej) dający oryginalną liczbę rozkładaną.

Aby ocenić koszt automatycznych operacji *boxing – unboxing* przeprowadzić te same obliczenia wykorzystując typ pierwotny **long** i typ opakowujący **Long**, a następnie porównać czasy obliczeń. Do pomiaru czasu można użyć usługi:

```
System.currentTimeMillis();
```

Przykład:

```
Podaj liczbe >=1: n = 123456789
123456789 = 3*3*3607*3803
...
```


==> PPJ04_FaktoryzacjaLong

Typy referencyjne

Jest 6 rodzajów typów referencyjnych (oprócz pierwotnych):

- Typy tablicowe: struktury jednorodne o ustalonym rozmiarze do przechowywania elementów tego samego typu.
- Typy definiowane przez klasy: zapewniają hermetyzację, dziedziczenie, polimorfizm; definiowane jako programowy odpowiednik opisu realnych obiektów z reprezentacją stanu wewnętrznego (pola – dane) i zachowania (metody).
- Enumeracje: odniesienia do określonego zbioru obiektów reprezentujących (identyfikujących) pewien zestaw opcji.
- Interfejsy: definiują pewien publiczny kontrakt programowy (API) realizowany (modyfikator `implements`) przez klasy.
- Rekordy: specjalizowane klasy z niemodyfikowalnym stanem
- Adnotacje: pozwalają skojarzyć atrybuty specjalne (metadane) z różnymi elementami programu.

Tablice

- Tablice w Javie są obiektowym substytutem tablic statycznych w C/C++ (ale bez wskaźników i arytmetyki adresowej)
- Każda tablica ma określony **typ** i **liczbę** elementów (ale liczba elementów nie wchodzi do charakterystyki typu)
- Typ elementów: dowolny niegeneryczny typ Javy, także tablica
- Dostęp do elementów poprzez **indeks** (od zera)

```
int[] itab;           // itab jest null
itab = new int[10];// teraz 10 elementów 0
int[3] ttt;           // błąd: nie podaje się rozmiaru
int[] t3 = {1,2,3};// trzy elementy, t3[0]==1, ...
int[][] t2x2 = {{1,2}, {3,4}}; // tablica 2 na 2
int[] t2x2[] = {{1,2}, {3,4}}; // można i tak (unikać)
t2x2[1][1] = 44;
String[] dir = {"E", "N", "W", "S};
```

- Do tablic stosuje się bogaty repertuar usług z **java.util.Arrays**

Tablice - tablica łańcuchów, sortowanie

```
// Proste operacje na tablicach łańcuchów
import java.util.Arrays;  //  Java Collections Framework
import java.util.Comparator;

public class StringTab {

  // Zagnieżdżona klasa komparatora
  static class RevComp implements Comparator<String>{
    public int compare(String a, String b){
      return b.compareTo(a);
    }
  }

  public static void main(String[] args){
    String[] sa={"jeden","dwa","trzy","cztery","pięć"};
  // ... cd 1
```

PPJ04_ArraysSortowanie

Tablice - tablica łańcuchów, sortowanie (cd1)

```
 System.out.println("\nSortowanie rosnąco");
  Arrays.sort(sa);
  for(String s : sa){
    System.out.print(s + " ");
  }
  System.out.println("\ntrzy na pozycji " +
                      Arrays.binarySearch(sa, "trzy"));
  System.out.println("\nSortowanie malejąco");
  RevComp rc = new RevComp();

  Arrays.sort(sa, rc); // sortowanie wg relacji odwróconej
  for(String s : sa){
    System.out.print(s + " ");
  }
}
```

```
Sortowanie rosnąco
cztery dwa jeden pięć trzy
trzy na pozycji 4

Sortowanie malejąco
trzy pięć jeden dwa cztery
```

Definiowanie klas

Schemat składniowy definicji klasy (w [ ] elementy opcjonalne):

```
[modyfikatory] class NazwaKlasy
     [extends KlasaBazowa]      // Dziedziczenie
     [implements Interfejs1, Interfejs2,...]
{ // Składowe (pola) danych
  // Konstruktory, metody
  // Klasy zagnieżdżone
}
```

sygnatura klasy

ciało (treść) klasy

Jeżeli w sygnaturze brak jawnego dziedziczenia, to implicite klasa dziedziczy względem klasy Object z pakietu `java.lang`.

```
public class Point2D {   // dziedziczy wg klasy Object
  private double x, y;   // zmienne (pola) instancyjne
  public Point2D(double x,double y){ this.x=x;this.y=y; }
  public double dist() { return Math.sqrt(x*x + y*y); }
  // ...
}
```

Definiowanie metod

Schemat składniowy definicji metody nieabstrakcyjnej:

```
[modyfikatory] Typ nazwaFunkcji([parametry])
        [specyfikacja wyjątków kontrolowanych]
{
  Deklaracje / Definicje / Instrukcje
}
```

sygnatura metody

ciało (treść) metody

Przykład (wewnątrz pewnej klasy): Największy wspólny podzielnik

```
// Oblicza największy wspólny podzielnik p i q
public static int nwp(int p, int q) { // q != 0
  int t;
  while((t = p%q) != 0) {
    p = q; q = t;
  }
  return q;
}
```

Definiowanie klas parametryzowanych

- Klasa parametryczna = **klasa generyczna** (rodzajowa)
- Wg klasy generycznej można generować **klasy konkretne** zastępując parametry formalne konkretnymi argumentami (typ-parametr ↔ typ-argument)

Prosty przykład

```
// K, V to nazwy parametrów reprezentujących typy
class Para<K, V> {
  private K key; private V val;

  public Para(K key, V val){ this.key=key; this.val=val; }
  public K getKey(){ return key; }
  public V getVal(){ return val; }
  public void setKey(K newKey){ key = newKey; }
  public void setVal(V newVal){ val = newVal; }
  public String toString(){ return "("+key+", "+val+")"; }
}
```

Definiowanie klas parametryzowanych (cd1)

```
public class GenericClassTest {
  public static void main(String[] args) {

    //Para<int, int> p = new Para<>(1939, 1945); // Error
    // Typ pierwotny nie może być typ-argumentem
    Para<Integer, Integer> pii = new Para<>(1939, 1945);
    System.out.println(pii);

    Para<Integer, String> pis = new Para<>(100, "sto");
    System.out.println(pis);

    Para<String, Para<String, String>> psp =
      new Para<>("en2pl", new Para("one", "jeden"));
    System.out.println(psp);
  }
}
```

==> PPJ04_GenericClassTest

```
(1939, 1945)
(100, sto)
(en2pl, (one, jeden))
```

Definiowanie metod generycznych

- Metoda generyczna może być definiowana w klasie generycznej lub w "zwykłej" klasie
- Decyzję o zastąpieniu typ-parametru formalnego odpowiednim typ-argumentem podejmuje (na ogół) kompilator na podstawie typów argumentów wywołania metody.

Przykład wyróżnik

```
static <T>void reverse(T[] tab)
{
  int n = tab.length;
  T temp;
  for(int i=0; i<n/2; ++i)
  { temp = tab[i];
    tab[i] = tab[n-1-i];
    tab[n-1-i] = temp;
  }
}
```

```
Integer[] a = { 1, 2, 3 };
String[]  b = { "raz", "dwa", "trzy" };
reverse(a); reverse(b);
System.out.println(a);
System.out.println(b);
System.out.println(Arrays.toString(a));
System.out.println(Arrays.toString(b));
// -------------------------
[Ljava.lang.Integer;@2a139a55
[Ljava.lang.String;@15db9742
[3, 2, 1]
[trzy, dwa, raz]
```

Definiowanie metod generycznych (cd1)

```
Integer[] x = new Integer[4];
String[]  y = new String[4];
x[0] = 1; x[1]=2; x[2]=3;
y[0] = "raz"; y[1]="dwa"; y[2]="trzy";
System.out.println("x=" + x); System.out.println("y=" + y);
reverse(x); reverse(y);
System.out.println("x=" + x); System.out.println("y=" + y);
System.out.println("x=" + Arrays.toString(x));
System.out.println("y=" + Arrays.toString(y));
```

```
x=[Ljava.lang.Integer;@6d06d69c
y=[Ljava.lang.String;@7852e922
x=[Ljava.lang.Integer;@6d06d69c
y=[Ljava.lang.String;@7852e922
x=[null, 3, 2, 1]
y=[null, trzy, dwa, raz]
```

Projektowanie klas (bez jawnego dziedziczenia)

Lista pytań

- Jakie przewidujemy cechy (atrybuty, stan wewnętrzny) obiektów tworzonych wg klasy i jak je reprezentować?
- Czy obowiązują jakieś *niezmienniki* stanu wewnętrznego obiektów?
- Które atrybuty można udostępniać publicznie a które powinny być kontrolowane?
- Jak będą tworzone i inicjowane obiekty; czy dopuszczamy istnienie obiektów z nieokreślonym stanem wewnętrznym?
- Czy likwidacja obiektu wymaga czynności porządkowych (finalizacji)?
- Zachowanie obiektu: jakie operacje będą wykonywane na obiektach? Które mają być prywatne, które publiczne?
- Jakie algorytmy zastosować w operacjach?
- Jak program ma korzystać z definicji klasy?

