Język Java i technologie WEB

dr inż. Andrzej Czajkowski
Instyt Sterowania i Systemów Informatycznych
Wydział Informatyki, Elektrotechniki i Automatyki

Plan Wykładu

dr inż. Andrzej Czajkowski

e-mail: a.czajkowski@issi.uz.zgora.pl
pokój: 325 A-2
tel: +68 328 2276
WWW: http://staff.uz.zgora.pl/aczajkow/

Warunki zaliczenia

- **Wykład** - warunkiem zaliczenia jest uzyskanie pozytywnej oceny z egzaminu przeprowadzonego w formie pisemnej.
- **Laboratorium** - warunkiem zaliczenia jest uzyskanie pozytywnych ocen ze wszystkich ćwiczeń laboratoryjnych, przewidzianych do realizacji w ramach programu laboratorium.
- **Metody weryfikacji** - wykład: egzamin w formie pisemnej - laboratorium: sprawdzian praktyczny.
- **Składowe oceny końcowej** = wykład: 50% + laboratorium: 50%

- **Udogodnienia**: Termin zerowy na ostatnim wykładzie, zwolnienie na podstawie oceny z laboratorium (ocena **bdb**).

Literatura

Literatura obowiązkowa:

1. Java Platform, Standard Edition, API Documentation `https://docs.oracle.com/javase`
2. Bruce Eckel, Thinking in Java, Wydanie 4, Helion, 2011. `http://mindview.net/Books/TIJ4`
3. Benjamin J Evans, David Flanaga, Java w pigułce. Wydanie VI

Literatura dodatkowa:

1. J . Gosling, B. Joy, G. Steele, G. Bracha, Java Language Specification, Addison-Wesley Professional.
2. Andrzej Marciniak, JavaServer Faces i Eclipse Galileo. Tworzenie aplikacji WWW, Helion 2010.
3. Cay S. Horstmann, Java 8. Przewodnik doświadczonego programisty.

Najważniejsze skróty

- J2SE
- J2EE
- JDK
- JRE
- JVM
- JIT
- JAR
- AWT

Java - historia

- Java jest wysokopoziomowym, kompilowanym, obiektowym językiem programowania z **silną kontrolą typów**.
- Stworzony przez grupę roboczą pod kierunkiem **Jamesa Goslinga** z firmy Sun Microsystems (obecnie przejęty przez Oracle).
- **Główne założenie**: "write once, run anywhere" (WORA)
- **Podstawowe koncepcje** zostały przejęte z języka Smalltalk (maszyna wirtualna, zarządzanie pamięcią) oraz z języka C++ (duża część składni i słów kluczowych).

Wersje

- JDK 1.0 (January 21, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 LTS (March 18, 2014)
- Java SE 9 (September, 2017)
- Java SE 10 (March, 2018)
- Java SE 11 LTS (September, 2018)
- Java SE 12 (March, 2019)

Mój pierwszy program

```
package wyklad1;

public class WitajSwiecie {
 public static void main(String[] args)
 {
  System.out.println("Witaj␣Swiecie");
 }
}
```

- nazwa pakietu
- nazwa klasy
- metoda statyczna
- wyświetlanie tekstu na konsoli

Zadanie domowe:

Zainstalowanie JDK i IDE (Eclipse, NetBeans lub IntelliJ) i uruchomienie powyższego programu.

Założenia języka

1. It must be "simple, object-oriented, and familiar".

Prosty (ang. simple) – łatwy do nauczenia w bardzo krótkim czasie, zwłaszcza dla programistów znających C++ – **znajomy**, (ang. familiar).
Zorientowany obiektowo (ang. object-oriented) – projektowanie obiektowe jest techniką, która kładzie nacisk na dane (obiekty) oraz interfejsy do nich.

Założenia języka

1. It must be "simple, object-oriented, and familiar".
2. It must be "robust and secure".

Odporny (ang. robust) – poświęcony jest duży nacisk na wczesne sprawdzanie możliwych błędów (podobnie jak w C++), sprawdzanie dynamiczne w trakcie pracy oraz eliminowanie sytuacji które mogą skłaniać do wystąpienia błędu.
Bezpieczny (ang. secure) – Java zostala zaprojektowana jako język dla systemów sieciowych/rozproszonych, co spowodwalo duży nacisk na problem bezpieczeństwa

Założenia języka

1. It must be "simple, object-oriented, and familiar".
2. It must be "robust and secure".
3. It must be "architecture-neutral and portable".

Neutralny (ang. architecture neutral) – kompilator generuje instrukcje w kodzie bajtowym, który jest wykonywalny na wielu maszynach pod warunkiem obecności maszyny wirtualnej Javy. Instrukcje w kodzie bajtowym nie mają nic wspólnego ze szczególną architekturą komputera – **przenośny** (ang. portable).

Założenia języka

1. It must be "simple, object-oriented, and familiar".
2. It must be "robust and secure".
3. It must be "architecture-neutral and portable".
4. It must execute with "high performance".

Wysoce wydajny (ang. high performance) Proces właściwego generowania kodu maszynowego z kodu bajtowego jest bardzo prosty, gdyż format kodu bajtowego został zaprojektowany z myślą o optymalizacji kodu maszynowego. Najbardziej kontrowersyjny aspekt platformy Java, w nowszych wersjach platformy założenie osiągnięte poprzez takie zaawansowane mechanizmy jak JIT czy współbieżne odśmiecanie pamięci.

Założenia języka

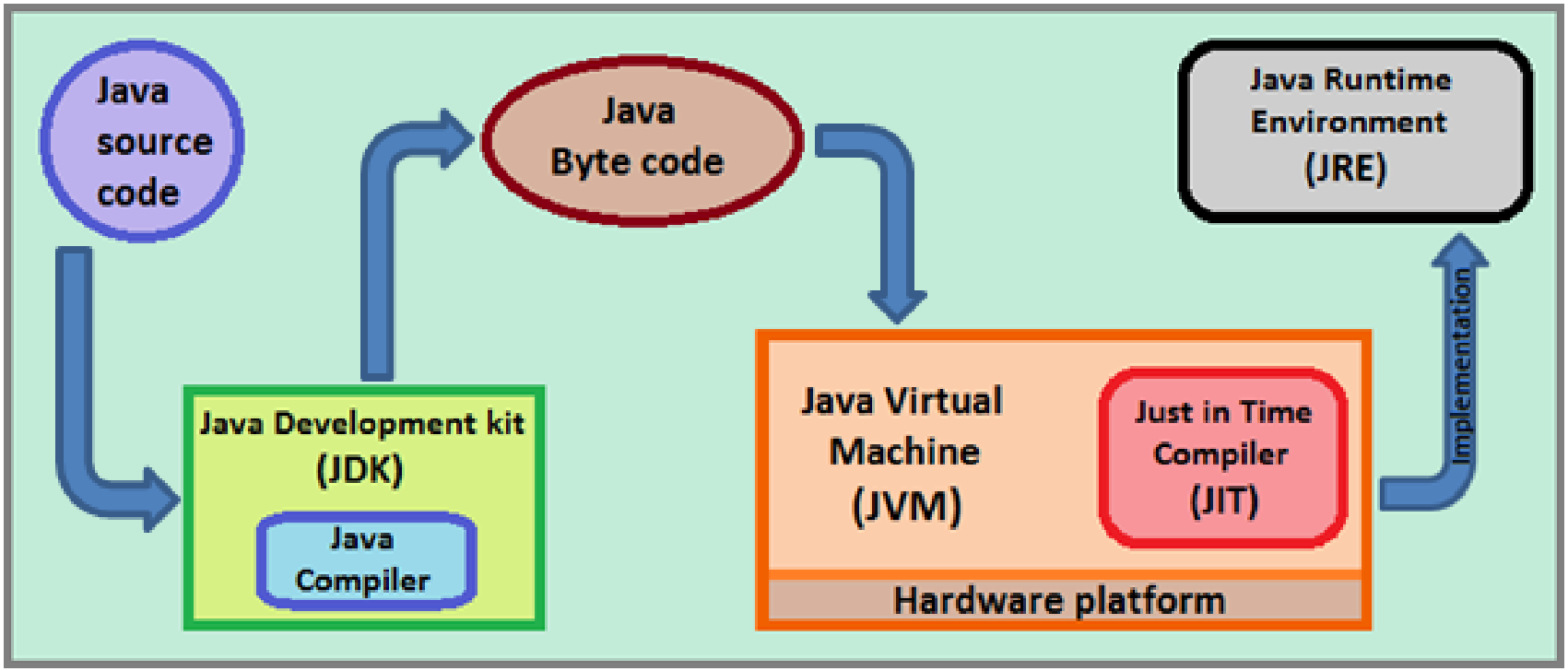
1. It must be "simple, object-oriented, and familiar".
2. It must be "robust and secure".
3. It must be "architecture-neutral and portable".
4. It must execute with "high performance".
5. It must be "interpreted, threaded, and dynamic".

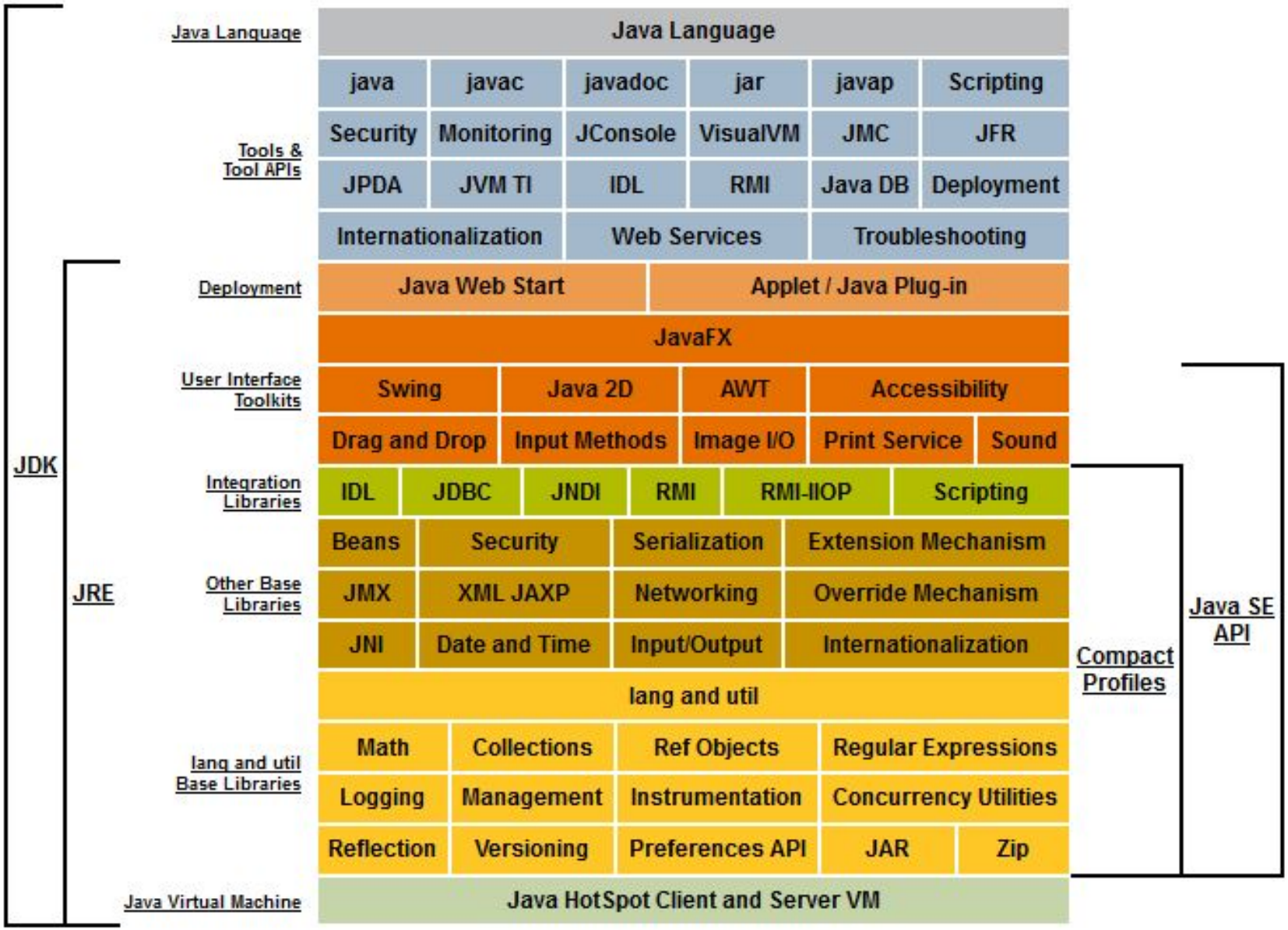
Interpretowany (ang. interpreted) – kod bajtowy Javy (Java bytecode) jest tłumaczony na bieżąco na kod maszynowy danego komputera (interpretowany). Kompilacja kodu występuje tylko raz, natomiast interpretacja zawsze gdy tylko program jest uruchamiany.
Wielowątkowy (ang. multithreaded) – możliwość wykorzystania mechanizmu implementacji wątków i synchronizacji zasobów aby program mógł być wykonywany w tym samym czasie na różnych rdzeniach i procesorach.
Dynamiczny (ang. dynamic) – język Java został zaprojektowany tak, aby adaptować się do rozwijających się środowisk.

