Język Java i technologie WEB

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

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%

Literatura

Literatura obowiązkowa:

1. Java Platform, Standard Edition, API Documentation `https://docs.oracle.com/javase`
2. Schildt H.,Java. Kompendium programisty. Wydanie XI, 2020
3. Horstmann C.S., Java. Techniki zaawansowane. Wydanie XI, Helion, 2020.
4. Bruce Eckel, Thinking in Java, Wydanie 4, Helion, 2011.

Literatura dodatkowa:

1. J . Gosling, B. Joy, G. Steele, G. Bracha, Java Language Specification, Addison-Wesley Professional.
2. Oaks S., Java Performance. In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond. 2nd Edition, O'Reilly Media, 2020
3. Urma R.G., Fusco M., Mycroft A., Nowoczesna Java w działaniu. Wyrażenia lambda, strumienie, programowanie funkcyjne i reaktywne, Promise, 2018

Plan Wykładu

- Platforma Java
- Składnia języka, operatory i typy danych
- Programowanie obiektowe w Javie
- Obsługa wyjątków
- Programowanie wielowątkowe
- Programowanie sieciowe
- Interfejsy graficzne
- Dodatkowe funkcjonalności w Javie 8-19

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

Version	Release date	End of Free Public Updates[1][4]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** December 2020 for Oracle (personal use) At least May 2026 for AdoptOpenJDK At least May 2026[5] for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least October 2024 for AdoptOpenJDK At least September 2027[5] for Amazon Corretto	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

Legend: Old version | Older version, still maintained | **Latest version** | Future release

Wersje

Oracle Java SE Support Roadmap*†		
Release	**GA Date**	**Premier Support Until**
7 (LTS)	July 2011	July 2019
8 (LTS)**	March 2014	March 2022
9 (non-LTS)	September 2017	March 2018
10 (non-LTS)	March 2018	September 2018
11 (LTS)	September 2018	September 2023
12 (non-LTS)	March 2019	September 2019
13 (non-LTS)	September 2019	March 2020
14 (non-LTS)	March 2020	September 2020
15 (non-LTS)	September 2020	March 2021
16 (non-LTS)	March 2021	September 2021
17 (LTS)	September 2021	September 2026****
18 (non-LTS)***	March 2022	September 2022
19 (non-LTS)***	September 2022	March 2023
20 (non-LTS)***	March 2023	September 2023
21 (LTS)***	September 2023	September 2028

Implementacje

Build	LTS	Permissive license	TCK Tested	build of unmodified upstream	Commercial support available
AdoptOpenJDK[23] / IBM Java SDK[24]	Yes	Yes	No	Optional	Yes
Alibaba Dragonwell[25]	Yes	Yes	Yes	No	No
Amazon Corretto[26]	Yes	Yes	Yes	No[27]	No
Azul Zulu[28]	Yes	Yes	Yes	No	Yes
BellSoft Liberica JDK[29]	Yes	Yes	Yes	No	Yes
ojdkbuild[30]	Yes	Yes	No	Yes	No
OpenLogic OpenJDK[31]	Yes	Yes	No	No	Yes
Oracle Java SE[32]	Yes	No	Yes	No	Yes
Oracle OpenJDK[33]	No	Yes	Yes	Yes	No
Red Hat build of OpenJDK[34]	Yes	Yes	Yes	No	Yes
Red Hat build of OpenJDK for Windows[34]	Yes	Yes	Yes	No	Yes
SAP SapMachine[35]	Yes	Yes	Yes	No	No

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".

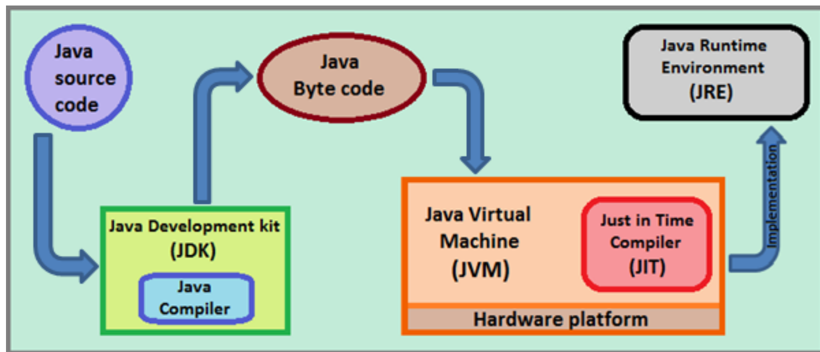
Wysoce wydajny (ang. high performance) Proces właściwego generowania kodu maszynowego z kodu bajtowego jest bardzo prosty, gdyż format kodu bajtowego został zaprojektowany z myślą o optymalizacji kodu maszynowego. Najbardziej kontrowersyjny aspekt platformy Java, w nowszych wersjach platformy założenie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

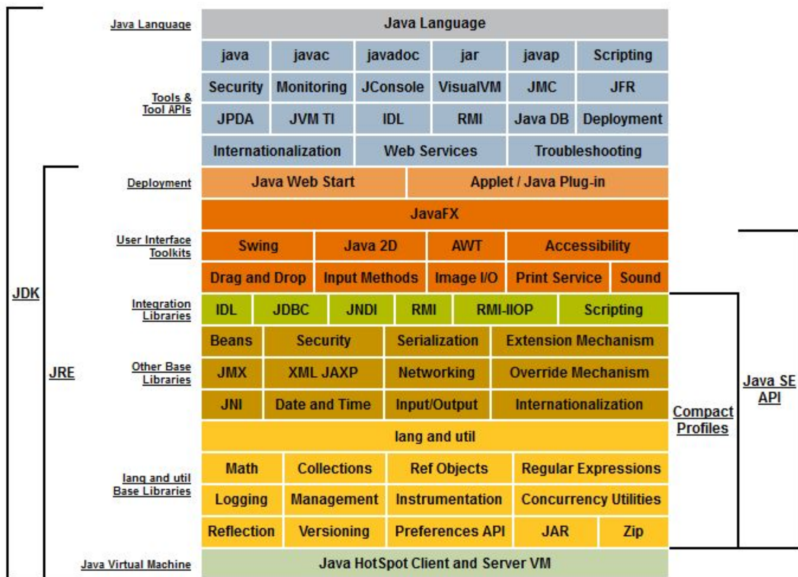
Założenia języka

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.