Klasa `ulamek`

Decyzje projektowe

- Atrybuty: `long l,m;` (licznik, mianownik)
- Niezmiennik: l, m względnie pierwsze (nieskracalne); m>=0
- Dziedzina: liczby wymierne w zakresie wynikającym z reprezentacji typu long z dodatkową konwencją:

  1/0 $\Rightarrow$ +∞; -1/0 $\Rightarrow$ - ∞; 0/0 $\Rightarrow$ wartość nieokreślona
- 4 sposoby inicjowania obiektów:

  (1) bez argumentów (inicjowanie domyślne);

  (2) przez podanie wartości całkowitej $\Rightarrow$ mianownik = 1;

  (3) przez podanie pary wartości całkowitych **l, m**;

  (4) przez podanie innego obiektu **`ulamek`**.
- Usuwanie obiektu - bez żadnych czynności porządkowych

==> PPJ05_LiczbyWymierne

Klasa Ulamek: implikacje konwencji

Tabliczki dodawania i odejmowania

+	l_2/m_2	-1/0	1/0	0/0
l_1/m_1	l/m	-1/0	1/0	0/0
-1/0	-1/0	-1/0	0/0	0/0
1/0	1/0	0/0	1/0	0/0
0/0	0/0	0/0	0/0	0/0

-	l_2/m_2	-1/0	1/0	0/0
l_1/m_1	l/m	1/0	-1/0	0/0
-1/0	-1/0	0/0	-1/0	0/0
1/0	1/0	1/0	0/0	0/0
0/0	0/0	0/0	0/0	0/0

Tabliczki mnożenia i dzielenia podobnie (ćwiczenie)

Klasa Ulamek

```
// Definicja klasy Ulamek (liczby wymierne): plik Ulamek.java
package wymierne;

public class Ulamek // extends Object
{ private long l, m;

  // Inicjuje skladowe l, m obiektu wedlug argumentow a, b;
  // postac znormalizowana ułamka: nieskracalna, m >=0.
  private void initUlamek(long a, long b) {
    long d = nwp(a, b);
    if(b<0) { a = -a; b = -b; }  // Mianownik zawsze nieujemny
    l = a/d; m = b/d;
  }
  public Ulamek(long a, long b) { initUlamek(a, b); }
  public Ulamek(long a)         { initUlamek(a, 1); }
  public Ulamek()               { initUlamek(0, 1); }
  public Ulamek(Ulamek u) { initUlamek(u.l, u.m); }     // cd1
```

Konstruktor kopiujący

Klasa Ulamek (cd1)

```
public long getLicznik()    { return l; }
public long getMianownik()  { return m; }
public void setLicznik(long nl)  { initUlamek(nl, this.m); }
public void setMianownik(long nm){ initUlamek(this.l, nm); }

public Ulamek add(Ulamek b) {
  long tl, tm;
  if(this.m == 0 && b.m == 0)
    { tl = (this.l+b.l)/2;  tm = 0; }
  else
    { tl = this.l*b.m + this.m*b.l; tm = this.m*b.m; }
  return new Ulamek(tl, tm);
}
public static Ulamek add(Ulamek a, Ulamek b) {
  return a.add(b);
}                                                  // ... cd2
```

Klasa Ulamek (cd2)

```
public Ulamek sub(Ulamek b)
{ long tl, tm;
  if(this.m==0 && b.m==0)
    { tl = (this.l-b.l)/2;    tm = 0; }
  else
    { tl = this.l*b.m - this.m*b.l; tm = this.m*b.m; }
  return new Ulamek(tl, tm);
}

public Ulamek mul(Ulamek b)
{ long d1,d2;
  d1 = nwp(this.l, b.m);
  d2 = nwp(this.m, b.l);
  return new Ulamek((this.l/d1)*(b.l/d2) ,
                    (this.m/d2)*(b.m/d1) );
}
                                                  // ... cd3
```

Klasa Ulamek (cd3)

```
public Ulamek div(Ulamek b) {
   long d1,d2;
   d1 = nwp(this.l, b.l);
   d2 = nwp(this.m, b.m);
   return new Ulamek((this.l/d1)*(b.m/d2),
                     (this.m/d2)*(b.l/d1));
 }
 // Medianta pary liczb wymiernych
 public static Ulamek med(Ulamek a, Ulamek b) {
   return new Ulamek(a.l+b.l, a.m+b.m);
 }
 public Ulamek med(Ulamek b) { return med(this, b); }

 public  Ulamek neg() { return new Ulamek(-this.l, this.m); }
 public static boolean lt(Ulamek a, Ulamek b) {
    Ulamek r = a.sub(b); return r.l<0;
 }                                                  // ... cd4
```

Klasa Ulamek (cd4)

```
public static boolean gt(Ulamek a, Ulamek b) {
    Ulamek r = a.sub(b);
    return r.l>0;
 }

 public boolean gt(Ulamek b) {
    Ulamek r = this.sub(b);
    return r.l>0;
 }

 public static boolean le(Ulamek a, Ulamek b) {
   return !gt(a, b);
 }
 public  static boolean ge(Ulamek a, Ulamek b) {
   return !lt(a, b);
 }
                                                  // ... cd5
```

Klasa Ulamek (cd5)

```
public  static boolean eq(Ulamek a, Ulamek b) {
   Ulamek r = a.sub(b);
   return r.l==0;
 }

 public static boolean ne(Ulamek a, Ulamek b) {
   return !Ulamek.eq(a,b);
 }

 public boolean ne(Ulamek b) { // Podobny schemat dla innych
   return ne(this, b);
 }

 public String toString() {
   String str = "" + this.l;
   if(this.m != 1)
     str += "/" + this.m;
   return str;
 }                                                  // ... cd6
```

Klasa Ulamek (cd6)

```
  // -----
  // Najwiekszy wspólny podzielnik
  public static long nwp(long p, long q) {
    long t;
    p = Math.abs(p); q = Math.abs(q);  // nwp zawsze dodatni

    if(p == 0)
      if(q == 0) return 1; else return q;
    else
    if(q == 0) return p;

    // p>0, q>0
    while((t=p%q) !=0) {
      p = q; q = t;
    }
    return q;
  }
}  // Koniec klasy Ulamek
```

Test klasy

```
// Test klasy Ulamek. plik TestWymierne.java

package wymierne;

import java.util.Scanner;

public class TestWymierne {

  public static void main(String[] args) {
    Ulamek a = new Ulamek(1, 2);
    Ulamek b = new Ulamek(2, 6);

    Scanner scn = new Scanner(System.in);

    System.out.println("Wprowadzane liczby to pary: licz mian\n");

                                                        // ... cd1
```

Test klasy (cd1)

```
 try  {
      while(true) {
        System.out.print("Pierwsza liczba a = "); a = readUlamek(scn);
        System.out.print("Druga    liczba b = "); b = readUlamek(scn);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a+b = " + a.add(b));
        System.out.println("a-b = " + a.sub(b));
        System.out.println("a*b = " + a.mul(b));
        System.out.println("a/b = " + a.div(b));
        System.out.println("a>b = " + a.gt(b));
      }
    }
    catch(Exception e) {
      System.err.println("\nBłędny format danych - koniec programu");
    }
  } // end try

  public static Ulamek readUlamek(Scanner is) {
    long licznik   = is.nextInt(), mianownik = is.nextInt();
    return new Ulamek(licznik, mianownik);
  }
}
```

```
Pierwsza liczba a = 1 2
Druga    liczba b = 3 4
a = 1/2
b = 3/4
a+b = 5/4
a-b = -1/4
a*b = 3/8
a/b = 2/3
a>b = false
Pierwsza liczba a =
```

Praca ze strumieniami `out` i `err`

Ćwiczenie

Napisać program zachowujący się zgodnie z następującym scenariuszem:

1. Wysyła do strumienia **out** zachętę do wprowadzenia liczby całkowitej *n* - liczby generowanych losowo liczb; <=0 oznacza przejście do p. 4.
2. Jeżeli w poprzednim kroku nie było zera, to program prosi o podanie zakresu generacji *z* – liczby całkowitej dodatniej
3. mając parę liczb *n, z* program wysyła do strumienia **`out`** i do strumienia **`err`** generowane losowo (ta sama liczba do obydwu strumieni); po zakończeniu tej czynności powraca do p. 1.
4. Wyprowadzany jest komunikat o zakończeniu programu i podawana jest liczba wszystkich wygenerowanych losowo liczb.