	Java	SmalTalk	ICL	Perl	C	C++
prosty (simple)	■	■	■	▒	▒	□
zorientowany obiektowo (object oriented)	■	■	□	■	□	▒
solidny (robust)	■	■	■	■	□	□
bezpieczny (secure)	■	▒	▒	■	□	□
interpretowany (interpreted)	■	■	■	■	□	□
dynamiczny (dynamic)	■	■	■	■	□	□
przenośny (portable)	■	▒	■	■	▒	▒
neutralny (neutral)	■	▒	▒	■	□	□
wątki (threads)	■	□	□	■	□	□
automatyczne zwalnianie pamięci (garbage collection)	■	■	□	□	□	□
wyjątki (exceptions)	■	■	□	■	□	▒
wydajność (performance)	wysoka	średnia	niska	średnia	wysoka	wysoka

Realizacja atrybutu:
- pełna ■
- częściowa ▒
- brak □

JRE, JDK i JVM

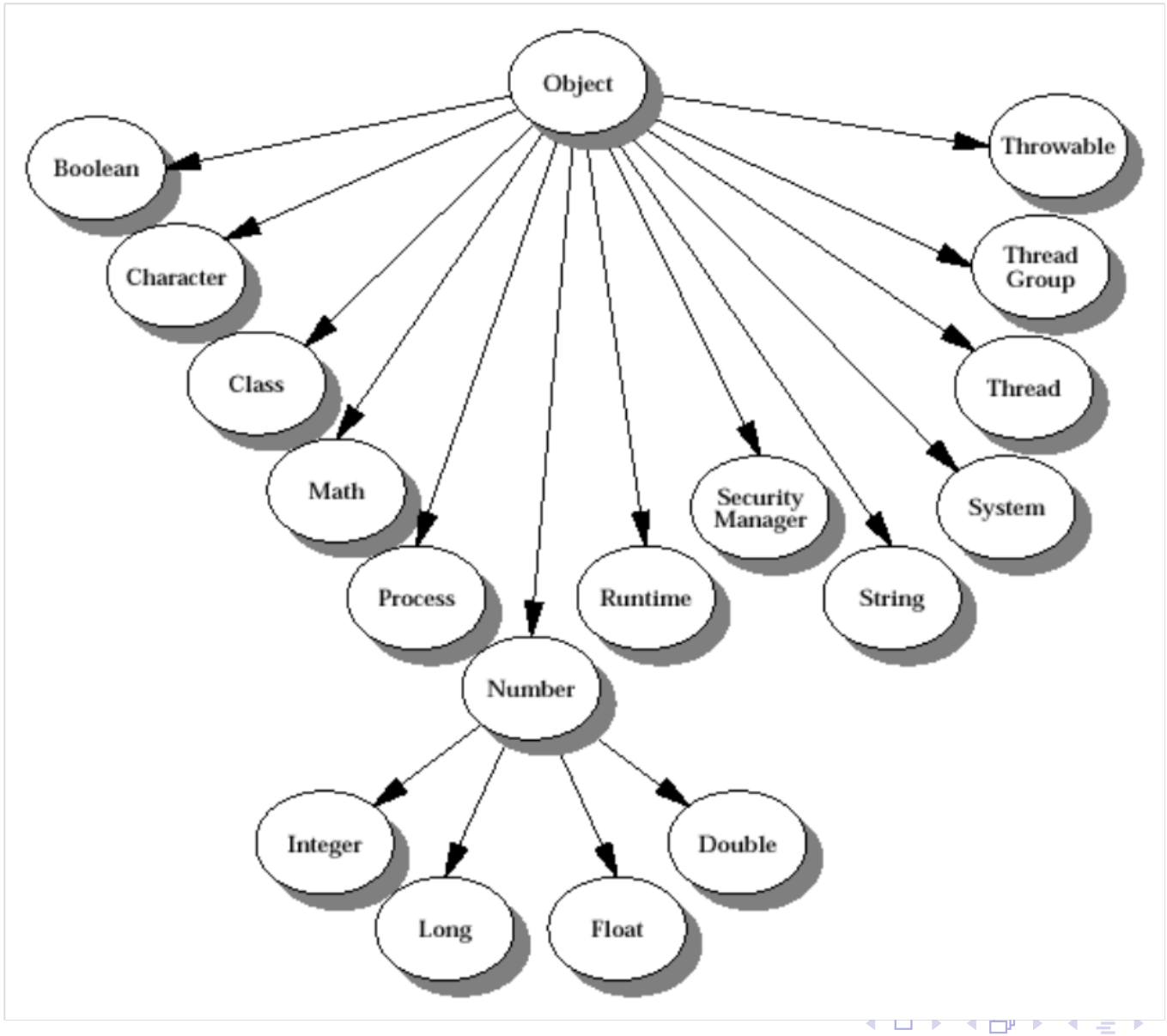

Java source code
Java Byte code
Java Runtime Environment (JRE)
Java Development kit (JDK)
Java Compiler
Java Virtual Machine (JVM)
Just in Time Compiler (JIT)
Hardware platform
Implementation


JDK
JRE
Java Language
Java Language
java
javac
javadoc
jar
javap
Scripting
Tools & Tool APIs
Security
Monitoring
JConsole
VisualVM
JMC
JFR
JPDA
JVM TI
IDL
RMI
Java DB
Deployment
Internationalization
Web Services
Troubleshooting
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface Toolkits
Swing
Java 2D
AWT
Accessibility
Drag and Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
Other Base Libraries
Beans
Security
Serialization
Extension Mechanism
JMX
XML JAXP
Networking
Override Mechanism
JNI
Date and Time
Input/Output
Internationalization
lang and util
lang and util Base Libraries
Math
Collections
Ref Objects
Regular Expressions
Logging
Management
Instrumentation
Concurrency Utilities
Reflection
Versioning
Preferences API
JAR
Zip
Java Virtual Machine
Java HotSpot Client and Server VM
Compact Profiles
Java SE API

Biblioteki Javy

- Podstawowa dystrybucja Javy (J2SE) to prawie cztery tysiące klas zgrupowanych w około dwustu pakietach.
- Najistotniejsze pakiety:
  1. `java.lang`
     Pakiet zawiera zbiór typów bazowych (typów języka) które są zawsze importowane do dowolnego kompilowanego kodu. Tam można znaleźć deklarację Object (korzeń hierarchii klas) i Class, oraz wątków, wyjątków, podstawowych typów danych oraz fundamentalnych klas.

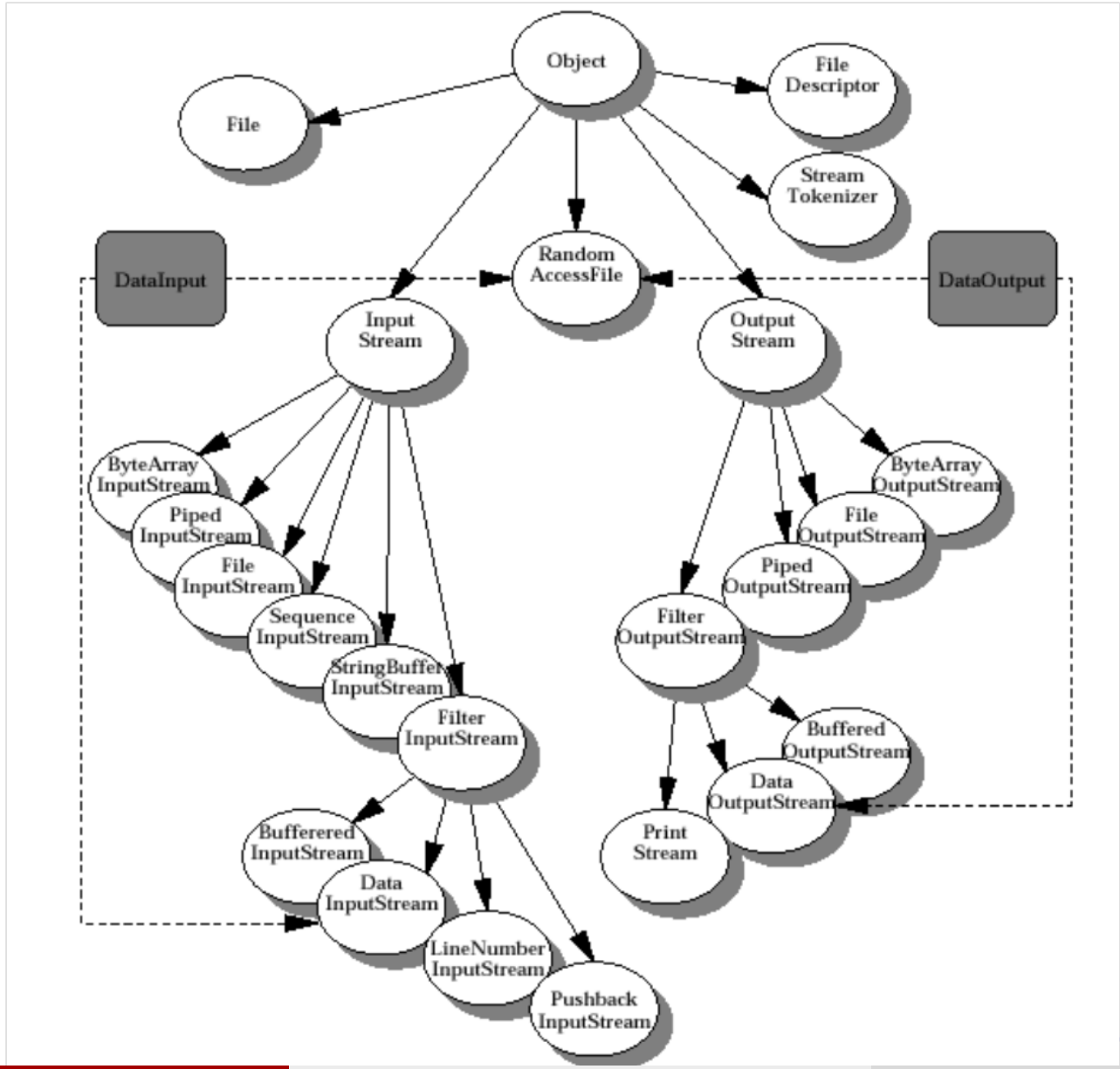
Biblioteki Javy



Biblioteki Javy

- Podstawowa dystrybucja Javy (J2SE) to prawie cztery tysiące klas zgrupowanych w około dwustu pakietach.
- Najistotniejsze pakiety:
  1. `java.lang`
  2. `java.io` i `java.nio`