JRE, JDK i JVM

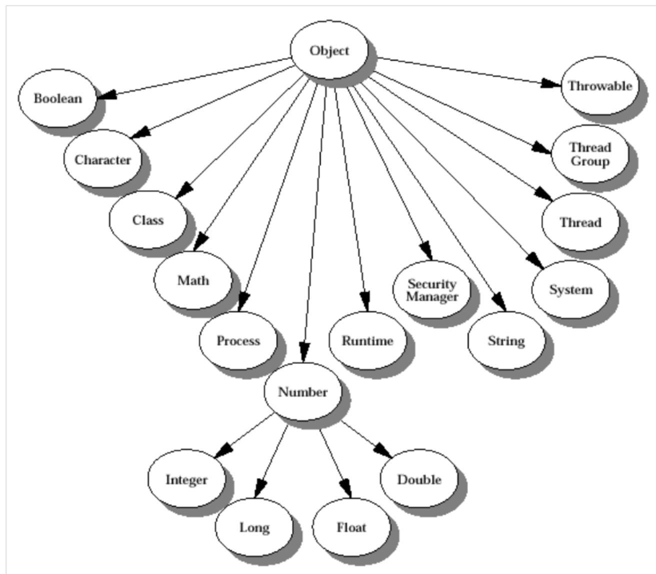



Java Language
Java Language
Tools & Tool APIs
java
javac
javadoc
jar
javap
Scripting
Security
Monitoring
JConsole
VisualVM
JMC
JFR
JPDA
JVM TI
IDL
RMI
Java DB
Deployment
Internationalization
Web Services
Troubleshooting
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface Toolkits
Swing
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 Base Libraries
lang and util
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
JDK
JRE
Compact Profiles
Java SE API

Biblioteki Javy

- Podstawowa dystrybucja Javy (Java SE) to ponad cztery tysiące klas.
- Najistotniejsze pakiety stanowiące trzon języka (`java.base`):
  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 (Java SE) to ponad cztery tysiące klas.
- Najistotniejsze pakiety stanowiące trzon języka (`java.base`):
  1. `java.lang`
  2. `java.io` i `java.nio`

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

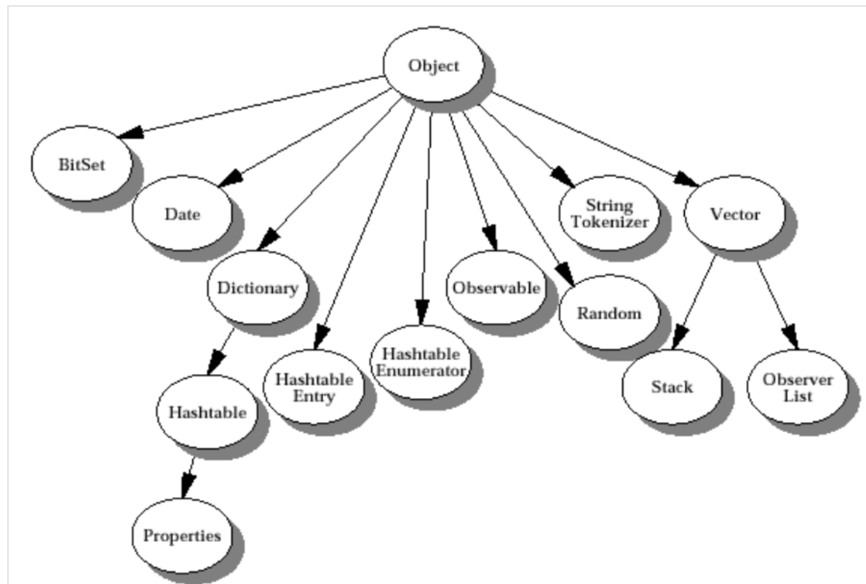
Biblioteki Javy

- Podstawowa dystrybucja Javy (Java SE) to ponad cztery tysiące klas.
- Najistotniejsze pakiety stanowiące trzon języka (`java.base`):
  1. `java.lang`
  2. `java.io` i `java.nio`

Biblioteki Javy

- Podstawowa dystrybucja Javy (Java SE) to ponad cztery tysiące klas.
- Najistotniejsze pakiety stanowiące trzon języka (`java.base`):
  1. `java.lang`
  2. `java.io` i `java.nio`
  3. `java.util`

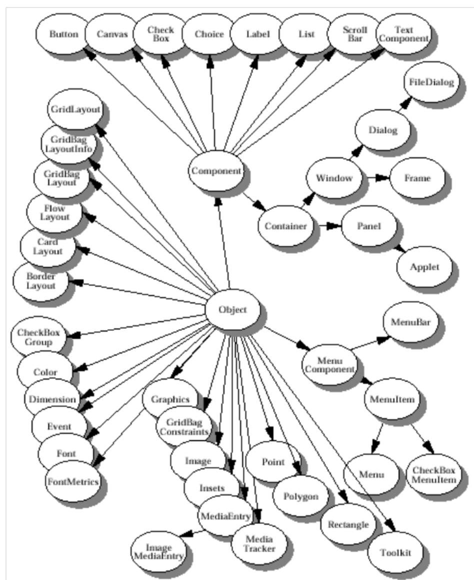
Biblioteki Javy



Biblioteki Javy

- Podstawowa dystrybucja Javy (Java SE) to ponad cztery tysiące klas.
- Najistotniejsze pakiety stanowiące trzon języka (`java.base`):
  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

void

class

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

Słowa kluczowe

abstract continue for if

else

void

class

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

Słowa kluczowe

abstract continue for if assert
else

void

class

```
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	else

void

class