Do generowania liczb losowych można użyć generatora Random:

```
import java.util.Random;
Random rg = new Random();     // Utwórz generator
int rnum = rg.nextInt(z)+1;   // Generuje liczbę z
                              // zakresu 1 .. z
```

==> PPJ06_LiczbyLosowe

Uwagi o stylu kodowania

Zalecenia

- Stosować nazwy ułatwiające zrozumienie kodu i zachowujące liternictwo przyjęte dla poszczególnych kategorii nazw
  - nazwy typów referencyjnych (klas, interfejsów) zawsze dużą literą (brać przykład z nazewnictwa w pakietach Javy)
  - nazwy zmiennych, parametrów zaczynać od małych liter
  - stosować "styl wielbłądzi" w nazwach wielowyrazowych (bez '_')
  - jeżeli w aplikacji obowiązuje jakaś metafora, to nazewnictwo powinno być względem niej spójne
- Komentarze - krótkie i pomocne dla czytającego
  - Komentarze ogólne dla klas i nietrywialnych metod (dla metod prostych nazwa powinna być wystarczająco sugestywna)
  - Komentarze lokalne w miejscach ważnych lub "subtelnych"
  - Komentować, nawet szkicowo, podczas pisania kodu (nie ex post)
- Uwzględniać strukturę hierarchiczną kodu
  - stosować konsekwentnie wyróżnienie zapisu dla tekstu „podległego” (wysunięcie dla treści klasy, bloku, wnętrza instrukcji strukturalnych).

Uwagi o stylu kodowania (cd1)

Kodowanie programów ćwiczebnych

- Często wystarcza 1 plik źródłowy `.java` z kompletnym kodem
- Jeżeli trzeba zdefiniować wiele klas, to należy utworzyć pakiet (nazwy pakietów zawsze małymi literami, np. z inwersją nazw domeny URL bądź wg przyjętej konwencji lokalnej – np. nazwa pakietu z ustalonym prefiksem wziętym z nazwy projektu)
- Nawet jeżeli program służy tylko do testowania języka i środowiska warto zachowywać poziom dyscypliny zbliżony do zaleceń rygorystycznych.

Zalecenia belfra

- Możliwie mało dowolności tam, gdzie jest ona zbędna
- Program trzeba **czytać**, także po uznaniu, że "działa"; zwykle okazuje się, że zawiera rozmaite "śmieci", a najczęściej potrzebuje uzupełnienia lub skorygowania dokumentacji
- Slogan Javy "**Write Once, Run Anywhere**" (WORA) można też odczytać tak "**Write Once, Read Again**"

Styl kodowania (cd 2)

Antyprzykład: co robi ten program?

```
// ???

public class StylKodowania { public static void
main(String[] args) { int k = 0; double i, j, r;
String s = " .:-;!/>)|&IH%*#"; for(double y=-17;
y<17; ++y) { for(double x = 0; x++ < 80; System.
out.print (s.charAt(k & 15))) { i = r = j = 0; k
= 0; while(j*j+i*i<11&&k++<111) { j = r*r-i*i-2+
x/25;i=2*r*i+y/10;r=j;}}System.out.println();}}}
```

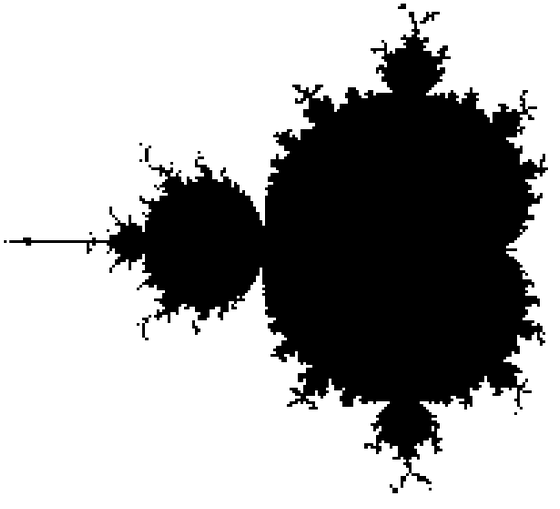
Styl kodowania (cd3)

```
// ??? Bardziej czytelne, ale bez komentarza trudne

public class StylKodowania2
{
  public static void main(String[] args) {
    int k=0;
    double i, j, r;
    String s = " .:-;!/>)|&IH%*#";

    for(double y = -17; y<17; ++y) {
      for(double x = 0; x++ < 80;
        System.out.print(s.charAt(k & 15)))
      { i = r = j = 0; k = 0;
        while(j*j+i*i < 11 && k++ < 111) {
          j = r*r-i*i-2+x/25;
          i = 2*r*i+y/10;
          r = j;
        }
      }
      System.out.println();
    }
  }
}
```

Styl kodowania (cd4)

Wynik działania programu

```
.............:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.....................
.........:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.............
.....:::::::::::::::::::::::::::::::::::::::----------:::::::::::::::::::.............
...:::::::::::::::::::::::::::::::-----------------------::::::::::::::.............
:::::::::::::::::::::::::::-------------;;;!:H!!;;;--------:::::::::::::::.........
:::::::::::::::::::::::::-------------;;;;!!/>&*|I !;;;--------:::::::::::::::.......
::::::::::::::::::::-------------;;;;;;!!/>)|.*#|>/!!;;;;-------:::::::::::::::.....
:::::::::::::::::-------------;;;;;;!!!!//>|:    !:|//!!!;;;;-----:::::::::::::::....
::::::::::::-----------;;;;;;;!!/>)I>>)||I#     H&))>////*!;;-----:::::::::::::::....
::::::::----------;;;;;;;;;;!!!//)H:  #|              IH&*I#/;;-----:::::::::::::::...
:::::---------;;;;!!!!!!!!!!//>|.H:                    #I>/!;;-----:::::::::::::::..
:----------;;;;!/||>//>>>>//>>)|%                       %|&/!;;----:::::::::::::::.
--------;;;;;!!//)& .;I*-H#&||&/                        *)/!;;-----:::::::::::::::
-----;;;;;!!!//>)IH:-        ##                        #&!!;;-----:::::::::::::::
;;;;!!!!!///>)H%.**           *                        )/!;;;------:::::::::::::::
                                                      &)/!!;;;------:::::::::::::::
;;;;!!!!!///>)H%.**           *                        )/!;;;------:::::::::::::::
-----;;;;;!!!//>)IH:-        ##                        #&!!;;-----:::::::::::::::
--------;;;;;!!//)& .;I*-H#&||&/                        *)/!;;-----:::::::::::::::
:----------;;;;!/||>//>>>>//>>)|%                       %|&/!;;----:::::::::::::::.
:::::---------;;;;!!!!!!!!!!//>|.H:                    #I>/!;;-----:::::::::::::::..
::::::::----------;;;;;;;;;;!!!//)H:  #|              IH&*I#/;;-----:::::::::::::::...
:::::::::::::-----------;;;;;;;!!/>)I>>)||I#     H&))>////*!;;-----:::::::::::::::....
:::::::::::::::::-------------;;;;;;!!!!//>|:    !:|//!!!;;;;-----:::::::::::::::....
::::::::::::::::::::-------------;;;;;;!!/>)|.*#|>/!!;;;;-------:::::::::::::::.....
:::::::::::::::::::::::::::-------------;;;;!!/>&*|I !;;;--------:::::::::::::::.......
:::::::::::::::::::::::::::::::-------------;;;!:H!!;;;--------:::::::::::::::.........
...:::::::::::::::::::::::::::::::-----------------------::::::::::::::.............
.....:::::::::::::::::::::::::::::::::::::::----------:::::::::::::::::::.............
.........:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.............
.............:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::.....................
```



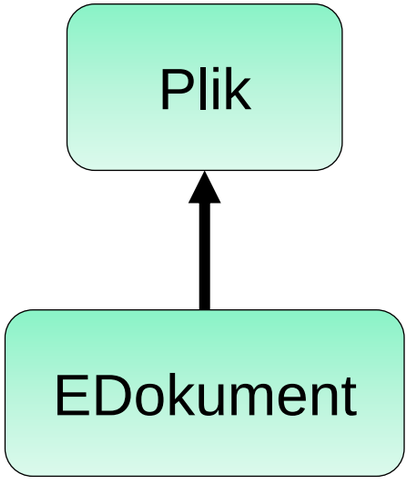
Dziedziczenie i polimorfizm

Mechanizm dziedziczenia

- Dziedziczenie: relacja pomiędzy klasami i obiektami wyrażająca przejęcie cech; dziedziczenie w Javie jest totalne
- Obiekt dziedziczący można traktować jako szczególny przypadek obiektu z którego dziedziczy - posiada wszystkie cechy tego obiektu, być może uzupełnione nowymi cechami
- Polimorfizm: ściśle związany z dziedziczeniem - obiekty należące na pewnym poziomie abstrakcji do wspólnej klasy mogą mieć odmienne cechy na poziomie szczegółowym (por. klasy `Number, Integer, Double`, ...)
- Dziedziczenie jest podstawą tworzenia klasyfikacji hierarchicznej obiektów; dobrze zdefiniowane hierarchie pozwalają uprościć strukturę złożonych systemów
- W Jawie jest JEDNA hierarchia dziedziczenia; jedyna klasa nie dziedzicząca to klasa **Object**