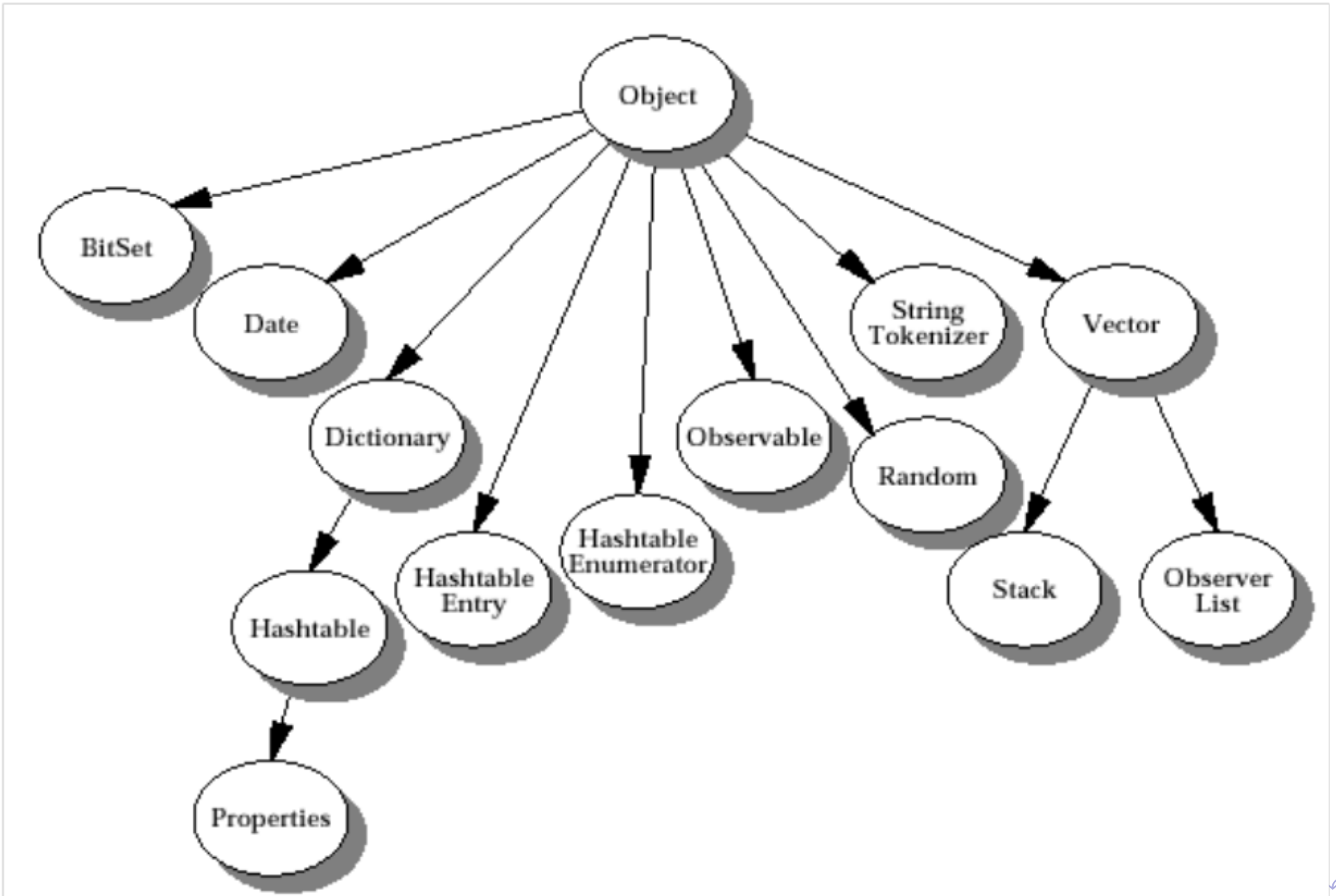
Biblioteki Javy



Biblioteki Javy

- Podstawowa dystrybucja Javy (J2SE) to prawie cztery tysiące klas zgrupowanych w około dwustu pakietach.
- Najistotniejsze pakiety:
  1. `java.lang`
  2. `java.io` i `java.nio`
  3. `java.util`

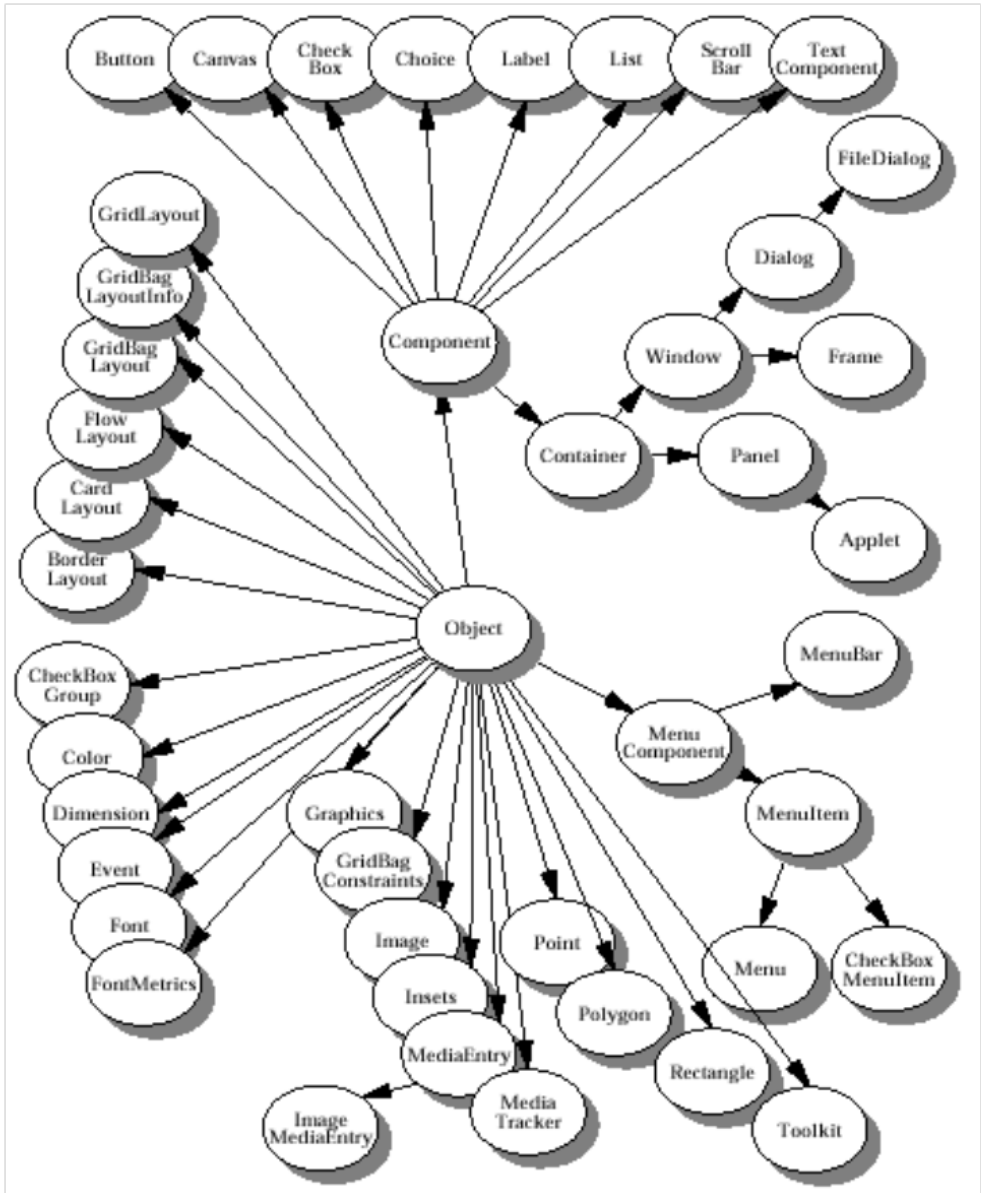
Biblioteki Javy



Biblioteki Javy

- Podstawowa dystrybucja Javy (J2SE) to prawie cztery tysiące klas zgrupowanych w około dwustu pakietach.
- Najistotniejsze pakiety:
  1. `java.lang`
  2. `java.io` i `java.nio`
  3. `java.util`
  4. `java.awt`, `javax.swing`

Biblioteki Javy



Rodzaje programów w Javie

Aplikacje – samodzielne programy uruchamiane na platformie Javy (w tym serwery, tj. aplikacje służące klientom w sieci np. Web serwery, proxy, serwery pocztowe, serwery drukarek itp.)
Applety – programy „doklejane" do stron internetowych i odczytywane za pomocą przeglądarek z zainstalowanymi interpreterami Javy,
Servlety – programy które podobnie jak applety są stworzone dla potrzeb Internetu, ale uruchamiane są po stronie serwera. Mogą być wykorzystane do budowy interaktywnych aplikacji internetowych, zastępując skrypty CGI.
Midlety – programy uruchamiane w urządzeniach PDA, tel. itp.

Słowa kluczowe

abstract

```
abstract class{
  abstract metod();
}
```

Słowa kluczowe

abstract continue for if

```
for (int i=0;i<5;i++){
  if(i%2==0){
    continue;
  }
}
```

Słowa kluczowe

abstract continue for if assert

```
assert ref != null : "ref is null";
if (ref == null)
  throw new AssertionError();
```

przełącznik przy uruchomieniu -enableassertions lub -ea

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally			

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long		

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const				

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	synchronized
boolean	do	goto	private	this
package	double	implements	protected	throw
byte	else	import	public	throws
new	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float			

Słowa kluczowe

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

Słowa kluczowe

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

Słowa kluczowe

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

Operatory

operatory arytm.:	+	++	−	−−	*	/	%
operatory relacyjne:	>	<	=>	=<	==	!=	
operatory logiczne:	!	&&	||	&	|	^	
operatory przypisania:	=	+=	−=	*=	/=	&=	
	|=	^=	<<=	>>=	>>>=	%=	
operatory bitowe:	<<	>>	>>>	~	^		
operatory pozostałe:	? :	.	[]	()			

Priorytet Operatorów

Priorytet	Operatory	Priorytet	Operatory
1	. [] ()	9	^
2	++ -- ! ~ *instanceof*	10	|
3	* / %	11	&&
4	+ -	12	||
5	<< >> >>>	13	? :
6	< > <= >=	14	= op=
7	== !=	15	,
8	&		

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,
- typy całkowitoliczbowe: byte, short, int, long, char,

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,
- typy całkowitoliczbowe: byte, short, int, long, char,
- typy zmiennopozycyjne: float, double

typ	rozmiar (bity)	przedział zmienności
boolean	8	true - false
byte	8	-128 - 127
char	16	Unicode 0-65536
short	16	-32 768 - 32 767
int	32	-2 147 483 648 - 2 147 483 647
long	64	$-9{,}2 \cdot 10^{18}$ - $9{,}2 \cdot 10^{18}$
float	32 IEEE 754	$3{,}4 \cdot 10^{-38}$ - $3{,}4 \cdot 10^{38}$
double	64 IEEE 754	$1{,}7 \cdot 10^{-308}$ - $1{,}7 \cdot 10^{308}$

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,
- typy całkowitoliczbowe: byte, short, int, long, char,
- typy zmiennopozycyjne: float, double

Typy referencyjne dzielą się z kolei na następujące kategorie:

- typy klas,

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,
- typy całkowitoliczbowe: byte, short, int, long, char,
- typy zmiennopozycyjne: float, double

Typy referencyjne dzielą się z kolei na następujące kategorie:

- typy klas,
- typy interfejsów,

Typy danych

Specyficzną cechą Javy jest to, że typy w tym języku są podzielone na dwie kategorie:

- typy pierwotne
- typy referencyjne

Typy pierwotne to grupa ośmiu typów zawierających wartości proste. Tymi typami są:

- typ wartości logicznych: boolean,
- typy całkowitoliczbowe: byte, short, int, long, char,
- typy zmiennopozycyjne: float, double

Typy referencyjne dzielą się z kolei na następujące kategorie:

- typy klas,
- typy interfejsów,
- typy tablic.

Wartościami typów referencyjnych są referencje (w pewnym uproszczeniu można o nich myśleć jako o wskaźnikach) do obiektów lub wartość **null**.