```
switch(key){
 case 0: code; break;
 default: code; break;
}
```

Słowa kluczowe

abstract continue for if assert
switch default case break else

return

void

class extends

int

```
class B extends A{
  int anotherMetod() {code; return 0;}
}
```

Słowa kluczowe

abstract continue for if assert
switch default case break else

return

void

new class extends

int

```
B obj=new B();
```

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	
			return	

				void
new	class	extends		
				int

```
private int x=5; public void metod(){code};
```

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
			return	

new class extends void int

```
import javax.swing.JButton;
```

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do		return	

				void
new	class	extends		
				int

```
 while (condition) {code;}
do {code;} while (condition)
```

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	
				void
new	class	extends		
				int

```
enum Level {
 LOW,
 MEDIUM,
 HIGH
}
```

Słowa kluczowe

abstract continue for if assert
switch default case break else
package public protected private import
while do enum return final

void

new class extends

int

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	final
throw	try	catch	finally	throws
				void
new	class	extends		
				int

```
try { code}
catch (Exception e) { code}
finally { code}
```

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	final
throw	try	catch	finally	throws
this	super			void
new	class	extends		
				int

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	final
throw	try	catch	finally	throws
this	super	volatile		void
new	class	extends		
				int

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	final
throw	try	catch	finally	throws
this	super	volatile	static	void
new	class	extends		
				int

Słowa kluczowe

abstract	continue	for	if	assert
switch	default	case	break	else
package	public	protected	private	import
while	do	enum	return	final
throw	try	catch	finally	throws
this	super	volatile	static	void
new	class	extends	interface	implements
				int

```
Interface interfejs { metody abstrakcyjne}
class C implements interfejs{
 implementacja metod abstrakcyjnych
}
```

Słowa kluczowe

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

Słowa kluczowe

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

Słowa kluczowe

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

```
if{ obj instanceof B };
```

Słowa kluczowe

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

Słowa kluczowe

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

Słowa kluczowe

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

Słowa kluczowe

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

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. Definiowanie klas,tworzenie obiektów
7. Modyfikatory klas, metod i pól
8. Główne mechanizmy programowania obiektowego
   - Enkapsulacja
9. Dziedziczenie
10. Polimorfizm
11. Klasy zagnieżdżone, anonimowe i lokalne
12. Obsługa wyjątków
13. Programowaie wielowatkowe
14. Programowanie sieciowe w Javie
15. Tworzenie GUI
    - Przegląd klas Swing
16. JavaFX
    - JavaFX-wprowadzenie
    - JavaFX vs Swing
    - JavaFX-podstawowe klasy i interfejsy

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.

Klasa i Obiekt

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**.

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{
}
```

Tworzenie obiektów – 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 konstruk
  //kopiujacego
```

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 przypominającymi funkcje z języka C++. Każda funkcja w Javie jest związana z definicją klasy (spełnia role jej metody) i deklarowana jest wg. schematu:

```
modyfikatory TypRezultatu NazwaMetody(ListaParametrów) {
//treść 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: `public`, `private`, `protected`, `package`- `domyślny`
- modyfikatory właściwości modyfikowanego elementu:
  - 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;
  }
}
```

Główne mechanizmy programowania obiektowego

Języki zorientowane obiektowo zawierają następujące mechanizmy wymuszające stosowanie obiektów:

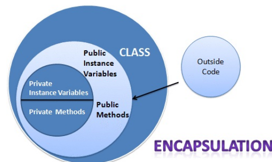
- enkapsulacja (ang. encapsulation),
- dziedziczenie (ang. inheritance),
- polimorfizm (ang. polymorphism).

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

Enkapsulacja 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*).

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.



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

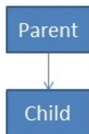
```
public class A {
        private int x;

        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);
        }
}
```

Dziedziczenie

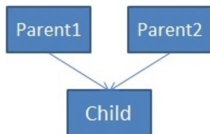
Większość ludzi czuje potrzebę uporządkowania obiektów świata rzeczywistego poprzez tworzenie złożonych taksonomii (klasyfikacji, kategoryzacji). W przypadku gdy pewne klasy tworzą abstrakcyjne, wspólne modele opisu (np. klasa Zwierzęta), wówczas na ich podstawie można utworzyć podklasy (np. klasa Ssaki) 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

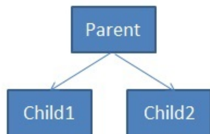


1. Single Inheritance

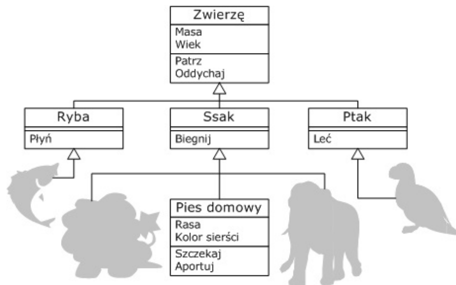


2. Multiple Inheritance

3. Multi-Level Inheritance



4. Hierarchical Inheritance



Modyfikatory dostepu i dziedziczenie

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.

Przykrywanie metod

```java
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
```

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();
}}
```

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. nie jest poprawne(ale mozna przeprowadzic konwersje referencji `this`).

```
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.

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 abstrakcyjne

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. Jednakże każda klasa, która ma metodę abstrakcyjną, 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 może zawierać ich nie-abstrakcyjne wersje.
- Kazda podklasa klasy abstrakcyjnej **musi** implementować 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
  }
}
```