Dziedziczenie - terminologia

```
public class EDokument extends Plik {
   // Składowe publiczne (interfejs publiczny)
   // Składowe chronione
   // Składowe prywatne
}
```



- `EDokument` jest **subklasą** `Pliku`
- `EDokument` jest klasą pochodną względem `Pliku`
- `EDokument` jest **podklasą** `Pliku`
- `EDokument` jest specjalizacją `Pliku`
- `Plik` jest **superklasą** `EDokumentu`
- `Plik` jest klasą bazową dla `EDokumentu`
- `Plik` jest **nadklasą** `EDokumentu`
- `Plik` jest uogólnieniem `EDokumentu`

Konwencja graficzna wg UML: strzałka w górę hierarchii

Klasa Object

- Należy do pakietu java.lang
- Jedyny konstruktor tej klasy to domyślny `Object()`;

```
package java.lang;

public class Object {
  // ...
  public native int hashCode();
  public boolean equals(Object ob) { return this == ob; }
  public String toString() { return getClass().getName() + "@" +
    Integer.toHexString(hashCode());
  }
  public final native Class<?> getClass();
  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long timeout) throws
    InterruptedException;
  public final void wait() throws InterruptedException { wait(0); }
  protected native Object clone() throws CloneNotSupportedException;
  protected void finalize() throws Throwable { }
}
```

Dostęp do składowych

- Dostęp do składowych jest kontrolowany poprzez modyfikatory **private, protected i public**, a przy braku modyfikatora **domyślnie** jest ograniczony do wnętrza pakietu.
- **private:** dostępność wewnątrz klasy; z zewnątrz dostępność tylko za pośrednictwem metod (w żargonie getters / setters)
- **protected:** dostępność wewnątrz pakietu oraz w podklasach dziedziczących
- **public:** dostępność gwarantowana wszędzie

Uwaga

Java nie dopuszcza kwalifikatorów dziedziczenia obecnych w C++ (**private, protected i public**); extends zachowuje się jak dziedziczenie w trybie **public** C++. Zatem dziedziczenie nie może zmienić poziomu ochrony dostępu do składowych w nadklasie,

Dziedziczenie vs zawieranie

- Klasa **zawiera obiekt** innej klasy, jeżeli ma zadeklarowaną składową tej klasy
- Klasa **dziedziczy podobiekt** innej klasy, jeżeli jest względem niej pochodną
- Możliwe jest **odziedziczenie** podobiektu w którym są **zawarte** pewne obiekty jako składowe
- Wybór dziedziczenia albo zawierania wynika z **roli** pełnionej przez klasę względem innej klasy:
  - jeżeli uprawnione jest traktowanie obiektów pochodnych jako szczególnych przypadków obiektów bazowych → **dziedziczenie**
  - jeżeli powyższe jest nieuprawnione, albo obiekt musi posiadać kilka instancji innej klasy → **zawieranie**
- **Tylko dziedziczenie może być podstawą zachowania polimorficznego klasy**

Dziedziczenie i zawieranie (cd1)

```
public class Punkt {
  private double x, y;
  public Punkt(double xx, double yy) { x=xx; y=yy; }
  // ... Inne składowe
}

// Zawieranie: Odcinek musi mieć 2 Punkty końcowe
public class Odcinek {
  protected Punkt p1, p2;
  public Odcinek(){ /* ... */ }
  public Odcinek(Punkt a, Punkt b) { p1=a; p2=b;
  // ... Pozostałe składowe
}

// Dziedziczenie: Wektor jest przypadkiem Odcinka
public class Wektor extends Odcinek {

  // ...
}
```

Referencje w hierarchii dziedziczenia

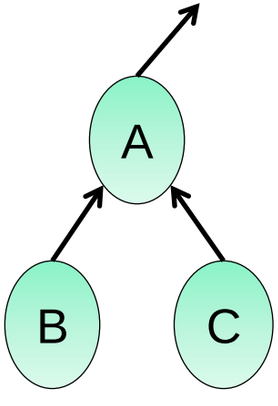
- Zmienna (referencja) klasy pochodnej może zawsze być przypisana do zmiennej klasy bazowej (obiekt klasy pochodnej jest szczególnym przypadkiem nadklasy).

◀

```
Odcinek odcinek1 = null, odcinek2 = new Odcinek();
Wektor wektor1 = new Wektor(), wektor2 = null;
odcinek1 = wektor1;  // OK, konwersja standardowa (**)
wektor2  = odcinek2; // Błąd
wektor2  = odcinek1; // Błąd, mimo (**)
wektor2  = (Wektor)odcinek1; // OK, programista wie
```

- Konwersje "w poprzek" hierarchii są zawsze błędne:

```
B b = new C();  // Błąd
```



Składowe statyczne i instancyjne

- Składowe klasy są dwu rodzajów
  - statyczne
  - instancyjne
- Składowe statyczne (*class variables, class methods*) istnieją niezależnie od obiektów danej klasy; są wspólnym wyposażeniem klasy, z którego można korzystać nie powołując się na obiekty
- Składowe instancyjne (niestatyczne) są obecne w każdym obiekcie danej klasy i muszą być inicjowane przez konstruktory klasy
- Metody niestatyczne są wywoływane na rzecz obiektu i realizują zachowanie obiektu względem innych obiektów
- Wywołanie metody niestatycznej jest w terminologii obiektowej traktowane jak przekazanie komunikatu między obiektami
- Metoda niestatyczna ma dostęp do "swojego" obiektu przez referencję **this**.

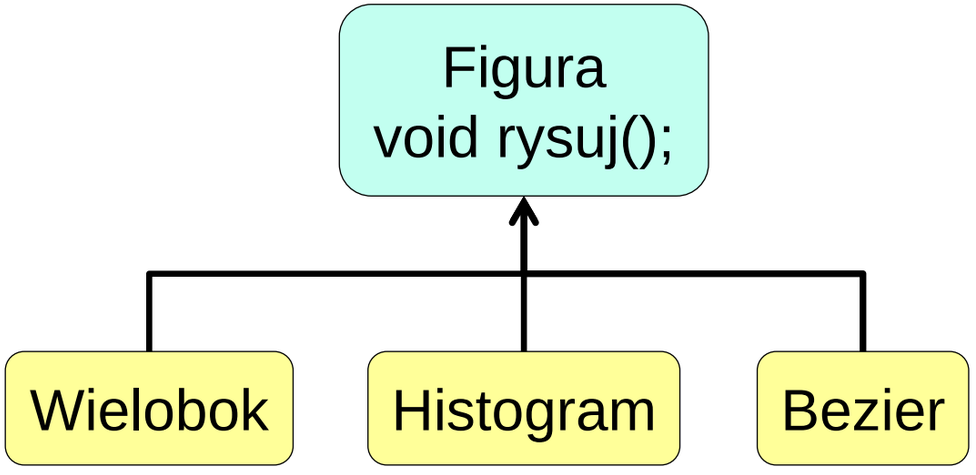
Ćwiczenie: Zmodyfikować klasę Ulamek tak, aby kończąc aplikację testującą można było się dowiedzieć ile obiektów tej klasy powstało w czasie działania programu.

Funkcje wirtualne i polimorfizm

- Metoda niestatyczna jest domyślnie wirtualna (polimorficzna) , a więc może być podmieniona (overridden) na specjalizowaną wersję w podklasie (w C++ trzeba użyć kwalifikatora virtual)
- Polimorfizm funkcji wirtualnych:
  - Funkcja **`wirtualna`** w klasie bazowej określa pewien aspekt potencjalnego zachowania obiektów w głąb hierarchii
  - zachowanie to można dostosować (**podmienić**) w klasie pochodnej zgodnie z potrzebami tej klasy; nowa wersja staje się domyślną realizacją dla kolejnego poziomu dziedziczenia
  - Na przykład: jeżeli klasa **Plik** definiuje metodę **open()**, to w podklasie **EDokument** możemy podmienić tę metodę tak, aby dokument został pokazany w przeglądarce:

```
Plik plik = new Edokument(path);
plik.open();   // Zachowanie zgodne z definicją w EDokument
```

Klasy i metody abstrakcyjne. Przykład



- Chcemy, aby klasa Figura (implicite dziedziczy z klasy Object) ustanowiła pewien standard reprezentacji i zachowania figur geometrycznych – jej podklas.
- Klasę Figura definiujemy jako klasę **abstrakcyjną** ponieważ niektóre jej metody nie mogą być zdefiniowane konkretnie – są **abstrakcyjne** (np. `rysuj()`); wiemy, że te metody muszą istnieć.
- Aby podklasa klasy Figura stała się klasą konkretną **wszystkie** odziedziczone **metody abstrakcyjne** muszą być zdefiniowane.
- Klasa z metodą abstrakcyjną musi być też abstrakcyjna.