Literały

W Javie mamy 6 rodzajów literałów:

- Liczby całkowite (np. 13 czy -2627). Format dziesiętny, szesnastkowy (0xC) lub ósemkowe (np. 015). Typ literałów całkowitych to `byte`, `short`, `int`, `long`.
- Liczby rzeczywiste (np. 1.0 czy -4.9e12), mogą być zapisanie w systemie dziesiętnym lub szesnastkowym.
- Literały logiczne false i true.
- Literały znakowe (np. 'a').
- Literały napisowe (np. "Ala ma kota"). Na uwagę zasługuje fakt, że napisy nie są w Javie wartościami typu pierwotnego, lecz obiektami klasy String.
- Literał null.

Zmienne

Zmienne są (zwykle) nazwanymi pojemnikami na pojedyncze wartości, typu z jakim zostały zadeklarowane. Zmienne typów pierwotnych przechowują wartości dokładnie tych typów, zmienne typów referencyjnych przechowują wartość null albo wartość referencji do obiektu.
Najczęściej wykorzystywane typy zmiennych:

- zmienne klasowe (statyczne - należące do klasy),
- zmienne egzemplarzowe (należące do obiektu),
- zmienne lokalne (iteratory pętli, zmienne deklarowane w metodach),
- zmienne tablicowe (mogą być anonimowe: `new int[]{ 1, 2, 3 }`),
- parametry metod i konstruktorów,
- parametry obsługi wyjątków.

Każda zmienna musi być zadeklarowana. Z każdą zmienną związany jest jej typ podawany przy deklaracji zmiennej. **Typ ten jest używany przez kompilator do sprawdzania poprawności operacji wykonywanych na zmiennych.**

Tablice

Tablica jest typem umożliwiającym grupowanie zmiennych tego samego typu i odwoływanie się do nich za pomocą wspólnej nazwy.

Tablice można deklarować z podaniem rozmiaru lub bez, np.

```
int month[], week[7];
month = new int[12];
```

Tablice można inicjalizować automatycznie np.

```
int month[]={1,2,3}
```

Tablice

W Javie podobnie jak w C/C++ nie ma tablic wielowymiarowych, a jedynie tablice tablic, np.

```
double macierz[] []= new double [3][3];
```

Tę samą deklarację można przedstawić również na inny sposób, np.

```
double macierz[][]= new double [3][];
macierz[0]=new double [3];
macierz[1]=new double [3];
macierz[2]=new double [3];
```


MAN
WOMAN
WHITE
BLACK
POOR
RICH
ASIAN
PEOPLE THAT
Think arrays
start at 1

Podstawowa obsługa wyjątków

Do obsługi wyjątków służy instrukcja **try-catch**. Po **try** podaje się blok instrukcji, których wyjątki chcemy obsługiwać. Następnie podawana jest lista bloków **catch**, które przypominają deklaracje metod.

```
try {
   //kod ktory moze zglosic wyjatki
} catch (Typ1 w) {
   //obsluga wyjatkow Typ1
} catch (Typ2 w) {
   //obsluga wyjatkow Typ2
} catch (Typ3 w) {
   //obsluga wyjatkow Typ3
}
```

Plan Wykładu

5. Koncepcja programowania obiektowego
6. Enkapsulacja
7. Definiowanie klas,tworzenie obiektów
8. Dziedziczenie
9. Polimorfizm
10. Modyfikatory klas, metod i pól
11. Klasy zagnieżdżone, anonimowe i lokalne

Koncepcja programowania obiektowego

Obiekt jest programowym zestawem powiązanych **zmiennych** i **metod**. Zmienne przechowują informacje dotyczące stanu modelowanych obiektów lub procesów świata rzeczywistego, a metody definiują ich zachowanie.

Programowanie Strukturalne a Obiektowe

Programowanie strukturalne

- Najistotniejszy element to realizowany proces.
- Top-down approach– podejscie do problemu poprzez zdemontowanie go na poszczególne funkcje.
- Realizacja poprzez funkcje.
- Mniej bezpieczne – brak możliwości ukrycia danych.
- Umożliwia realizację średnio skomplikowanych problemów.
- Mniejsze możliwość ponownego użycia kodu, mniejszy poziom abstrakcji i elastyczności.

Programowanie obiektowe

- Najistotniejszy element to dane.
- Bottom-up approach– podejście do problemu poprzez realizację poszczególnych funkcjonalności i połączenie ich rezultatów.
- Realizacja poprzez obiekty.
- Bardziej bezpieczne – pola prywatne.
- Umożliwia realizację dowolnie skomplikowanych problemów.
- Większe możliwość ponownego użycia kodu, większy poziom abstrakcji i elastyczności.

Główne mechanizmy programowania obiektowego

Języki zorientowane obiektowo zawierają następujące mechanizmy wymuszające stosowanie obiektów:

- enkapsulacja (ang. encapsulation),
- dziedziczenie (ang. inheritance),
- polimorfizm (ang. polymorphism).

Mechanizmy te funkcjonują w Javie bardzo podobnie jak w C++.

Enkapsulacja

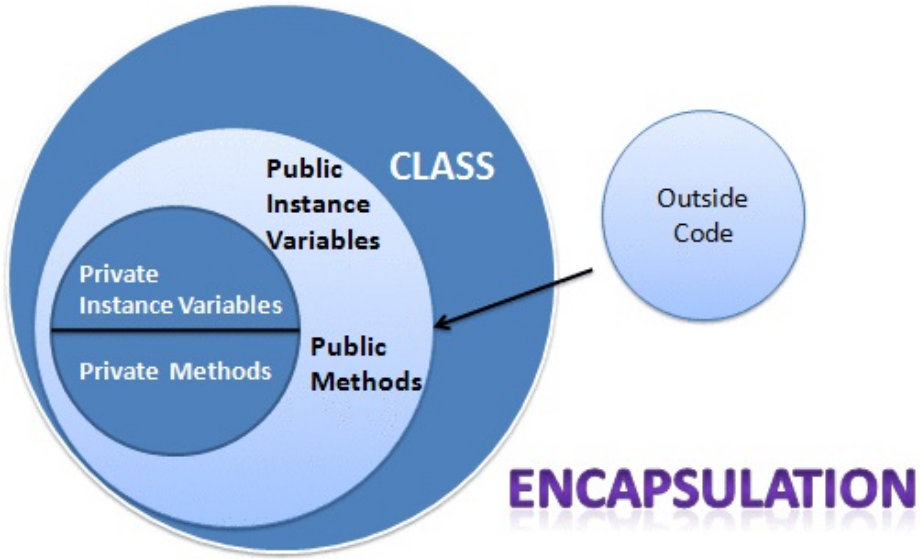
Enkapsulacja polega na łączeniu danych i instrukcji, które wykonują na nich działania, przez umieszczanie ich we wspólnych obiektach. Środkiem do osiągnięcia enkapsulacji w Javie sa klasy (ang. *classes*).

Klasa stanowi model abstrakcyjny pewnej grupy obiektów wyróżniających się ta sama struktura i zachowaniem, jest modułem posiadającym nazwę i atrybuty w postaci pól danych i metod. Zatem **obiekt** (ang. *object*) to pojedynczy egzemplarz klasy.

Obiekty nazywa się nieraz **egzemplarzami klas** (ang. *instances of classes*). Dane należące do klasy i przechowujące informacje o stanie każdego obiektu określa się **zmiennymi egzemplarzowymi** (ang. *instance variables*). Wykonywane zadania i sposób dostępu do danej klasy określają jej **metody**.

Celem enkapsulacji jest zmniejszenie stopnia złożoności programu poprzez możliwość ukrycia szczegółów dotyczących funkcjonowania klasy przez zadeklarowanie ich jako **prywatne**. Natomiast elementy tworzące interfejs klasy deklaruje się jako **publiczne**.
Definicja klasy jest jedynym sposobem zdefiniowania nowego typu danych w Javie. Posługując się pojęciami klasy, programista może w wygodny i elegancki sposób definiować różnorodne typy danych wykorzystując:

- strukturę hierarchiczna deklaracji klas (kompozycja),
- prefiksowanie klas tzw. "dziedziczenie", umożliwiające tworzenie hierarchii typu: ogólny - bardziej szczegółowy.



Definicja Klasy

Definicja klasy przyjmuje następującą formę (modyfikatory zostaną omówione później):

```
[modyfikatory] class NazwaKlasy [ extends NazwaNadklasy]
[implements NazwyInterfejsow] {
Cialo klasy:
Tutaj znajduja sie definicje pol danych,
metod i klas wewnetrznych klasy
}
```

Elementy deklaracji pomiędzy nawiasami [i] sa opcjonalne. Stąd najprostsza postać definicji to:

```
class A{
}
```

Operator new i konstruktory

Deklaracja zmiennej nie powoduje utworzenia obiektu - jest on tworzony dopiero za pomocą operatora `new` i konstruktora. Do tego czasu referencja jest pusta (ang. *null reference*).

Operator `new` tworzy pojedynczy egzemplarz danej klasy i zwraca wartość odwołania do niego.

```
Punkt p=new Punkt();
Punkt p2=p;
```

Zapisanie wartości zmiennej p w p2 nie powoduje zarezerwowania dodatkowej pamięci lub przekopiowania jakiejkolwiek części obiektu wskazywanego przez p. Istnieje tylko jeden obiekt i dwa odwołania.

Konstruktory

Konstruktor zostaje wywołany podczas tworzenia nowego obiektu klasy. Każda klasa może posiadać wiele konstruktorów, różniących się lista argumentów. Ponieważ każda klasa w Javie dziedziczy z klasy `Object`, posiada tez konstruktor bezparametrowy odziedziczony z tej klasy.

```
class MojaKlasa {
        MojaKlasa(){ //pierwszy konstruktor
                System.out.print("Konstruktor");
        }
        MojaKlasa(int a){...} //drugi konstruktor
}

...

MojaKlasa zmienna=new MojaKlasa();
```

Konstruktor kopiujący

W Javie nie używa się konstruktora kopiującego niejawnie i nie jest on tak często wykorzystywany jak w C++.

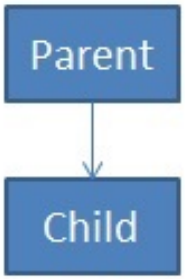
```
class MojaKlasa {
        String dane;
        MojaKlasa(){//...jakies cialo
        }
        MojaKlasa(MojaKlasa x){
                dane=new String(x.dane);
        }

...

MojaKlasa a=new MojaKlasa();
MojaKlasa zmienna=new MojaKlasa(a); //wywolanie konstruktora
                                    //kopiujacego
```

Dziedziczenie

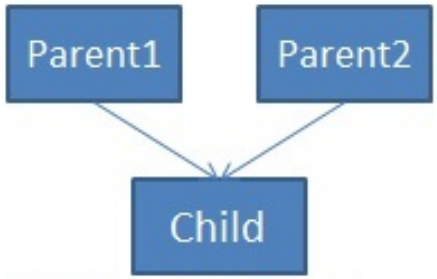
Większość ludzi czuje potrzebę uporządkowania obiektów świata rzeczywistego poprzez tworzenie złożonych taksonomii (klasyfikacji, kategoryzacji). W przypadku gdy pewne klasy tworzą abstrakcyjne, wspólne modele opisu (np. klasa Zwierzęta), wówczas na ich podstawie można utworzyć podklasy (np. klasa Ssaki) będące w relacji zawierania się w klasach nadrzędnych. Opis ssaków zawierać może dodatkowe informacje dotyczące ich charakterystycznych cech.

Ponieważ ssaki sa dokładniej opisanymi zwierzętami to dziedziczą wszystkie ich atrybuty. Klasę zwierząt nazwiemy nadklasa (ang. *superclass*), natomiast klasę ssaków podklasa (ang. *subclass*).