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";
    }
}
```

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, 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, StringBuffer, StringBuilder i ich metody

Platforma Javy dostarcza trzy klasy obsługujace napisy: String, StringBuffer i StringBuilder. Klasa String jest wykorzystywana do przechowywania stałych napisów (lepsza optymalizacja kodu), a StringBuffer i StringBuilder 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);
```

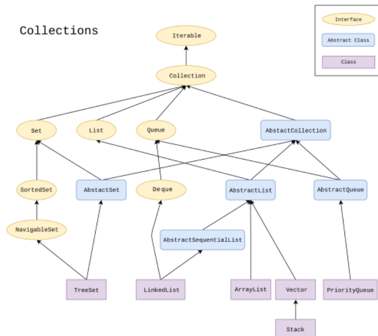
Konkatenacja łancuchów:

```
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.hasNex
  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 and Channel 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();
    }
  }
}
```

Pakiet java.util.Date jest generalnie źle zaprojektowany i początkowe błędy w trakcie rozwoju platformy Java były coraz bardziej pogłębiane (np., wycofanie wielu metod, dodanie klas Calendar czy GregorianCalendar). Ponadto klasa Date przedstawia nie datę a punkt w czasie. Przez co:

- nie określa strefy czasowej (time zone),
- nie określa formatu daty,
- nie określa systemu kalendarza.

Dodatkowo traktuje zapis rocznika jako dwie cyfry poczynając od 1900 roku. Mnogość rozwiązań obchodzących ten problemu powoduje brak przejrzystości implementacji dla dat sprzed tego roku. Miesiące są indeksowane od 0 (0- styczeń, 11 December) co nie jest intuicyjne i powoduje wiele błędów.

Data i czas – API, java.time

Java od wersji 1.8 dostarcza interfejs programistyczny do obsługi dat, czasu oraz kalendarza zgodnie z specyfikacja JSR-310 i modelem ISO 8601. Nowa implementacja jest zgodna z wcześniejszymi rozwiązaniami i wprowadza szereg narzędzi konwersji czasu np.:

```
Calendar c = Calendar.getInstance();
// klasa z java.util.date
Instant i = c.toInstant();
// konwersja na nowe API
Date d = Date.from(i);
// konwersja powrotna z Instant na klase Date (do starego API)
LocalDate date =
= d.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
// ponowna kowersja Date do LocalDate
```

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());
```

Podstawy obsługi wyjatków

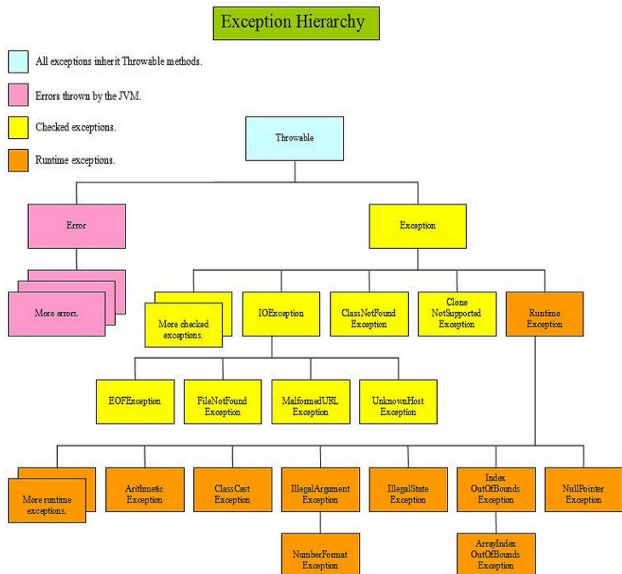
Wyjatkiem jest sytuacja nietypowa pojawiajaca sie podczas działania programu, która zakłóca jego prawidłowe wykonanie.

Mechanizm obsługi wyjatków w Javie umozliwia zaprogramowanie ”wyjscia” z takich sytuacji krytycznych, dzieki czemu program nie zawiesi sie po wystapieniu błedu wykonujac ciag operacji obsługujacych wyjatek.

Generowanie i obsługe sytuacji wyjatkowych w Javie zrealizowano przy wykorzystaniu paradygmatu programowania zorientowanego obiektowo. Z obsługa wyjatków zwiazane sa nastepujace słowa kluczowe: `try`, `catch`, `throw`, `throws`, `finally`.

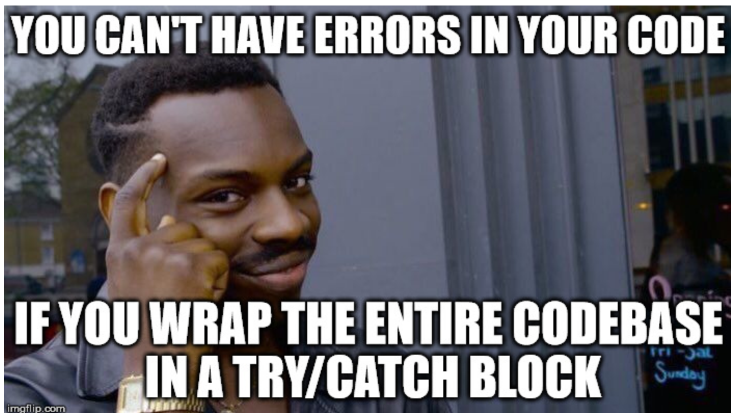
Typy wyjatków

W Javie kazdy wyjatek jest reprezentowany przez obiekt okreslonego typu. Wszystkie wyjatki, jakie moga wystapic w programie musza byc podklasa klasy `Throwable`.



Typy wyjatków

- Wyjatki typu `Error` wystepuja wtedy, gdy wystapi sytuacja specjalna w maszynie wirtualnej (np. bład podczas dynamicznego łaczenia) -nie powinny byc obsługiwane w "zwykłych" programach Javy.
- W wiekszosci programów generowane sa i obsługiwane obiekty, które dziedzicza z klasy `Exception`. Wyjatek tego typu oznacza, ze w programie wystapił bład, lecz nie jest to powazny bład systemowy.
- Klasa `RuntimeException` reprezentuje wyjatki, których wystapienie zostało spowodowane przez system czasu przebiegu Javy - sa generowane automatycznie w nastepstwie nieprawidłowego działania programu (np.: `NullPointerException`, `ClassCastException`, `IllegalThreadStateException` i `ArrayOutOfBoundsException`).


YOU CAN'T HAVE ERRORS IN YOUR CODE
IF YOU WRAP THE ENTIRE CODEBASE
IN A TRY/CATCH BLOCK
imgflip.com