Przykład

```
// Dziedziczenie, klasy i metody abstrakcyjne
// plik FiguryGeometryczne.java

//Figury geometryczne
abstract class Figura {  // Klasa abstrakcyjna
  abstract void rysuj(); // Metoda abstrakcyjna
  void transform(){/* ... */} // Metoda wirtualna "zwykła"
  // inne pola i metody
}

// W podklasach metoda rysuj() pozorowana komunikatami
class Wielobok extends Figura {
  Wielobok()  { System.out.println("Powstał wielobok"); }
  void rysuj() { System.out.println("Wielobok.rysuj()"); }
}
                                                  // ... cd1
```

Przykład (cd1)

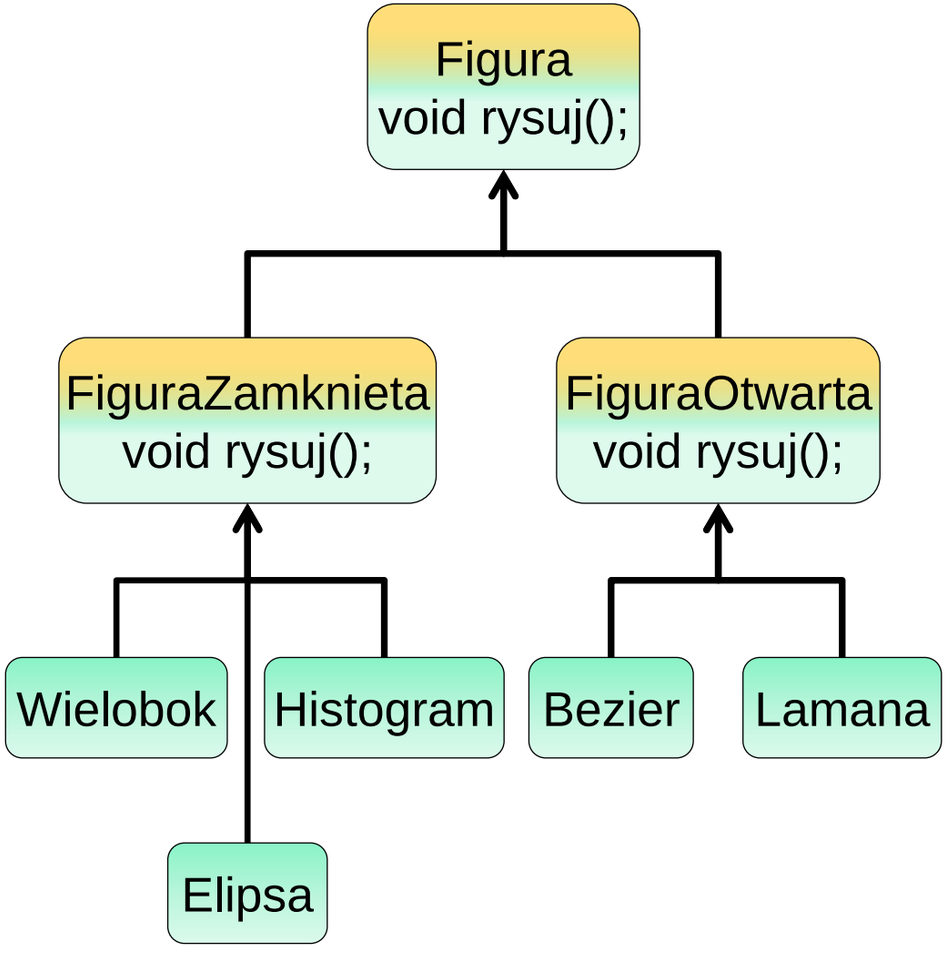
```
class Histogram extends Figura {
  Histogram() {
    System.out.println("Powstał histogtam");
  }
  void rysuj() {
    System.out.println("Histogram.rysuj()");
  }
}

class Bezier extends Figura {
  Bezier() {
    System.out.println("Powstała krzywa Beziera");
  }
  void rysuj() {
    System.out.println("Bezier.rysuj()");
  }
}                                                    // ... cd2
```

Przykład (cd2)

```
public class FiguryGeometryczne {
  public static void main(String[] args) {
    //Figura ff = new Figura();  // Nie można tworzyć instancji Figura
    Figura hf = new Histogram();
    Figura wf = new Wielobok();
    Figura bf = new Bezier();

    Figura[] figury = new Figura[]{ hf, bf, wf, hf, null };
    System.out.println("\nPrzed pokazem");
    rysuj(figury);
    System.out.println("Po pokazie\n");
 }

  static void rysuj(Figura[] figury) {
    int i;
    for(i=0; figury[i] != null; ++i)
      figury[i].rysuj();

    System.out.println("Liczba figur: " + i);
  }
}
```

```
Powstał histogram
Powstał wielobok
Powstała krzywa Beziera

Przed pokazem
Histogram.rysuj()
Bezier.rysuj()
Wielobok.rysuj()
Histogram.rysuj()
Liczba figur: 4
Po pokazie
```

Hierarchia klas abstrakcyjnych




```
abstract class FiguraZamknieta
                  extends Figura {}
abstract class FiguraOtwarta
        extends Figura {}
class Lamana extends FiguraOtwarta {
 Lamana() {
  System.out.println("Powstała lamana");
 }
 void rysuj(){
  System.out.println("Lamana.rysuj()");
  }
}
class Elipsa extends FiguraZamknieta {
 Elipsa() {
  System.out.println("Powstała elipsa");
 }
 void rysuj(){
  System.out.println("Elipsa.rysuj()");
 }
}
```

Enumeracje

Enumeracje – specjalizowane klasy zawierające ustalony (statyczny) wykaz obiektów oznaczonych mnemonicznymi nazwami (z możliwością dodawania dodatkowych atrybutów):

```
enum Agent {                  // "Wyliczanka" 3 agentów
  KLOSS("J23"),               // Wywołuje Agent("J23");
  BOND("007"),
  SKARBEK("Granville"); // Polska agentka w GB

  private final String pseudonim;// atrybut
  private Agent(String pseudo) { // Konstruktor
    pseudonim = pseudo;
  }
  public String getPseudo() {
    return pseudonim;
  }
}
// ...
for(Agent a : Agent.values())
  System.out.println(a + " " + a.getPseudo());
```

```
KLOSS J23
BOND 007
SKARBEK Granville
```

Interfejsy

- Interfejs, podobnie jak klasa definiuje nowy **typ referencyjny** ale (zgodnie z nazwą) określa tylko pewien zbiór metod
- Metody interfejsu są implicite deklarowane jako **abstrakcyjne publiczne**; klasa implementująca **musi** zdefiniować te metody
- Od wersji Java 8 istnieje możliwość zdefiniowania w interfejsie metod **domyślnych** (słowo kluczowe **`default`**); klasa implementująca **może** zdefiniować własną wersję
- Wg interfejsu nie można tworzyć obiektów
- Obiekty utworzone wg klasy ***Klasa*** implementującej interfejs ***Interfejs*** należą jednocześnie do typu ***Klasa*** i ***Interfejs***.
- Interfejsy, podobnie jak klasy, mogą wchodzić w relacje dziedziczenia (**interface** FullFun **extends** BasicFun); relacja extends w tym przypadku może dotyczyć **wielu** interfejsów

Interfejsy (cd1)

- W definicji interfejsu nie mogą pojawić się pola instancyjne
- Interfejs może zawierać:
  - **stałe** (deklarowane z modyfikatorami `static final`)
  - **metody statyczne** (od Java 8)
  - **typy zagnieżdżone**

```
interface Dokowalny {
  void dokuj(int x, int y); // Implicite public abstract
  // ... inne metody
}
interface Dekorowalny extends Dokowalny, Klonowalny {
  void dekoruj(int wariant);
  // ...
}
```

Interfejsy i klasy abstrakcyjne

- Interfejsy i klasy abstrakcyjne służą podobnym celom, ale w inny sposób wg specyficznych możliwości; **mogą być generyczne**