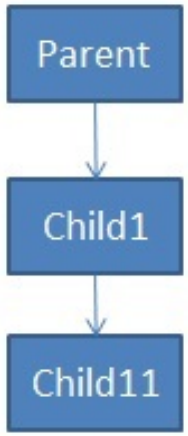
Types of Inheritance



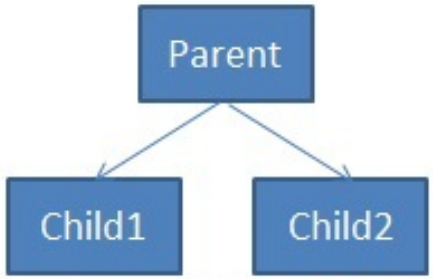
1. Single Inheritance



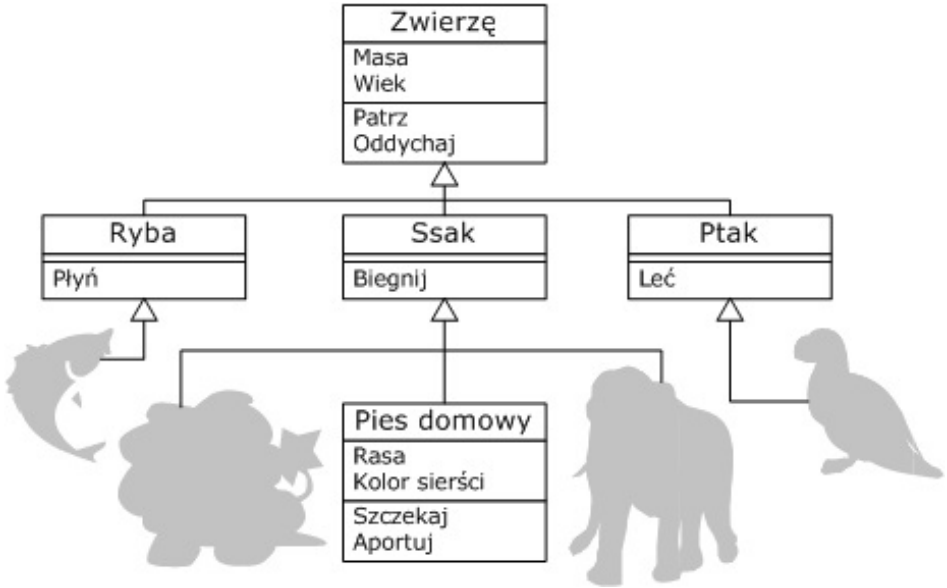
2. Multiple Inheritance



3. Multi-Level Inheritance



4. Hierarchical Inheritance



Polimorfizm

Aby wykonać dwa różne zadania w większości języków programowania trzeba utworzyć dwie funkcje o różnych nazwach. **Polimorfizm** to możliwość tworzenia wielu metod o takiej samej nazwie, zgodnie z zasada: *jeden przedmiot, wiele kształtów*.

Polimorfizm pozwala tworzyć wiele implementacji tej samej metody, co w informatyce nazywa się jej przeładowywaniem (ang. *overloading*). Wybór implementacji metody jest zależny od przekazywanych jej parametrów.

```
public class StatycznyPolimorfizm {
        void metoda(){

        }
        void metoda(double y){

        }
        int metoda(int x){
                return x;
        }
/*      int metoda(){ //blad!!!

        }
*/
}
```

Polimorfizm czasu przebiegu

Przeładowywanie metod umożliwia ujednolicenie działań wykonywanych na danych różnych typów i jest najbardziej przydatne podczas tworzenia małych klas, gdyż ma charakter statyczny (trzeba przewidzieć wszystkie typy danych na których trzeba będzie wykonywać działania).

Czasami potrzebne jest rozwiązanie przewidujące możliwość zmiany implementacji danej metody w podklasie. Wówczas jej wersja zdefiniowana w nad klasie może w ogóle nie wykonywać żadnego działania. Tego typu polimorfizm nazywa się polimorfizmem czasu przebiegu (ang. *runtime polymorphism*).

```
class A{
        void metoda(){
                system.out.println("to metoda A");
        }
}
class B extends A{
        void metoda(){
                system.out.println("to metoda B");
        }
}
class C extends A{
        void metoda(){
                system.out.println("to metoda C");
        }
}
class Aplikacja{
        public static void main(String [] args){
                A obj=null;
                double val=Math.random();
                if (val<0.5)     obj=new B();
                else                    obj=new C();
                obj.metoda();
}}
```

Odwołania do obiektów

Deklaracja zmiennej typu obiektowego przyjmuje postać podobna do deklaracji zmiennej typu pierwotnego. Aczkolwiek w odróżnieniu od takiej zmiennej jej inicjalizacja musi zostać wykonana z wykorzystaniem operatora **new** i określonego konstruktora lub metod tworzących dany obiekt (wzorzec projektowy fabryka – factory). Utworzenie obiektu może nastąpić równocześnie z deklaracją zmiennej obiektowej:

```
KontoOsobiste konto=new KontoOsobiste();
```

lub w późniejszej części kodu:

```
KontoOsobiste konto;
...
konto=new KontoOsobiste();
```

Odwołania do obiektów przypominają wskaźniki z C/C++, z ta różnica że nie można na nich wykonywać operacji arytmetycznych.
Bezpieczeństwo Javy wynika właśnie z braku możliwości utworzenia odwołania, które by wskazywało na dowolny fragment pamięci.
Odwołania do obiektów danej klasy sa kompatybilne z odwołaniami do obiektów wszystkich jej podklas.
Ponadto, wszystkie typy wewnętrzne maja swoje odpowiedniki obiektowe (np.: typ `int` ma odpowiadajacy mu typ obiektowy `Integer`) i jako obiekty maja wiele konstruktorów i metod.

Pola danych i metody

- **Pola Danych** - są atrybutami klasy, pełniącymi role podobna do zmiennych lub stałych. Sa one deklarowane na tych samych zasadach, co zmienne lokalne, wg. schematu:
  `modyfikatoryPola TypPola NazwaPola;`
- **Metody** – są modułami programowymi przypominajacymi funkcje z jezyka C++. Kazda funkcja w Javie jest zwiazana z definicja klasy (spełnia role jej metody) i deklarowana jest wg. schematu:

```
modyfikatory TypRezultatu NazwaMetody(ListaParametrów)
{
//tresc metody
}
```

  Metody sa modułami programowymi przypominającymi funkcje z języka C++. Każda funkcja w Javie jest związana z definicja klasy (spełnia role jej metody)

Pola danych i metody – przykład

```
public class Punkt {
        public int x = 0;
        public int y = 0;

        public ustaw(int x, int y){
                this.x = x;
                this.y = y;
        }
}
```

Modyfikatory klas, metod i pól

W Javie modyfikatory możemy podzielić na dwa rodzaje:

- modyfikatory dostępu wpływające na reguły widoczności i umożliwiające kontrole dostepu do pól danych i metod klasy z innych klas,
- modyfikatory właściwości modyfikowanego elementu.

Modyfikatory dostepu

Modyfikatory dostepu:

- public - wszystkie klasy maja dostęp do pól danych i metod public,
- private - dostęp do metod i pól danych posiadają jedynie inne metody tej samej klasy,
- protected - metoda lub pole danych protected lub może być używana jedynie przez metody swojej klasy oraz metody wszystkich jej klas pochodnych,
- package - jest to **modyfikator domyślny**, wszystkie metody i pola danych bez modyfikatora dostępu traktowane sa jako typu package. Metody (lub pola danych) typu package mogą być używane przez inne klasy danego pakietu.

Poziomy dostępu

modyfikator	klasa	podklasa	pakiet	wszędzie
private	X			
package	X	*	X	
protected	X	X	X	
public	X	X	X	X

* - dostęp istnieje o ile klasa i podklasa należą do tego samego pakietu.

Gettery i settery

- Gettery i settery to publiczne metody, które pozwalają odpowiednio pobierać i zapisywać wartości prywatnych pól klasy.
- Można je traktować jak swego rodzaju interfejs do obsługi funkcjonalności jaką oferuje dana klasa.
- Pozwalają na ukrycie operacji, których sposób działania jest nie istotny z perspektywy użycia danej klasy.

Gettery i settery – przykład

```
public class A {
        private int x;

        public int getX() {
                return x;
        }

        public void setX(int x) {
                this.x = x;
        }
}
public class B extends A {
        void metoda () {
                this.setX(10);
        }
}
public class C {
        A obj=new A();
        void metoda() {
                obj.setX(10);
        }
}
```

Modyfikatory właściwości

- pól danych
  - static,
  - final,
  - transient,
  - volatile.
- klas
  - public,
  - final,
  - abstract,
- metod
  - abstract,
  - static,
  - final,
  - synchronized,
  - native;

Pola i metody statyczne

Deklaracja ze słowem **static** oznacza, ze pole danych lub metoda dotyczy klasy, a nie obiektu, tzn. dla wszystkich obiektów danej klasy pole statyczne ma te sama wartość.

Metody statyczne podobnie jak statyczne pola danych są przypisane do klasy, a nie konkretnego obiektu i służą do operacji tylko na polach statycznych. Przykład. Mamy klasę opisującą rachunek bankowy. Dla wszystkich rachunków jednego typu oprocentowanie wkładów jest jednakowe, wiec oprocentowanie rachunku zadeklarowane zostało jako pole statyczne aby w razie zmiany oprocentowania nie zmieniać jego wartości dla wszystkich obiektów tej klasy.

```
class KontoOsobiste{
  static byte oprocentowanie = 10;
  private String wlasciciel;
  private String saldo;
  static void ZmienOprocentowanie(byte nowyProcent) {
    oprocentowanie = nowyProcent;
    }
  public void doliczProcent{
      saldo+=saldo*oprocentowanie/100;
  }

  public static void main (String[] args){
    KontoOsobiste Kowalski
      =new KontoOsobiste("Grazyna Kowalska",500);
    KontoOsobiste Nosacz
      =new KontoOsobiste("Janusz Nosacz",1500);
    Kowalski.doliczProcent();
    KontoOsobiste.ZmienOprocentowanie(15);
    Nosacz.doliczProcent();
  }
}
```

Inicjowanie pól statycznych

Do inicjalizacji zmiennych statycznych wykorzystuje się zamiast konstruktorów tzw. inicjatory zmiennych statycznych. Inicjacja następuje wtedy, gdy klasa jest pierwszy raz ładowana do pamięci (gdy nie ma jeszcze żadnego obiektu danej klasy).
Każda klasa może zawierać dowolna liczbę inicjatorów statycznych.
Inicjatory statyczne wykonywane sa w kolejności ich wystąpienia w definicji klasy.

```
class MojaKlasa {
  static int licznik=0;
  static {
    System.out.println("inicjalizacja␣pol␣statycznych")
    licznik=0;
  }
}
```

Klasy zagnieżdżone (ang. *nested*)

Klasa zagnieżdżona jest członkiem innej klasy. Klasy zagnieżdżone, mogą mieć modyfikator static , wskazujący ze maja takie same właściwości jak klasa zewnętrzna. Jeśli nie sa statyczne, wówczas określa się je jako wewnętrzne. Oprócz tego, klasy zagnieżdżone mogą być oczywiście opatrzone modyfikatorami: private, protected i public oraz abstract i final - a znaczenia tych modyfikatorów sa takie same jak w przypadku zwykłych klas.

```
class KlasaZewnetrzna{
  static class KlasaZagnizdzonaStatyczna {
    . . .
  }
  class KlasaWewnetrzna {
    . . .
  }
}
```

Klasy wewnętrzne a statyczne zagnieżdżone

Klasy zagnieżdżone statyczne:

- nie mogą bezpośrednio odwoływać się do atrybutów klasy zewnętrznej (musza uzywac kwalifikowanego odnośnika)
- obiekty tych klas mogą istnieć nawet gdy nie istnieją obiekty klas zewnętrznych,
- mogą być traktowane jak klasy zewnętrzne ale ze specjalizacją funkcjonalności dla konkretnej klasy w której są zagnieżdzone. Usprawniają proces enkapsulacji poprzez powiązanie metod operujących na danych klasach z tymi klasami.