```
try
{
caly kod naszego programu
}
catch (Exception nazwaZmiennej){
// obsluga wszystkich mozliwych wyjatkow
}
```

Prawidłowa implementacja obsługi wyjatków

```
try
{ //blok instrukcji gdzie moze wystapic wyjatek
}
catch (PodklasaThrowable nazwaZmiennej){
// blok instrukcji obslugujacych wystapienia sytuacji wyjatkowej
// jest wykonywany tylko, gdy wystapi wyjatek typu takiego jak
// typ zmiennej bedacej parametrem bloku catch
}
catch (PodklasaThrowable nazwaZmiennej)
{ . . . }
catch (PodklasaThrowable nazwaZmiennej)
{ . . . }
finally //opcjonalnie
{
// domyslna instrukcja obslugi wyjatku,
// wystapi nawet jesli blok try zawiera instrukcje
// return lub spowodowal wystapienie wyjatku
}
```

Obsługa wyjatków przez system czasu przebiegu

```
class Wyjatek {
  public static void main(String args[]) {
   int d=0;
   int a=42/d;
  }
}
...
java.lang.ArithmeticException: / by zero
at Wyjatek.main(Wyjatek.java:5)
Exception in thread "main"
```

System czasu przebiegu generuje (ang. throws) wyjatek.Wpodanym przykładzie nie została zdefiniowana procedura obsługi wyjatku, wiec wykonana zostaje procedura domyslna, nalezaca do systemu czasu przebiegu (wyswietlajaca wartosc obiektu typu String).

Obsługa wyjatków przez program

```
class Wyjatek {
  public static void main(String args[]) {
    try {
      int d=0; int a=42/d;
      int c[]={1};
      c[42]=99;
    } catch (ArithmeticException e) {
      System.out.println("dzielenie␣przez␣zero");
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Przekroczenie␣zakresu␣tablicy");
    }
  }
}
```

Zagniezdzone instrukcje try

```
class MultiWyjatek {
  static void procedure() {
    try {
    int c[]={1};
    c[42]=99;
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Przekroczenie␣zakresu␣tablicy");
      try {
      int d=0;
      int a=42/d;
      } catch (ArithmeticException e) {
        System.out.println("dzielenie␣przez␣zero");
      }
    }
  }
}
```

Generacja sytuacji wyjatkowych

Wyrazenie `throw` przekazuje sterowanie do skojarzonego z nim bloku `catch` (łap, blok obsługujacy wystapienie sytuacji wyjatkowej).

Jesli nie ma bloku `catch` w bieżącej metodzie, sterowanie natychmiastowo, bez zwracania wartości, przekazywane jest do metody, która wywołała bieżąca funkcje. W tej metodzie szukany jest blok `catch`.

Jesli blok `catch` nie zostanie znaleziony, przekazuje sterowanie do metody, która wywołała te metode. Sterowanie przekazywane jest zatem zgodnie z **łancuchem wywołan metod**, aż do momentu znalezienia bloku `catch` odpowiedzialnego za obsługe wyjatku.

Jesli wyjatek nie moze byc przechwycony, nalezy go zabezpieczyc (przy pomocy `throws`).

Generacja sytuacji wyjatkowych

```
public class WywolajWyjatek {
  static public void main(String args[]) throws Exception {
    Liczba liczba = new Liczba();
    liczba.dziel(1);
  }
}
class Liczba {
  int m_i = 10;
  int dziel(float i) throws Exception {
    if (i/2 != 0)
    throw new Exception("Liczba nieparzysta!");
    if (i == 0)
      throw new Exception("Dzielenie przez zero!");
    return (int)(m_i/i);
  }
}
.....
java.lang.Exception: Liczba nieparzysta!
at Liczba.dziel(WywolajWyjatek.java:16)
at WywolajWyjatek.main(WywolajWyjatek.java:6)
Exception in thread "main"
```

Słowo kluczowe throws

Jezeli metoda jest zdolna generowac wyjatek którego sama nie przechwytuje, nalezy to zaznaczyc w tekscie programu, aby metody ja wywołujace mogły sie przed nim zabezpieczyc. Liste wszystkich wyjatków generowanych przez dana metode tworzy sie za pomoca słowa kluczowego `throws`.

Próba skompilowania poprzedniego programu bez uzycia `throws` zakonczyłaby sie wyswieteleniem komunikatu:
`Unhandled exception type Exception`

Słowo kluczowe throws - przykład