Interfejs	Klasa abstrakcyjna
• Bez możliwości tworzenia instancji	• Bez możliwości tworzenia instancji
• Metody abstrakcyjne i domyślne	• Metody abstrakcyjne i inne
• Pola tylko public static final	• Pola dowolne, np. protected, private
• Dziedziczenie (extends) względem dowolnej liczby interfejsów	• Dziedziczenie (extends) względem tylko jednej nadklasy
• Modyfikator implements nie ma zastosowania	• Może deklarować realizację interfejsów (implements)
Główne zastosowanie	**Główne zastosowanie**
• Opis funkcjonalności dla dowolnych klas (także bez dziedziczenia)	• Współdzielenie kodu w ramach hierarchii dziedziczenia
• Możliwość realizacji wielu niezależnych scenariuszy zachowania	• Standaryzacja zachowania i części stanu obiektów w podklasach

Interfejsy i klasy abstrakcyjne - przykład

```
// FiguryInterfejsy.java (fragment)

interface Dokowalny {
  void dokuj(int x, int y); // public abstract
  // ... inne metody
}
interface Dekorowalny extends Dokowalny {
  void dekoruj(int wariant);
  // ...
}
abstract class Figura implements Dekorowalny {
  abstract void rysuj();          // Metoda abstrakcyjna
  void transform(){ /* ... */ } // Metoda "zwykła"
  public void dokuj(int x, int y){ this.x=x; this.y=y; /*...*/}
  public void dekoruj(int w) { this.wariant = w; ; /*...*/ }
  private int x, y, wariant;
  // inne metody
}
```

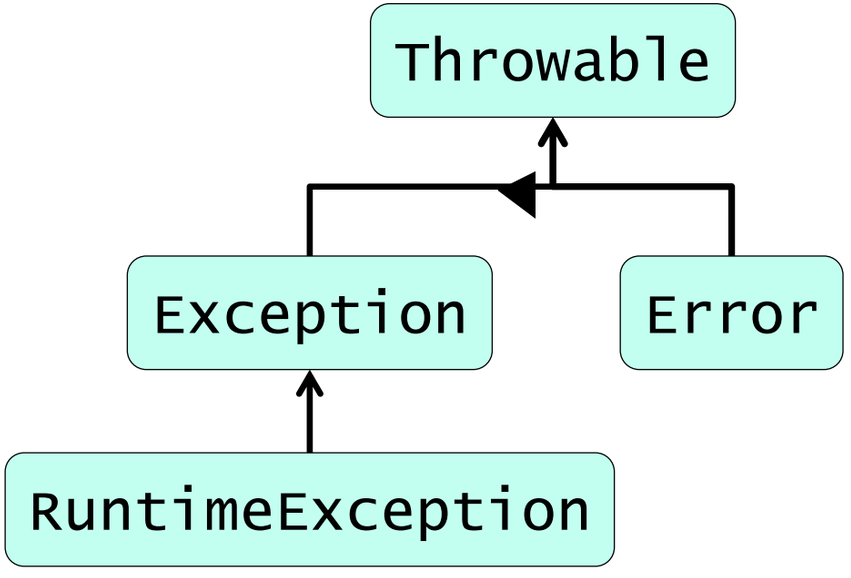
Interfejs "zespolony"

Klasa abstrakcyjna
nie musi implementować
wszystkich metod interfejsu

Wyjątki

- Wyjątek – zdarzenie w czasie wykonania programu uniemożliwiające normalny przebieg przetwarzania
- Przyczyną wystąpienia wyjątku może być:
  - uchybienie regułom semantycznym języka (np. próba wywołania metody poprzez referencję null, dzielenie przez 0)
  - awaria środowiska wykonawczego (np. brak pamięci)
  - awaria mediów / zasobów w otoczeniu programu (np. zerwane połączenie sieciowe)
- Brutalne zdarzenia (typu brak zasilania) nie podlegają obsłudze.
- Program może sam zgłosić wyjątek w sytuacji ewidentnego zagrożenia, co uruchamia procedurę obsługi wyjątku i pozwala przenieść sterowanie do odpowiedniego **punktu kompetencji**.
- W Javie wyjątki są reprezentowane przy pomocy obiektów wyprowadzonych z klasy **Throwable** (dziedziczy bezpośrednio z Object) lub jej podklas.

Wyjątki - klasyfikacja

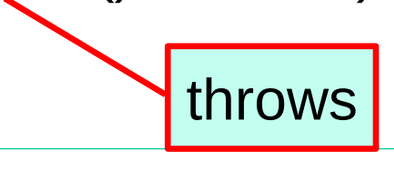


- Względem `Exception` dziedziczą wszystkie wyjątki potencjalnie nadające się do obsługi naprawczej (np. `IOException`)
- `RuntimeException` i podklasy służą do reprezentowania wyjątków zgłaszanych w czasie ewaluacji wyrażeń ale możliwe do obsługi (`ArithmeticException`, `NullPointerException`, ...)
- Klasa `Error` i podklasy reprezentują sytuacje praktycznie nienaprawialne (`IOError, ThreadDeath`, ...)

Wyjątki - klasyfikacja

- Wyjątki `RuntimeException` (i podklasy) oraz `Error` (i podklasy) są zbiorczo nazywane *niekontrolowanymi* (*unchecked exceptions*)
- Pozostałe wyjątki `Exception` (i podklas) tworzą kategorię wyjątków kontrolowanych (*checked exceptions*).
- Technicznie rozróżnia się 3 rodzaje zgłoszeń wyjątków:
  - przy pomocy instrukcji `throw` *`wyrażenie`*;
  - wyjątek synchroniczny - maszyna wirtualna wykryła sytuację nieprawidłową (np. ewaluacja wyrażenia narusza reguły semantyczne)
  - wyjątek asynchroniczny (na ogół błąd krytyczny)
- Wyjątki kontrolowane są pod nadzorem kompilatora; jeżeli metoda wywołuje inną zgłaszającą wyjątek (fraza **throws**) to też powinna zawierać frazę throws, chyba że ma odpowiedni blok catch:

```
void metoda1() throws Exception { /*  ... */ metoda2();  /* ... */ )
```



Schemat składniowy obsługi wyjątków

```
//  Blok try z obsługą wyjątków
try {
  // Blok try zawiera kod potencjalnie "niebezpieczny"
}
catch(Wyjątek1 w1) { /* obsługa1 */ }
...
catch(Wyjątekn wn) { /* obsługan */ }  // n >= 0
finally { /* blok finally */ } // opcjonalnie jeśli catch

//  Blok try na zasobie(zasobach) realizujących
//  interfejs java.lang.AutoCloseable, java.io.Closeable
try (zasób1; /*...*/ zasóbk) {
  // Operacje na zasobach
} // Zamknięcie gwarantowane jak z blokiem finally
```

Przykład try na zasobie

```
import java.util.Scanner;

public class PodstawyZapisu {
  public static void main(String[] args) {

    try (Scanner scn = new Scanner(System.in)) {
      long n; // n - liczba do prezentacji rozwinięć
      int  p; // Podstawa zapisu { 2 .. 36 }

      System.out.println("Liczba n w zapisie przy podstawie p.");
      System.out.println("Podstawa p z[2..36]; n=0 kończy program\n");

      while(true) { // Konwersacja
        System.out.print("n = ");  n = scn.nextLong();
        if(n<=0) break;

        System.out.print("p = "); p = scn.nextInt();
        if(p<2 || p>36) break;
        System.out.println("n = " + Long.toString(n, p));
      }
    }  // Automatycznie scn.close();
  }
}
```

```
n = 123456
p = 20
n = f8cg
n = 0
```

O kontenerach generycznych (stos)

```
// Wersja (historyczna) z tablicą Object[]
class Stos {
  private Object[] stos;
  private int top;  // Indeks wolnego miejsca

  public Stos(int size){ stos=new Object[size]; top=0; } // Stos pusty

  public void push(Object e) throws Exception {
    if(top == stos.length) throw new Exception("Stos pełny");
    stos[top++] = e;
  }

  public Object pop() throws Exception {
    if (top == 0) throw new Exception("Stos pusty");
    Object e = stos[--top];
    stos[top] = null;  // Kasowanie rezydenta
    return e;
  }

  public boolean empty() { return top == 0; }
  public boolean full () { return top == stos.length; }
}
```

O kontenerach generycznych (stos) cd1

```
//Wersja generyczna
class Stack<E> {
  private E[] stack;
  private int top;  // Indeks wolnego miejsca

  @SuppressWarnings("unchecked")
  public Stack(int size) { stack = (E[]) new Object[size]; top = 0; }

  public void push(E e) throws Exception {
    if(top == stack.length) throw new Exception("Stack full");
    stack[top++] = e;
  }

  public E pop() throws Exception {
    if (top == 0) throw new Exception("Stack empty");
    E e = stack[--top];
    stack[top] = null;  // Kasowanie rezydenta
    return e;
  }

  public boolean empty() { return top == 0; }
  public boolean full () { return top == stack.length; }
}
```

O kontenerach generycznych (stos) cd2

```
// Stos z tablicą Object[]
Stos s1 = new Stos(10);
s1.push(new Integer(11));
s1.push(22);
s1.push("wstawka");
s1.push(3.3);
System.out.println("Stos z tablicą Object[]");
while(!s1.empty())
  System.out.print(s1.pop() + " ");
System.out.println();

// Stos generyczny
Stack<String> s2 = new Stack<>(10);
//s2.push(new Integer(111)); // Tylko String
s2.push("111");
s2.push("222");
s2.push("WSTAWKA");
s2.push("333");
System.out.println("Stos generyczny Stack<String>");
while(!s2.empty())
  System.out.print(s2.pop() + " ");
System.out.println();
// ...
```

```
Stos z tablicą Object[]
3.3 wstawka 22 11
Stos generyczny Stack<String>
333 WSTAWKA 222 111
```

Zmodyfikować klasę Stos (Stack) tak, aby wyeliminować błąd przepełnienia stosu. Skorzystać z usług Arrays.copyOf(...).