Klasy wewnętrzne:

- maja nieograniczony dostęp do atrybutów klasy zewnętrznej,
- obiekty tych klas istnieją tylko gdy istnieją obiekty klas zewnętrznych,
- sa często wykorzystywane w mechanizmie obsługi zdarzeń AWT

```
public class OuterClass {
        private int val;
        private static int staticVal;
        public class InnerClass {
                void method() {
                        val = 2;
                }
        }
        static class StaticClass {
                void methodInStaticClass() {
                        staticVal = 5;
                }
        }
        public static void main(String[] args) {
                StaticClass myStaticObj=new StaticClass();
                myStaticObj.methodInStaticClass();
                OuterClass myObj=new OuterClass();
                InnerClass myInnerObj= myObj.new InnerClass();
                myInnerObj.method();
                System.out.println(myObj.val);

        }
}
```

Klasy wewnetrzne - przykład uchwytu

Załóżmy, że do istniejącej klasy kontenerowej `Stack` chcemy dodać nową funkcjonalność – pozwolić innym klasom na zliczanie elementów na stosie wykorzystując interfejs `java.util.Enumeration`.

Interfejs zawiera dwie deklaracje metod:

```
public boolean hasMoreElements();
public Object nextElement();
```

i definiuje interfejs dla pętli po elementach:

```
while (hasMoreElements()) nextElement()
```

Jeżeli `Stack` zaimplementuje `Enumeration` w sobie, nie będzie można zliczyć zawartości stosu więcej niż raz (zostanie wyczyszczony), jak również zastosować dwóch wyliczeń równolegle. Musi istnieć klasa pomocnicza (adapter), w tym przypadku wewnętrzna, która ma dostęp do wszystkich elementów ponieważ klasa `Stack` wspiera tylko kolejki LIFO.

```
public class Stack {
  private Vector<Integer> items = new Vector<Integer>();
  public Stack() {
    items.add(5); items.add(6); items.add(7); items.add(8);
  }
  public Enumeration<Integer> enumerator() {
    return new StackEnum();
  }

  class StackEnum implements Enumeration<Integer> {
    int currentItem = items.size() - 1;

    public boolean hasMoreElements() {
                        return (currentItem >= 0);
    }

    public Integer nextElement() {
      if (!hasMoreElements())
        throw new NoSuchElementException();
      else
        return items.elementAt(currentItem--);
    }
  }
```

```
  public static void main(String[] args) {
    Stack obj=new Stack();
    for (Enumeration e =obj.enumerator();
    e.hasMoreElements();){
      int liczba= (int) e.nextElement();
      System.out.println(liczba);
      for (Enumeration e2 =obj.enumerator();
      e2.hasMoreElements();){
        int liczba2= (int) e2.nextElement();
        System.out.println(liczba2);
      }
    }
  }
}
```

Klasy anonimowe

Klasa anonimowa jest to klasa wewnętrzna zadeklarowana bez nazwy i konstruktora (tylko przy użyciu `new`). Ze względu na nieczytelność kodu jest stosowana rzadko.

```
public class AnonKlassExample {
        int x;
        void metoda() {
                x = (new Object() {
                        int obliczInt() {
                                return 5;
                        }
                }).obliczInt();
        }
}
```

Możliwe jest utworzenie klasy anonimowej implementującej określony interfejs lub dziedziczącej po określonej klasie (w obu przypadkach klasa ta nie zostaje nazwana i możliwe jest utworzenie tylko jednej instancji tego typu).

Klasy lokalne

Klasy lokalne to klasy zdefiniowane w bloku programu Javy. Klasa taka może odwoływać się do wszystkich zmiennych widocznych w miejscu wystąpienia jej definicji. Klasa lokalna jest widoczna, i może zostać użyta, tylko w bloku w którym została zdefiniowana.

```
class Test {
  void test()
  {
    // definicja klasy lokalnej
    class KlasaLokalna{
        ...
    }
    // deklaracja obiektu typu: KlasaLokalna
    KlasaLokalna kl = new KlasaLokalna();
  }
}
```

Klasy i metody abstarkcyjne

Niekiedy definiujemy klase reprezentujaca jakas abstrakcyjna koncepcje, opisujaca pewne własnosci wspólne dla reprezentowanej abstrakcji. Przykładem takiej koncepcji abstrakcyjnej niech bedzie pojecie: "figura". Tworzenie obiektu klasy figura nie ma sensu, ze wzgledu na jego abstrakcyjnosc, ale juz podklasy figury, np. kwadrat, trójkat czy koło moga byc instancjowane.

Klasa abstrakcyjna moze zawierac metody abstrakcyjne, które nie zawieraja implementacji. W ten sposób klasa abstrakcyjna definiuje interfejs programistyczny, który musi znalezc sie w nie-abstrakcyjnych jej podklasach.

Klasy abstrakcyjne - przykład

```
abstract class GraphicObject {
  int x, y;
  . . .
  void moveTo(int newX, int newY) { //metoda zwykla
    x=newX;y=newY;
  }
  abstract void draw(); // bez implementacji
                        // metoda abstrakcyjna
}
class Circle extends GraphicObject {
  void draw() {
    . . . //wymaga implementacji
  }
}
class Rectangle extends GraphicObject {
  void draw() {
    . . . //wymaga implementacji
  }
}
```

Klasy i metody abstrakcyjne-własciwosci

- Nie jest wymagane, aby klasa abstrakcyjna zawierała metody abstrakcyjne. Jednakze kazda klasa, która ma metode abstrakcyjna, lub która nie implementuje metod abstrakcyjnych dziedziczonych z nadklasy, musi być zadeklarowana jako klasa abstrakcyjna.
- Nie wolno deklarowac abstrakcyjnych konstruktorów oraz abstrakcyjnych metod statycznych, choc klasa abstrakcyjna moze zawierac ich nie-abstrakcyjne wersje.
- Kazda podklasa klasy abstrakcyjnej **musi** implementowac wszystkie metody abstrakcyjne nadklasy, chyba ze sama została zadeklarowana jako abstrakcyjna.

Interfejsy

Interfejs (ang. interface) jest "urzadzeniem" które pozwala róznym obiektom współpracowac ze soba (analogicznym do protokołu). Słowo kluczowe **interface** definiuje kolekcje definicji metod oraz wartosci stałych, które moga byc pózniej wykorzystane przez inne klasy, przy uzyciu słowa **implements**. Interfejs klasy definiuje jej protokół zachowań (dostępu do klasy), który może być zaimplementowany przez dowolną klasę w dowolnym miejscu w hierarchii klas. Interfejsy sa użyteczne do

- wychwytywania podobieństw miedzy nie powiązanymi klasami
- deklarowania metod, które maja być zaimplementowane w jednej lub większej liczbie klas
- odsłaniania interfejsu programistycznego obiektu bez ujawniania jego klasy

Interfejsy a klasa abstrakcyjna

Interfejs jako kolekcja publicznych metod abstrakcyjnych (bez implementacji), różni się od klasy abstrakcyjnej (KA) w sposób następujący:

- z zasady nie implementuje żadnych metod (KA może, w javie 1.8 umożliwiono na domyślną implementację w interfejsie)
- nie jest czescia hierarchii klas (KA jest) - nie powiązane klasy moga miec takie same interfejsy,
- klasa moze miec wiele interfejsów, ale tylko jedna nadklase - interfejs moze dziedziczyc z innych interfejsów, ale nie moze dziedziczyc z klas,
- jesli istnieja w interfejsie pola danych, to sa one domyslnie publiczne, finalne i statyczne (KA-nie tylko takie),
- klasa, która implementuje interfejs, musi posiadac definicje wszystkich metod zadeklarowanych w interfejsie (KA -tylko abstrakcyjnych), w przeciwnym wypadku klasa taka bedzie klasa abstrakcyjna.

Definiowanie interfejsu

```
[modyfikator] interface NazwaInterfejsu [extends
listaInterfejsów] {
członkowie klasy nie moga byc: volatile, synchronized,
transient, private, protected . . . }
```

```
interface Kolekcja {
  int MAXIMUM = 200;
  void dodaj(Object obj);
  Object znajdz(Object obj);
  int liczbaObiektow();
}
```

```
class Wektor implements Kolekcja {
  private Object obiekty[] = new Object[MAXIMUM];
  private short m_sLicznik = 0;
  public void dodaj(Object obj)
    {
      obiekty[m_sLicznik++]=obj;
    }
  public Object znajdz(Object obj) {
    for (int i = 0; i<m_sLicznik;i++)
    {
      if (obiekty[i].getClass() == obj.getClass()) {
        System.out.println("Znaleziono␣obiekt␣klasy␣"
        + obj.getClass() );
        return obiekty[i];
      }
    }
    return null;
  }
  public int liczbaObiektow(){
    return m_sLicznik;
  }
}
```

Interfejs jako typ

```
class Test {
  public void funkcja_testowa(Kolekcja kolekcja, int delta) {
    //... cialo funkcji
  }
  public static void main(String args[]) {
    Kolekcja zmienna = new Wektor();
    zmienna.liczbaObiektow(); // dynamiczne wywolanie
  }
}
```

Metody sa wybierane dynamicznie w czasie przebiegu. Poniewaz odszukiwanie metod w czasie działania programu wpływa negatywnie na szybkosc jego działania, to w miejscach, w których ma ona decydujace znaczenie, nalezy unikac deklarowania zmiennych jako odwołan do interfejsów.

Rozrastanie interfejsów

Ze wzgledu na funkcjonalnosc tworzonych bibliotek, interfejsy nie powinny sie rozrastac! Dodanie nowych metod do interfejsu Kolekcja spowodowałoby koniecznosc przedefiniowania wszystkich klas po nim dziedziczacych (inaczej byłyby abstrakcyjne). Inni programisci przestaliby uzywac takich bibliotek - zamiast tego lepiej zadeklarowac podinterfejs.

```
public interface NowaKolekcja extends Kolekcja {
  void currentValue(String tickerSymbol, double newValue);
}
```

Interfejs - domyślna implementacja

Od JAvy 8 istnieje możliwość zdefiniowania tak zwanych metod domyślnych interfejsu. Metody te mogą mieć właściwą implementacje w ciele interfejsu. Metody takie poprzedzone są słowem kluczowym **default** jak w przykładzie poniżej:

```
public interface MicrowaveOven {
    void start();

    void setDuration(int durationInSeconds);

    boolean isFinished();

    void setPower(int power);

    default String getName() {
        return "MicrovaweOwen";
    }
}
```

Przykrywanie metod

```
class Punkt{
  int x; int y;
  public void Zeruj(){
      X=0;Y=0;
      System.out.println("Zerowanie Punktu");
    }
}
class Punkt3D extends Punkt {
  int z;
  public void Zeruj(){
    super.Zeruj(); z=0; System.out.println("Zerowanie Punktu3D
  }
}
// ----------wywolanie metod-----------------------------
Punkt a=new Punkt(); Punkt3D b=new Punkt3D();
a=b; //mozna przypisac - odwrotne przypisanie byloby bledne
a.Zeruj(); //wywoluje metode dla klasy 3D
```

Przykrywanie metod - wielokrotne dziedziczenie

```
class Raz {
  String m_SNazwa= "Raz"; //zmienna
  String s() {  //funkcja
    return "1";
  }
}
class Dwa extends Raz {
  String m_SNazwa= "Dwa";
  String s() {
    return "2";
  }
}
class Trzy extends Dwa {
  String m_SNazwa= "Trzy";
  String s() {
    return "3";
  }
}
```

Odwołanie typu `super.super.NazwaMetody()` przy wielokrotnym dziedz.

```
  void test(){
    System.out.println("s()=" + s());
    ...println("m_SNazwa=" + m_SNazwa);
    ...println(".........");
    ...println("super.s()=" + super.s());
    ...println("super.m_SNazwa=" + super.m_SNazwa);
    ...println(".........");
    ...println("((Dwa)this).s()=" + ((Dwa)this).s());
    ...println("((Dwa)this).m_SNazwa=" + ((Dwa)this).m_SNazwa);
    ...println(".........");
    ...println("((Raz)this).s()=" + ((Raz)this).s());
    ...println("((Raz)this).m_SNazwa=" + ((Raz)this).m_SNazwa);
}
```

Wywołanie:

```
Trzy trzy = new Trzy();
trzy.test();
```

spowoduje wyswietlenie:

s()= 3

m SNazwa= Trzy

.....

super.s()= 2

super.m SNazwa= Dwa

.....