```
public class WywolajWyjatek {
static public void main(String args[]) {
  Liczba liczba = new Liczba();
  try {
    liczba.dziel(1);
  } catch(Exception e){System.out.println("wyjatek");}
    // wylapuje wyjatek "wyrzucany" poprzez funkcje dziel
  }
}
class Liczba {
  int m_i = 10;
  int dziel(float i) throws Exception {
    if (i/2 != 0)
      throw new Exception("Liczba␣nieparzysta!");
    if (i == 0)
      throw new Exception("Dzielenie␣przez␣zero!");
    return (int)(m_i/i);
  }
}
....
wyjatek
```

Słowo kluczowe finally

Sterowanie opuszcza blok `try` w przypadku wystapienia instrukcji `return` lub sytuacji wyjatkowej. Java pozwala jednak zdefiniowac blok instrukcji, które będą wykonane zanim sterowanie opuści metode niezaleznie od tego, czym jest to spowodowane. Jest to blok finalny (ang. *finally block*), nazywany tak od słowa kluczowego `finally`.

Zastosowanie bloku `finally` pozwala uniknac dublowania kodu, który musiałby byc napisany zarówno dla przypadku, gdy wystapi wyjatek, jak i dla normalnego toku wykonania programu. Blok finalny jest odpowiednim miejscem do zwolnienia zasobów zarezerwowanych przez metode, poniewaz zasoby te powinny byc zwolnione niezaleznie od tego, czy wykonanie programu przebiegło w sposób zaplanowany, czy tez wystapił wyjatek.

Przykład konstrukcji bloku finalnego

```
...
PrintWriter out = null;
try {
  out = new PrintWriter(new FileWriter("plik.txt"));
  // FileWriter generuje wyjatek jesli plik
  // nie moze byc otwarty
  out.println("tekst␣do␣pliku");
} catch (IOException e) {
  System.err.println("Wylapany␣IOException:␣" + e.getMessage()
} finally {
  if (out != null) {
  System.out.println("Zamykanie␣PrintWriter");
  out.close();
  }
  else {
    System.out.println("PrintWriter␣nie␣zostal␣otwarty");
  }
}
```

Tworzenie podklas klasy Exception

W Javie umozliwiono definiowanie klasy wyjatków, które beda obsługiwały sytuacje, uznane przez programiste za wyjatkowe.

```
class NaszException extends Exception {
  private int detail;
  NaszException(int a) {
    detail=a;
  }
  public String toString() {
    return "NaszExeption[" + detail + "]";
  }
}
```

Przykład przechwycenia zdefiniowanego wyjatku

```
class ExceptionDemo {
  static void oblicz(int a) throws NaszException {
    System.out.println("Wywolana␣metoda␣oblicz␣(" + a +")");
    if (a>10) throw new NaszException(a);
      System.out.println("Wszystko␣OK");
  }
}
public static void main(String args[]) {
  try {
    oblicz(1);
    oblicz(20);
  } catch (NaszException e){
    System.out.println("Przechwycony␣wyjatek:␣" + e);
  }
}
```

```
try {
   something
} catch(e) {
   window.location.href =
       "http://stackoverflow.com/search?q=[js] + "
       + e.message;
}
```

Programowanie wielowatkowe

W programowaniu **sekwencyjnym** , kazdy program ma poczatek, sekwencje instrukcji do wykonania i koniec. W kazdym momencie działania programu mozemy wskazac miejsce, w którym znajduje sie sterowanie. Taki program stanowi zatem pojedynczy, sekwencyjny przepływ sterowania, zwany watkiem (ang. *thread*).

Program moze składac sie z wielu przepływów sterowania, co okreslamy jako programowanie **wielowatkowe**.

Do zadan wykonywanych współbieznie nalezy m.in. odczyt i zapis plików, korzystanie z zasobów sieciowych, reagowanie na dane wprowadzane przez uzytkownika, wykonywanie skomplikowanych obliczen, wyswietlanie animacji i interfejsu uzytkownika.

Wielozadaniowosc vs. wielowatkowosc

- W srodowisku wielozadaniowym procesy nazywaja sie zadaniami(*tasks*) lub procesami ciezkimi (*heavy-weight processes*). Zadania zajmuja oddzielne przestrzenie adresowe i z punku widzenia uzytkownika sa róznymi programami działajacymi w tym samym systemie (np. Word i Excel). Komunikowanie sie zadan, zmiana aktywnego zadania czy zmiana biezacego kontekstu wymagaja znacznych nakładów i podlegaja istotnym ograniczeniom. Zwykle w danej chwili korzysta sie tylko z jednego programu, nawet jesli uruchomionych zostało wiecej.