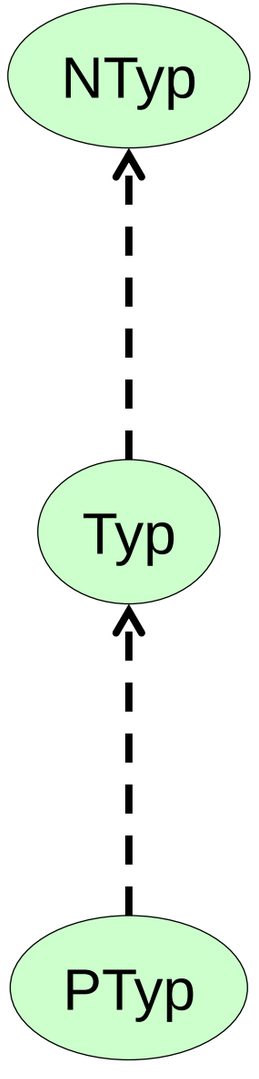
Typy wieloznaczne (wildcard)

- Często parametryzacja metody generycznej wymaga złagodzenia wymagań na zgodność typów
- Niech NTyp, Typ, PTyp będą typami w hierarchii; generalnie obowiązuje reguła substytucji:

```
NTyp nt1 = new PTyp();// OK, dziedziczenie
NTyp nt2 = new Typ();
Typ  t   = new Ptyp();
```

- Ale typy `Kontener<NTyp>`, `Kontener<Typ>`, `Kontener<PTyp>` są względem siebie **niezależne** i inwariantne (np. Stack<Number> i Stack<Integer>)
- Pytanie: jak napisać metodę generyczną move(....):

```
Stack<Number> sn = ....;
Stack<Integer>si = ....;
move(si, sn); // Dopisz zawartość si do sn
```



Typy wieloznaczne (wildcard) cd1

```
// Ta wersja nie zadziała
public static <T> void move(Stack<T> src, Stack<T> dst)
  throws Exception
{ while(!src.empty())
    dst.push(src.pop());
}

// Wersja z ograniczeniem dolnym i górnym dla wildcard
public static <T> void move(Stack<? extends T> src,
                            Stack<? super T>   dst)
      throws Exception
{ while(!src.empty())
  dst.push(src.pop());
}
// ...
Stack<Number> sn = ....; // push 4.56, 7.89
Stack<Double> sd = ....; // push 11, 22, 33
move(si, sn); // Dopisz zawartość si do sn
// Zawartość sn (w głąb stosu): 11 22 33 7.89 4.56
```