((Dwa)this).s()= 3

((Dwa)this).m SNazwa= Dwa

.....

((Raz)this).s()= 3

((Raz)this).m SNazwa= Raz

Przykrywanie metod- wielokrotne dziedziczenie

Dla pól danych uzycie słowa kluczowego super lub konwersji do klasy Raz lub Dwa powoduje wypisanie na ekranie wartosci pola `m_SNazwa` z odpowiedniej klasy. Natomiast dla metod, konwersja typu `((Dwa)this).s()` jest równowazna wywołaniu `this.s()`. Dzieje sie tak dlatego, ze w Javie kazda metoda, która nie jest statyczna (static) lub prywatna (private) jest wirtualna (ang. virtual). Dlatego wywołanie metody `s()` klasy `Raz`: `(((Raz)this).s())` jest przekształcane w wywołanie przedefiniowujacej ja metody `s()` z klasy `Trzy`. Te właściwość nazywa się dynamicznym rozdzielaniem metod.

Usuwanie obiektów

Java nie wymaga definiowania destruktorów, gdyz istnieje mechanizm automatycznego zarzadzania pamiecia (ang. garbage collection). Obiekt istnieje w pamieci dopóki istnieje do niego jakakolwiek referencja w programie, w tym sensie, ze gdy referencja do obiektu nie jest juz przechowywana przez zadna zmienna obiekt jest automatycznie usuwany, a zajmowana przez niego pamiec zwalniana. Proces zbierania nieuzytków jest właczany okresowo, uwalniajac pamiec zajmowana przez obiekty, które nie sa juz potrzebne. W czasie działania programu przegladany jest obszar pamieci przydzielanej dynamicznie, zaznaczane sa obiekty, do których istnieja referencje. Po przesledzeniu wszystkich mozliwych sciezek referencji do obiektów, te obiekty, które nie sa zaznaczone (tzn. do których nie ma referencji) zostaja usuniete.

Usuwanie obiektów

Program w Javie moze jawnie uruchomic mechanizm zbierania nieuzytków poprzez wywołanie metody System.gc() . Mechanizm oczyszczania pamieci z nieuzytków działa w watku o niskim priorytecie synchronicznie lub asynchronicznie, zaleznie od sytuacji i srodowiska systemu operacyjnego na którym wykonywany jest program w Javie. W systemach, które pozwalaja srodowisku przetwarzania Javy sprawdzac, kiedy watek sie rozpoczał i przerwał wykonanie innego watku (takich jak np. Windows 2000), mechanizm czyszczenia pamieci działa asynchronicznie w czasie bezczynnosci systemu. Mimo ze Java nie wymaga destruktorów, istnieje mozliwosc deklaracji specjalnej metody finalize (zdefiniowanej w klasie `java.lang.Object`), która bedzie wykonywana przed usunieciem obiektu z pamieci. Deklaracja takiej metody ma zastosowanie, gdy nasz obiekt np.: ma referencje do urzadzen wejscia-wyjscia i przed usunieciem obiektu nalezy je zamknac.

Usuwanie obiektów - Finalizacja

```
class OtworzPlik {
  FileInputStream m_plik = null;
  OtworzPlik(String nazwaPliku){ //Otwarcie pliku
    try{
      m_plik = new FileInputStream(nazwaPliku);
    }
    //Obsluga wyjatku
    catch (java.io.FileNotFoundException e) {
      System.err.println("Nie moge otworzyc pliku" + nazwaPliku
    }
  }
  protected void finalize () throws Throwable
  {
    if (m_plik != null)
    {
      m_plik.close();
      m_plik = null;
    }
  }
}
```

Pakiety

Pakiety w Javie sa pewnym zbiorami klas i interfejsów dostarczajacymi ochrone dostepu dla swoich składników oraz zapewniajacymi zarzadzanie nazwami (brak konfliktów nazw).

Aby utworzyc pakiet, nalezy umiescic wyrazenie package na poczatku kazdego pliku zródłowego który ma byc powiazany w pakiecie, np.

```
package nazwaPakietu;
```

Uwaga! Jesli plik .java zawiera więcej niż jedna klase, tylko jedna z nich moze byc publiczna i jej nazwa musi byc taka sama jak nazwa pliku. Jesli plik zródłowy nie zawiera słowa **package** , wszystkie jego klasy i interfejsy znajduja sie w pakiecie *default package* , bedacym pakietem dla małych i tymczasowych aplikacji.

Import pakietów

Import pojedynczego elementu pakietu odbywa sie za pomoca instrukcji import , np:

```
import java.awt.image.mojaklasa;
```

a wszystkich elementów pakietu za pomoca gwiazdki (asterisk):

```
import java.awt.image.*;
```

Import pakietów-uwagi

- umieszczenie gwiazdki w wiekszej liczbie instrukcji import powoduje wydłuzenie czasu kompilowania programu, szczególnie gdy importowane pakiety zawieraja wiele klas - nie ma to jednak wpływu na szybkosc działania skompilowanego programu,
- deklaracja importu nie oznacza włączania do pliku tekstu zawartego w pliku zródłowym pakietu(nie jest odpowiednikiem dyrektywy **include** preprocesora z C++)
- wazna jest kolejnosc deklaracji: najpierw deklaracja pakietu, po niej deklaracje importu, po czym definicje klas.
- standardowe klasy Javy należące do pakietu java.lang sa importowane automatycznie (domyślne `import java.lang.*;`)

Klasa object i jej metody

Klasa Object znajduje się w pakiecie `java.lang` i jest na najwyższej pozycji w hierarchii klas języka Java. Wszystkie inne klasy dziedziczą po niej w sposób bezpośredni lub pośredni. Stąd wszystkie klasy w programach Javy dziedziczą metody zdefiniowane w klasie `Object` takie jak:

- `protected Object clone() throws CloneNotSupportedException`
- `public boolean equals(Object obj)`
- protected void finalize() throws Throwable
- `public final Class getClass()`
- `public int hashCode()`
- `public String toString()`

Dodatkowo w klasie `Object` są zdefiniowane metody związane z programowaniem wielowątkowym, które będą omówione w osobnym wykładzie (`notify()`, `notifyAll()`, `wait()`, `wait(long timeout)`, `wait(long timeout, int nanos)`).

Metoda `clone()`

W Javie do kopiowania obiektów wykorzystuje się często metodę clone(), ale żeby móc jej używać nasza klasa powinna najpierw implementować interfejs `Cloneable`, jeśli tego nie zrobimy otrzymamy wyjątek `Clone not supported exception`. Klonowanie służy do stworzenia kopii naszego obiektu, w swojej podstawowej formie powoduje utworzenie nowej instancji obiektu tego samego typu co obiekt, na rzecz którego metoda została wywołana zachowując wszystkie pola wewnętrzne w takiej samej formie.

Metoda `clone()` tworzy nowy obiekt i kopiuje pola, ale pola kopiuje jedynie na zasadzie przepisania wartości pól typów prostych oraz przypisania tych samych referencji w przypadku pól typów obiektowych.

Metoda equals()

Metoda equals() służy w Javie do porównywania typów obiektowych. W odróżnieniu od typów prostych, operator == w przypadku typów obiektowych porównuje referencje, a nie równość strukturalną obiektów, z tego powodu porównanie dwóch obiektów poprzez:
`obiekt1 == obiekt2`
w większości przypadków jest nieskuteczne i otrzymujemy wynik, którego nie oczekiwaliśmy.
Metoda `equals()` zdefiniowana jest w klasie `Object`, a ponieważ jest to klasa, po której dziedziczą wszystkie klasy Javy, to możemy ją wywołać na dowolnym obiekcie. Domyślna implementacja metody `equals()` sprowadza się jednak jedynie do porównania referencji ==, czyli w większości przypadków nie jest zbyt użyteczna. W klasach, których obiekty będziemy ze sobą porównywali należy ją nadpisać.

Metoda hashCode()

Metoda hashCode() służy w Javie do zwrócenia unikalnej wartości liczbowej (typu int) dla każdego unikalnego obiektu. Istnieje kontrakt mówiący o tym, że dwa obiekty, których porównanie przy pomocy metody `equals()` zwraca `true`, to metoda `hashCode()` powinna zwracać dla tych obiektów taką samą wartość.

Metoda `hashCode()` jest dziedziczona z klasy `Object` i domyślnie powinna zwrócić unikalną wartość dla każdego obiektu. Jeśli jej nie nadpiszemy, to zwróci ona różne wartości nawet dla obiektów, które pod względem przechowywanych wartości są równe. Dzieję się tak ponieważ domyślnie metoda ta wykorzystuje do wyliczenia zwracanej wartości **adres obiektu w pamięci**. Ta sama wartość jest wykorzystywana w domyślnej metodzie toString, ale w postaci szesnastkowej.

Metody `getClass()` i `toString()`

Metoda `toString()` zwraca łańcuch znaków opisujący w sposób "tekstowy" obiekt dla którego została wywołana. W domyślnej implementacji w klasie `Object` jest to wartość referencji danego obiektu oraz jego typ w formacie `nazwaKlasyObiektu@adresWformacieSzesnastkowym` np. `mojaklasa@4383f74d`

Metoda `getClass()` zwraca obiekt klasy `Class`, która określa typ obiektu dla jakiego została wywołana. Metoda jest przydatna do określenia typu obiektu w trakcie przebiegu programu, kiedy odwołanie do niego jest zrealizowane za pomocą klasy nadrzędnej np. **Object**.

Klasy String i StringBuffer i ich metody

Platforma Javy dostarcza dwie klasy obsługujace napisy: String i StringBuffer. Klasa String jest wykorzystywana do przechowywania stałych napisów (lepsza optymalizacja kodu), a StringBuffer dla modyfikowalnych napisów.

```
public class StringsDemo {
  public static void main(String[] args) {
    String palindrome = "Dot␣saw␣I␣was␣Tod";
    int len = palindrome.length();
    StringBuffer dest = new StringBuffer(len);
    for (int i = (len - 1); i >= 0; i--) {
      dest.append(palindrome.charAt(i));
    }
  System.out.println(dest.toString());
  }
}
```

Konstruktory klas String i StringBuffer

`String()`, `StringBuffer()` inicjalizuje pusty łancuch,
`String(byte []bytes)` dekoduje tablice byte-ów i tworzy
`String(byte[] bytes, int offset, int length, String charsetName)` jw. ale z zastosowaniem tablicy kodowej
`String(char[] value)` kopiuje elementy z tablicy znaków do łancucha
`String(char[] value, int n, int m)` jw. ale tylko od n-tego elementu i m znaków
`String(String)`, `StringBuffer(String)` kopiujacy
`String(StringBuffer)` jw. ale ze StringBuffer
`StringBuffer(int n)` konstruuje (allokuje) łancuch o n elementach

Konstrukcja obiektów klasy String i StringBuffer:

```
StringBuffer string = new StringBuffer(10);
String s=new String();
char helloArray []= { 'h', 'e', 'l', 'l', 'o' };
String helloString [] = new String(helloArray);
byte ascii []= { 65,66,67 };
String ascii2=new String(ascii,0);
```

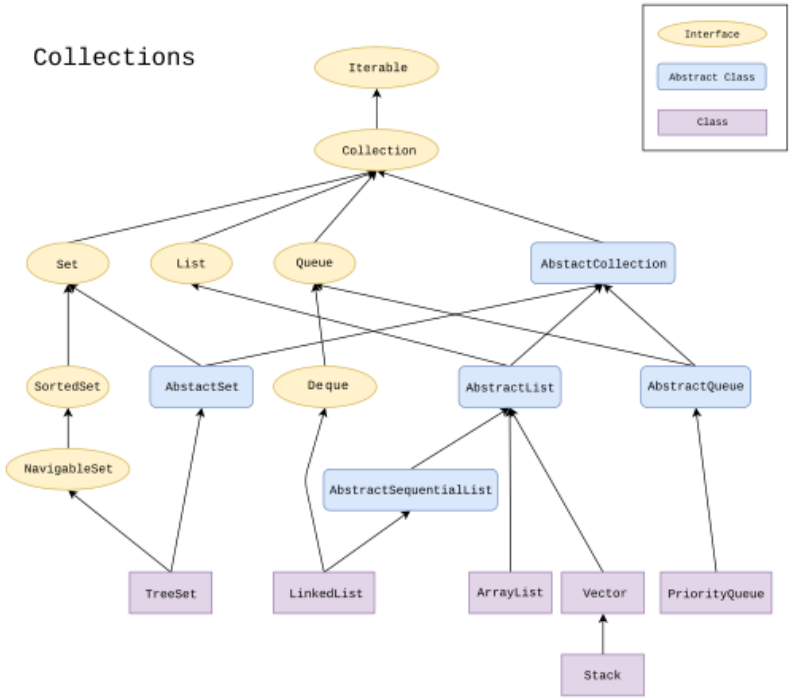
Konkatenacja łancuchów:

```
String s = "On␣ma␣" + age + "lat"; // tak naprawde wykona sie:
/* String s = new StringBuffer("On ma ").append(age)
                    .append("lat").toString();*/
```

Jest to spowodowane tym ze egzemplarze klasy String sa niezmienialne.
Przeciazenie operatora + dla klasy String jest odstepstwem od reguł Javy.