- W srodowisku wielowatkowym procesy nazywa sie watkami. Współdziela one te sama przestrzen adresowa i naleza do tego samego zadania. Dlatego ich wzajemne komunikowanie sie i zmiana aktywnego procesu przebiegaja szybko i sa integralna czescia wykonywania pojedynczego programu.

Watek ...

- ma poczatek, sekwencje instrukcji i koniec,
- nie jest niezaleznym programem, jest wykonywany jako czesc programu,
- moze byc wykonywanych jednoczesnie z innymi, a kazdy z nich moze wykonywac w tym samym czasie odmienne zadania (ang. *tasks*),
- jest obiektem zdefiniowanym za pomoca specjalnego rodzaju klas.
- ma swoje zarezerwowane zasoby (jak np. licznik instrukcji), lecz oprócz tego moze korzystac z zasobów programu, w którym jest wykonywany.
- definiujemy jako podklase klasy `Thread` i/lub implementujac interfejs `Runnable`.

Srodowisko wielowatkowe

- Jesli program napisany wielowatkowo (ang. multithreaded), wykonywany jest na maszynie wieloprocesorowej, to rózne watki moga byc wykonywane w tym samym czasie na róznych procesorach. Sterowanie programu w takich przypadkach przebiega współbieznie (ang. concurrent).
- Na komputerach jednoprocesorowych wykonanie programów wielowatkowych jest tylko emulowane. Emulacja ta polega na naprzemiennym przydzielaniu czasu procesora poszczególnym watkom wg. pewnego algorytmu (zaimplementowanego w systemie operacyjnym). To, w jakim stopniu watek bedzie mógł wykorzystywac procesor, zalezy od priorytetu watku.

Model watków w Javie

- srodowisko programowania Javy jest asynchroniczne (nie wystepuje pojedyncza petla zdarzen),
- jesli jakis watek został zablokowany (np. czeka na wprowadzenie danych uzytkownika) to moze zostac zawieszony, a w tym czasie zostanie wykonany inny. Wznowiony watek kontynuuje działanie od miejsca w którym został zawieszony,
- watek moze sam zwolnic procesor lub moze zostac wywłaszczony przez watek o wyzszym priorytecie,
- priorytety watków sa liczbami całkowitymi 1 - 10,
- jesli dwa watki o takim samym priorytecie zabiegaja o czas procesora, to kazdy z nich musi sam oddac sterowanie drugiemu (jeden nie moze wywłaszczyc drugiego)

Model watków w Javie

- istnieje mozliwosc synchronizacji watków (konieczne np.gdy dwa watki maja współdzielic skomplikowana strukture danych) za pomoca modelu synchronizacji wewnatrzprocesowej Hoare'a (zwanej monitorem),
- monitor jest to "mała skrzynka" zdolna pomiescic tylko jeden watek - pozostałe musza czekac az monitor zostanie zwolniony. W Javie kazdy obiekt ma swój własny monitor, do którego kazdy watek moze wejsc przez wywołanie jednej z metod oznaczonych słowem synchronized. W czasie gdy jeden watek wykonuje metode synchronizowana, zaden inny nie moze wywołac jakiejkolwiek innej metody synchronizowanej tego samego obiektu,
- watki wymieniaja miedzy soba informacje za pomoca metod `wait()` i `notify()`. Kazdy watek moze rozpoczac wykonanie synchronizowanej metody i czekac w niej na zajscie pewnego zdarzenia tak długo az inny watek nie zawiadomi go o tym ze zdarzenie to juz wystapiło.

Ciało watku

Wszystkie zadania, jakie ma wykonywac watek umieszczone sa w metodzie `run` watku. Po utworzeniu i inicjalizacji watku, srodowisko przetwarzania wywołuje metode `run`. W ciele metody `run` czesto pojawia sie petla. Na przykład, watek odpowiedzialny za animacje w petli w metodzie `run` moze wyswietlac serie obrazków. Niekiedy metoda run watku wykonuje operacje, które zajmuja duzo czasu np. ładowanie i odgrywanie dzwieków lub filmów. Istnieja dwie metody dostarczenia metody `run`

- tworzenie podklasy klasy Thread i przykrywanie metody run,
- implementowanie interfejsu `Runnable`.

Reguła wyboru: jeśli klasa musi dziedziczyć po innej klasie, to należy wybrać opcje druga.

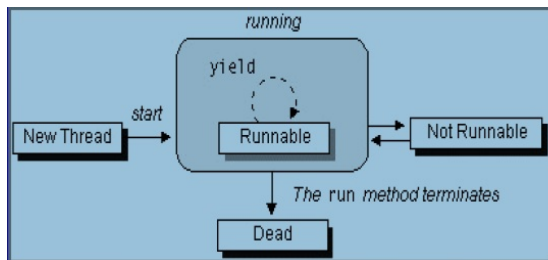
Implementacja wielowątkowości