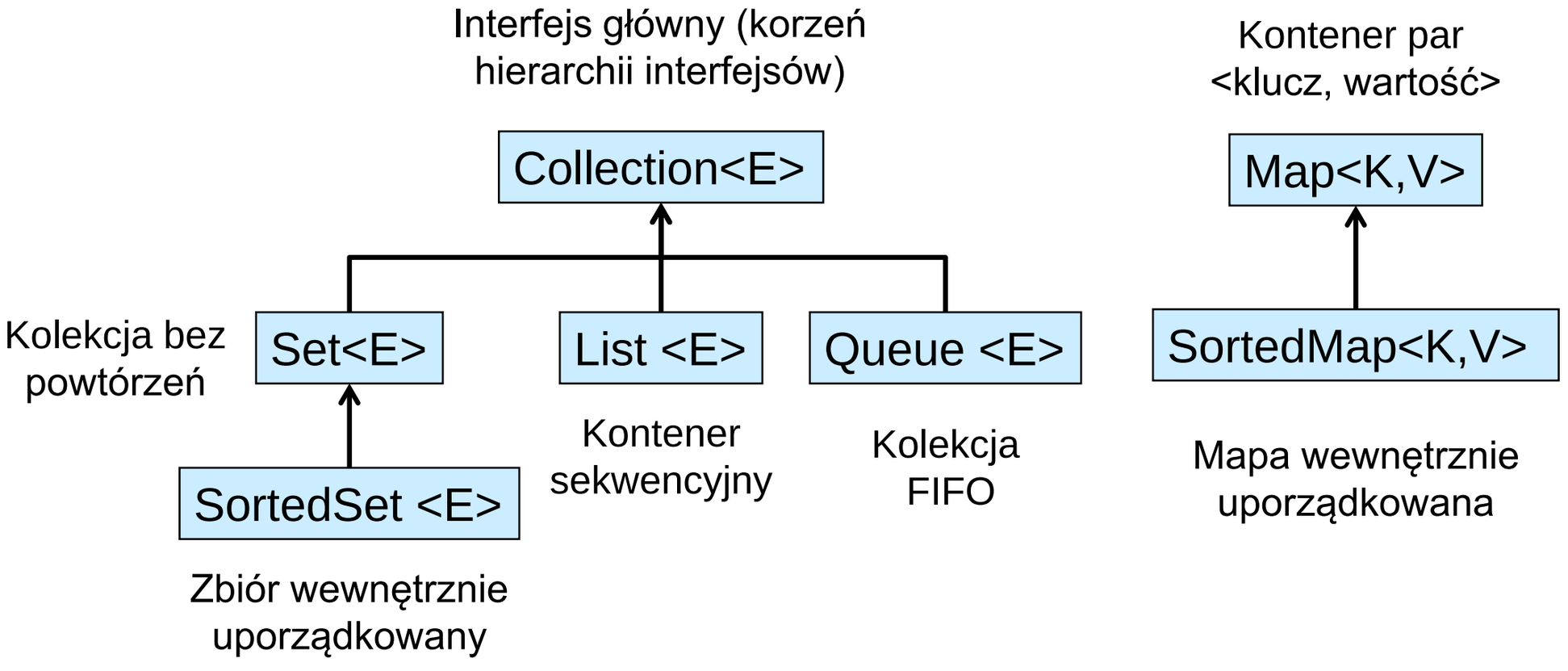
upper bound

lower bound

JCF – Java Collections Framework

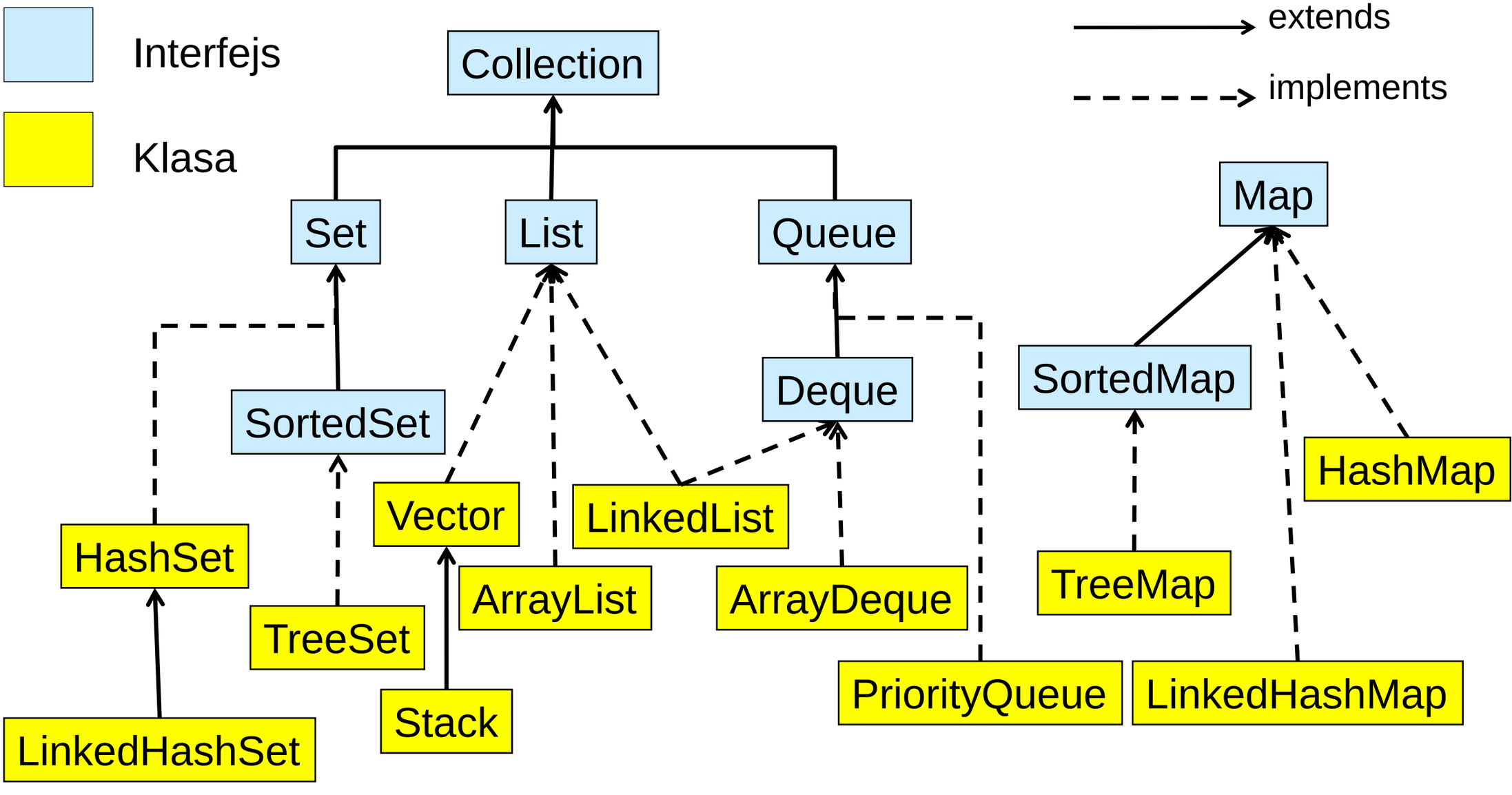
- JCF jest odpowiedzią Javy na "wyzwanie" C++ w postaci biblioteki rodzajowej STL – Standard Template Library (od Java 1.2/5.0)
- JCF składa się z interfejsów definiujących protokoły korzystania z różnych kolekcji i implementujących je klas
- Kolekcje są wyprowadzone z dwu głównych interfejsów:
  - `java.util.Collection<E>`
  - `java.util.Map<K,V>`
- Dodatkowe interfejsy to
  - `java.util.Iterator<E>`
  - `java.lang.Iterable<T>`
- Typy w JCF są parametryzowane (generyczne) pozwalając na bardzo elastyczne budowanie struktur danych

JCF (cd1)



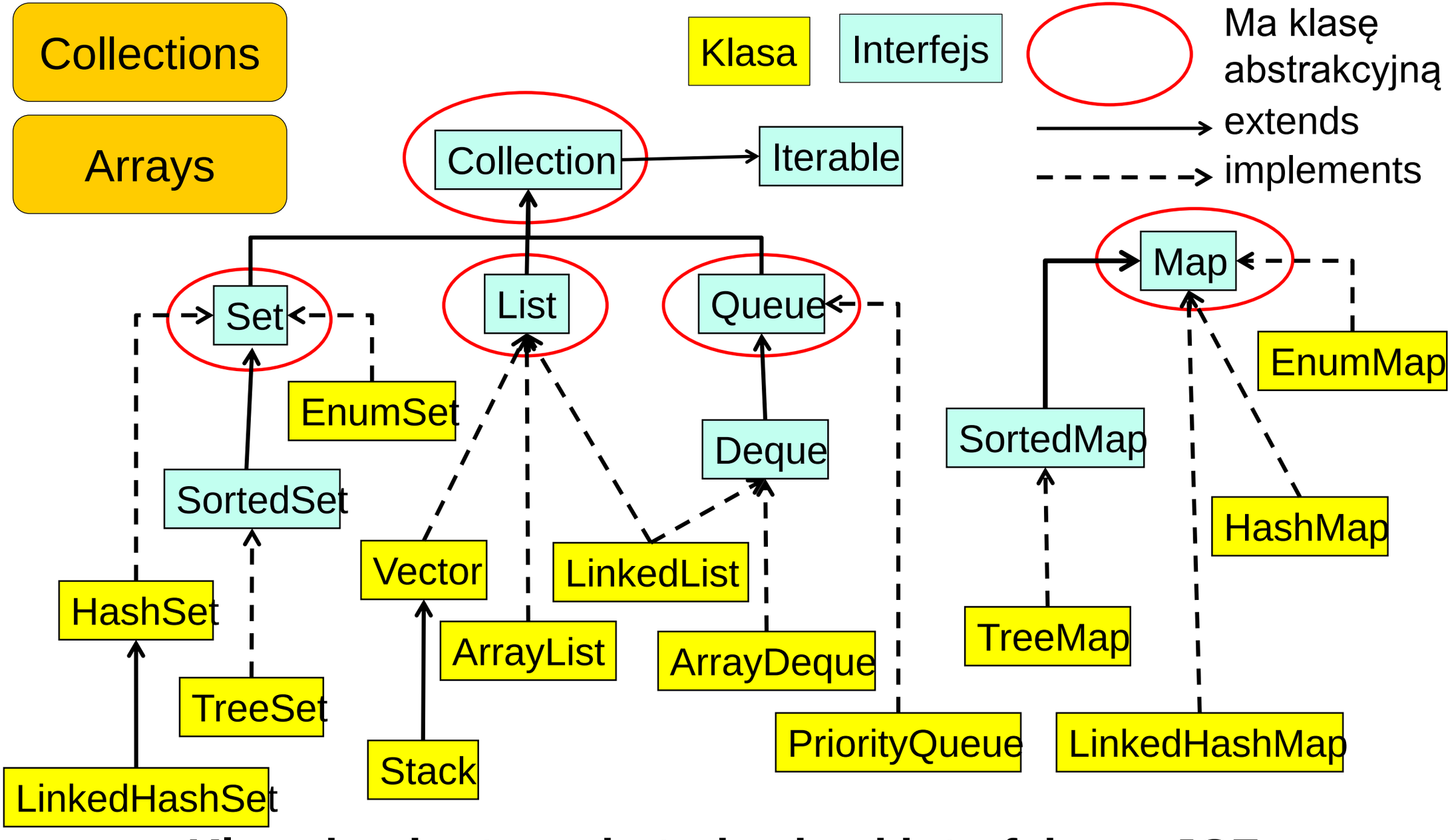
Uproszczona struktura zależności interfejsów w JCF

JCF (cd2)



Najważniejsze klasy i interfejsy w JCF

JCF (cd3)



Klasy konkretne, abstrakcyjne i interfejsy w JCF

JCF (cd4)

Metody interfejsu Collection<E>		
boolean	add(E e)	OPT
boolean	addAll(Collection<? extends E> c)	OPT
boolean	remove(Object o)	OPT
boolean	removeAll(Collection<?> c)	OPT
boolean	retainAll(Collection<?> c)	OPT
void	clear()	OPT
int	size()	
boolean	isEmpty()	
boolean	equals(Object o)	
boolean	contains(Object o)	
boolean	containsAll(Collection<?> c)	
int	hashCode()	
Iterator<E>	iterator()	
Object[]	toArray()	
<T> T[]	toArray(T[] a)	
default boolean	removeIf(Predicate<? super E> filter)	
default Stream<E>	parallelStream()	
default Spliterator<E>	spliterator()	
default Stream<E>	stream()	

Przykład – zliczanie słów w strumieniu WE

```
// Zliczanie słów różnych z wejścia in (Set<String>)

import java.util.Scanner;
import java.util.TreeSet;

public class ZliczanieSlowIn {

  public static void main(String[] args) throws Exception {
    TreeSet<String> words = new TreeSet<String>();
    Scanner scn = new Scanner(System.in);

    System.out.println(
         "Wpisz tekst (Ctrl+Z kończy wprowadzanie)\n");

                                                  // ...cd1
```

Przykład – zliczanie słów w strumieniu WE (cd1)

```
    while (scn.hasNext()) {
      String word = scn.next();
      words.add(word);
    }

    System.out.println(
        "Liczba słów różnych: " + words.size());

    // Wykaz słów
    for(String w: words)
      System.out.println(w);
    scn.close();
  }
}
```

```
Wpisz tekst (Ctrl+Z kończy wprowadzanie)

public static void main(
public static void main(
Liczba słów różnych: 4
main(
public
static
void
```

Zmodyfikować program tak, aby można było wczytywać tekst z klawiatury albo ze wskazanego pliku.