Collections – kolekcje

Kolekcje służą przechowywaniu danych. W odróżnieniu od tablic przede wszystkim nie narzucają niezmienności rozmiaru. Mogą dynamicznie zmieniać rozmiar w trakcie dodawania bądź usuwania elementów. Kolekcje nie mają możliwość przechowywania typów pierwotnych, muszą przechowywać zmienne referencyjne co musi zostać wykonane poprzez parametryzację na konkretny typ danych.



Do najczęściej używanych kolekcji należą klasy ArrayList, Vector i LinkedList.

ArrayList, Vector i LinkedList

Wszystkie trzy klasy implementują interfejs `List`. W kontekście użycia wszystkie trzy klasy są identyczne a różnią się implementacją sposobu dostępu i przechowywania danych. `ArrayList` i `Vector` są zaimplementowane jako tablice o dynamicznie dopasowywanym rozmiarze. Tzn jeśli elementy są dodawane do listy to jej rozmiar jest automatycznie powiększany. Elementy mogą być modyfikowane poprzez bezpośrednie użycie metod `get()` i `set()`. `LinkedList` jest zaimplementowana jako dwukierunkowa kolejka. Dzięki czemu operacje dodawania i usuwania elementów są realizowane szybciej w porównaniu do `ArrayListy` i `Vectora` ale kosztem dostępu i modyfikowania elementów na liście (co z kolei jest wykonywane szybciej w klasach `ArrayList` i `Vector`. Klasa `Vector` różni się od `ArrayList` dodatkową funkcjonalnością związaną z synchronizacją. Oznacza to, że klasa `ArrayList` może być modyfikowana poprzez 2 lub więcej wątków jednocześnie a vector dopuszcza tylko jeden wątek aby operował na przechowywanej kolekcji.

Dodatkowe różnice dotyczą alokacji pamięci podczas zmiany rozmiaru: `Vector` podwaja zaalokowaną pamięć a `ArrayList` zwiększa pojemność o 50% aktualnego rozmiaru.

ArrayList, Vector i LinkedList - przykład użycia

```
ArrayList<Integer> colection1 = new ArrayList<Integer>();
Vector<Integer> colection2 = new Vector<Integer>();
LinkedList<Integer> colection3 = new LinkedList<Integer>();
colection1.add(3); colection1.add(2);
colection2.add(3); colection2.add(2);
colection3.add(3); colection3.add(2);
Iterator<Integer> iter1 = colection1.itera      tor();
while(iter1.hasNext()){
  System.out.println(iter1.next());
}
for (Iterator<Integer> iter2=colection2.iterator();iter2.hasNe
  System.out.println(iter2.next());
}
for(int x : colection3){
  System.out.println(x);
}
```

Typy pierwotne i ich wrappery

Wrapperem nazywamy klasy, które enkapsulują inne typy (tworzony jest obiekt który zawiera w sobie inny nieobiektowy typ). Stąd wrapery typów pierwotnych udostępniają mechanizm tworzenia dla nich obiektów wraz z dedykowanymi dla nich polami i metodami.

typ pierwotny	nazwa wrappera	parametr konstruktora
byte	Byte	byte lub String
short	Short	short lub String
int	Integer	int lub String
long	Long	long lub String
float	Float	float, double lub String
double	Double	double lub String
char	Character	char
boolean	Boolean	boolean lub String

Wrappery – najczęściej wykorzystywane metody i pola

- Wrapery klas numerycznych (dziedziczące po klasie Number) – Byte, Short, Integer, Long, Float, Double:
  - Pola statyczne MIN_VALUE i MAX_VALUE,
  - Metody `parseXxxx(String s)` (np parseInt czy parseByte),
  - Metody `compare(typ a, typ b)`, `compareTo(typ a)`
  - Metoda `equals(typ a)`
  - Metoda `valueOf(typ a)`, `valueOf(String s)`
- Klasa Character
  - Metody `isAlphabetic()`, `isDigit()`, `isLower(Upper)Case()`, `toLower(Upper)Case()`,
- Klasa Boolean
  - Pola statyczne `Boolean.FALSE` i `Boolean.TRUE`
  - Metody z grupy `logical()`, `parseBoolean(String s)`,

Autoboxing i unboxing

- **Autoboxing** jest automatyczna konwersja typu pierwotnego do odpowiadającej mu klasy wrappera. Dzięki temu możliwe jest automatyczne wykorzystywanie typów pierwotnych w kolekcjach sparametryzowanych na ich wrappery.
  Dodatkowo możliwe jest bezpośrednie przypisanie typu pierwotnego do obiektu jego wrappera np.:
  `Integer zmienna= 8;`
  Aczkolwiek jest to niewskazane i bardziej zalecane jest wykorzystanie metody `valueOf`:
  `Integer zmienna= Integer.valueOf(8);`
- Z kolei `unboxing` jest to automatyczna konwersja wrappera to typu pierwotnego. Podobnie wykorzystanie tego mechanizmu zalecane jest w przypadku kolekcji czy innych metod. W bezpośrednim przypisaniu lepiej wykorzystać metodę *type*`Value()`, np.:
  `int x = zmienna.intValue();`

Podstawowe operacje wejścia i wyjścia (in/out, I/O)

Java dostarcza wiele klas do podstawowej obsługi operacji wejścia i wyjścia. Klasy te są wykorzystywane do zapisu i odczytu z plików, sieci czy konsoli. Podstawowe klasy służące do obsługi danych należą do pakietów `java.io i java.nio`. Klasy te obsługują podstawowe formatowanie danych, kompresję czy szyfrowanie.
Oba pakiety różnią się podejściem do zarządzania danymi:

IO	NIO
Stream oriented Blocking IO	Buffer oriented Non blocking IO

Obsluga plikow - odczyt

Odczyt **znakow** z pliku najwygodniej wykonuję się poprzez obiekt klasy `BufferedReader` należącą do pakietu `java.io`.:

```
BufferedReader br = new BufferedReader(new FileReader("plik.txt"));
String lineContents;
while ((lineContents = br.readLine())!= null) {
        \\operacje na wczytanych danych }
br.close();
```

Odczyt **danych** z pliku najwygodniej wykonuję się poprzez obiekt klasy `DataInputStream` należącą do pakietu `java.io`.:

```
DataInputStream inStream
= new DataInputStream(new FileInputStream("plik.bin"));
inStream.read();
```

lub w przypadku dużej liczby danych:

```
DataInputStream inStream = new DataInputStream(...
...   new BufferedInputStream(new FileInputStream(plik.bin)));
```

Metoda `read()` wczytuje dane bajt po bajcie. Aby zwrócić dane do strumienia można posłużyć się metodą `unread(int /int[])`

Obsluga plikow - zapis

Zapis danych tekstowych:

```
String in = "linia tekstu";
PrintWriter pw = new PrintWriter(new FileWriter("plik.txt"));
pw.println(in);
pw.close();
```

Zapis danych binarnych

```
File plik = new File("plik.bin");
int[] liczby = {0, 1, 2, 3, 4};
DataOutputStream outStream = new DataOutputStream(...
        ... new FileOutputStream(plik));
for (int i = 0; i < liczby.length; i++)
        outStream.writeInt(liczby[i]);
```

Jeśli podany plik nie istnieje to zostanie utworzony.

RandomAccessFile

```
private static byte[] readFromFile(String filePath, int position,
int size) throws IOException {
   RandomAccessFile file = new RandomAccessFile(filePath, "r");
   file.seek(position);
   byte[] bytes = new byte[size];
   file.read(bytes);
   file.close();
   return bytes;
}

private static void writeToFile(String filePath, String data,
int position) throws IOException {
   RandomAccessFile file = new RandomAccessFile(filePath, "rw");
   file.seek(position);
   file.write(data.getBytes());
   file.close();
}
```

Data i czas - java.util.date

```
public class SimpleDateFormatExample {
  public static void main(String[] args) {
    Date curDate = new Date();
    SimpleDateFormat format = new SimpleDateFormat("yyyy/MM/dd");
    String DateToStr = format.format(curDate);
    System.out.println(DateToStr);
    format = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
    DateToStr = format.format(curDate);
    System.out.println(DateToStr);
    format = new SimpleDateFormat("dd MMMM yyyy zzzz", Locale.ENGLISH
    DateToStr = format.format(curDate);
    System.out.println(DateToStr);
    format = new SimpleDateFormat("E, dd MMM yyyy HH:mm:ss z");
    DateToStr = format.format(curDate);
    System.out.println(DateToStr);
    try {
      Date strToDate = format.parse(DateToStr);
      System.out.println(strToDate);
    } catch (ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Data i czas – API, java.time

Java od wersji 1.7 dostarcza interfejs programistyczny do obsługi dat, czasu oraz kalendarza zgodnie z specyfikacja JSR-310 i modelem ISO 8601. Nowa implementacja jest zgodna z wcześniejszymi rozwiązanami i wprowa0dza szereg narzędzi konwersji czasu np.:

```
Calendar c = Calendar.getInstance();
Instant i = c.toInstant();
Date d = Date.from(i);

ZonedDateTime zdt =
ZonedDateTime.parse("2014-02-24T11:17:00+01:00"
+ "[Europe/Gibraltar]")
GregorianCalendar gc = GregorianCalendar.from(zdt);
LocalDateTime ldt
= gc.toZonedDateTime().toLocalDateTime();
```

Data i czas – przykłady

```
LocalDate currentDate = LocalDate.now();
LocalDate tenthFeb2014 = LocalDate.of(2014, Month.FEBRUARY, 10);
LocalDate firstAug2014 = LocalDate.of(2014, 8, 1);
LocalDate sixtyFifthDayOf2010 = LocalDate.ofYearDay(2010, 65);

LocalTime currentTime = LocalTime.now(); // current time
LocalTime midday = LocalTime.of(12, 0); // 12:00
LocalTime afterMidday = LocalTime.of(13, 30, 15); // 13:30:15
LocalTime fromSecondsOfDay = LocalTime.ofSecondOfDay(12345);


LocalDateTime currentDateTime = LocalDateTime.now();
LocalDateTime secondAug2014 = LocalDateTime.of(2014, 10, 2, 12, 30);
LocalDateTime christmas2014 = LocalDateTime.of(...
...   2014, Month.DECEMBER, 24, 12, 0);

LocalTime currentTimeInLosAngeles = ...
...   LocalTime.now(ZoneId.of("America/Los_Angeles"));
LocalTime nowInUtc = LocalTime.now(Clock.systemUTC());
```