```
public class WatekDemoThread extends Thread {
  public WatekDemoThread(String str) { super(str); }
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i + "␣" + getName());
      try sleep((long)(Math.random() * 1000));
      catch (InterruptedException e) {}
    }
    System.out.println("Zrobione!␣" + getName());
  }
}
public class WatekDemoRunnable implements Runnable{
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println(i + "␣" + Thread.currentThread().getName());
      try Thread.sleep((long)(Math.random() * 1000));
      catch (InterruptedException e) {}
    }
    System.out.println("Zrobione!␣" + Thread.currentThread().getName());
  }
}
```

Implementacja wielwątkowości

```
public class DemoObuMetod {
public static void main (String[] args) {
new WatekDemoThread("Jamaica").start();
new WatekDemoThread("Fiji").start();
new Thread(new WatekDemoRunnable(),"Fiji").start();
new Thread(new WatekDemoRunnable(),"Jamaica").start();
}
}
```

Cykl zycia watku



Cykl zycia watku - nowy watek

W pierwszej kolejności wątek musi zostać utworzony z pomocą konstruktora jak każdy inny obiekt. Po wykonaniu tej instrukcji mamy zaledwie pusty obiekt `Thread`. Zadne zasoby systemowe nie zostały jeszcze alokowane dla tego watku. Kiedy watek znajduje sie w tym stanie, mozemy jedynie wykonac metode start, uruchamiajaca watek, lub stop, konczaca działania watku. Wszelkie próby wywołania innych metod dla watków w tym stanie nie maja sensu i powoduja wystapienie wyjatku `IllegalThreadStateException`.

Cykl zycia watku - wykonywanie

Metoda start() tworzy zasoby systemowe potrzebne do wykonania watku, przygotowuje watek do uruchomienia, oraz wywołuje metode run. Od tego momentu watek jest w stanie "wykonywany". Nie oznacza to jednak automatycznie, ze watek zostaje uruchomiony. Niemozliwe jest uruchomienie wielu watków w tym samym momencie (liczbę tą ogranicza dostępność rdzeni procesora). Srodowisko przetwarzania Javy musi implementowac system przydziału czasu procesora (ang. scheduler), który dzieli czas procesora miedzy wszystkie watki bedace w stanie "wykonywany"

Cykl zycia watku - nie wykonywanie

Watek przechodzi do stanu ''nie wykonywany'' gdy zachodzi jedno z ponizszych zdarzen:

- wywołano jego metode sleep,
- wywołano jego metode suspend,
- watek wykonuje swoja metode wait,
- watek jest zablokowany przy operacji wejscia / wyjscia (ang. I/O).

Cykl zycia watku - zakonczenie działania

Watek moze zakonczyc działanie z dwóch powodów: albo naturalnie zakonczy swe działanie (gdy jego metoda run konczy sie normalnie) albo zostanie zabity (przez wywołanie metody stop w metodzie run).

Metoda stop oznacza nagłe zakonczenie wykonania metody run watku i nie jest zalecana.

Cykl zycia watku - metoda isAlive

Interfejs programistyczny dla klasy Thread zawiera metode `isAlive`. Wynikiem wykonania metody isAlive jest wartosc true, gdy watek został uruchomiony a nie został jeszcze zakonczony. Gdy wynikiem jest wartosc true wiemy, ze jest albo w stanie "wykonywany" lub "nie wykonywany".

Gdy wynikiem wykonania metody isAlive jest wartosc false oznacza to, ze watek jest albo w stanie "nowy watek" lub "zakonczony".

Priorytety watków

W praktyce wiekszosc komputerów posiada mniejszą ilość procesorów bądź jego rdzeni niż aktualnie obsługiwana ilość watków, wiec w danej chwili czasu wykonywany moze byc tylko ograniczona liczba wątków i wielowatkowosc jest częsciowo emulowana. Wykonanie wielu watków w jakiejs kolejnosci nazywane jest szeregowaniem (ang. scheduling).

Java implementuje bardzo prosty, deterministyczny algorytm szeregowania znany jako "planowanie priorytetowe" (ang. fixed priority scheduling). Kazdemu watkowi przypisuje sie pewien priorytet, po czym przydziela sie dostępny rdzeń/procesor temu watkowi, którego priorytet jest najwyzszy (priorytety watków sa liczbami całkowitymi 1 - 10; stałe MIN PRIORITY i MAX PRIORITY klasy Thread).

Priorytety watków

- nowy watek dziedziczy priorytet z watku, który go utworzył,
- domyslnie wartosc priorytetu dla watku wynosi 5 (NORM PRIORITY),
- priorytety watku moga byc modyfikowane w kazdej chwili po utworzeniu watku poprzez uzycie metody setPriority .
- gdy watek zatrzymuje sie, ustepuje czas procesora lub przechodzi do stanu ''nie wykonywany'', watek o nizszym priorytecie moze byc wykonywany.

Priorytety watków

Wybrany watek bedzie wykonywany do momentu, gdy jeden z ponizszych warunków bedzie spełniony

- watek o wyzszym priorytecie znalazł sie w stanie "wykonywany",
- watek ustepuje procesor lub metoda run konczy działanie,
- w systemie, w którym stosuje sie segmentowanie czasu, przydział (kwant) czasu sie skonczył.

Uwaga: Nie jest całkowicie zagwarantowane ze w danym momencie watek o najwyzszym priorytecie jest wykonywany. Program szeregujacy watki moze wybrac do wykonania watek z nizszym priorytetem, aby uniknac zagłodzenia. Z tego powodu nie nalezy całkowicie polegac na priorytetach (dla poprawnosci algorytmu), a raczej uzywac ich jako strategii harmonogramowania podnoszacej wydajnosc programu.

Segmentowanie czasu SC(ang. time slicing)

- gdy watki o tym samym prioryt. czekaja na przydział czasu procesora, program szeregujacy wybiera jeden z nich i przydziela czas według algorytmu ”szeregowania cyklicznego”(karuzeli - ang. round-robin);
- niektóre systemy (np. Windows) zwalczaja ”samolubne” zachowanie watków poprzez strategie SC. W algorytmie tym ustala sie mała jednostke czasu, nazywana kwantem czasu lub odcinkiem czasu. Planista przydziału procesora przeglada kolejke cykliczna i kazdemu watkowi przydziela odcinek czasu nie dłuzszy od jednego kwantu czasu. Gdy watek ma czas wykonania dłuzszy, niz kwant czasu, to nastapi przerwanie wykonywania watku i zostanie on odłozony na koniec kolejki.

Priorytety watków - przykład

Dana jest klasa reprezentujaca watek:

```
public class SelfishRunner extends Thread {
 private int tick = 1;
 private int num;
 public SelfishRunner(int numer_watku) {
  this.num = numer_watku;
 }
 public void run() {
  while (tick < 400000) {
   tick++;
   if ((tick \% 50000) == 0)
   System.out.println("Watek␣#" + num + ",␣czas␣=␣" + tick);
  }
 }
}
```

Priorytety watków - przykład

Klasa Wyscig pozwala sprawdzic czy system operacyjny pozwala na segmentowanie czasu. Jesli tak, to dwa watki powinny wykonywac sie naprzemiennie, co mozna zauwazyc na ekranie gdyz metoda run klasy SelfishRunner to wyswietla.

```
public class Wyscig {
 private final static int NUMRUNNERS = 2;
 public static void main(String[] args) {
  SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];
  for (int i = 0; i < NUMRUNNERS; i++) {
   runners[i] = new SelfishRunner(i);
   runners[i].setPriority(2);
  }
  for (int i = 0; i < NUMRUNNERS; i++)
  runners[i].start();
 }
}
```

Synchronizacja watków

Gdy dwa lub wiecej watków ubiega sie jednoczesnie o dostep do współdzielonych danych, to trzeba spowodowac by korzystały one z nich po kolei. W Javie kazdy obiekt ma swój własny monitor, do którego kazdy watek moze wejsc przez wywołanie jednej z metod oznaczonych słowem synchronized. W czasie gdy jeden watek wykonuje metode synchronizowana, zaden inny nie moze wywołac jakiejkolwiek innej metody synchronizowanej tego samego obiektu.

Segment kodu w programie, w którym nastepuje dostep do tej samej danej z róznych watków nazywany jest sekcja krytyczna i oznaczany jest słowem synchronized (paradygmat programowania obiektowego wymaga by była to metoda).

Producent/Konsument

Jedna z sytuacji współdzielenia zasobów jest problem typu Producent/ Konsument (ang. producer/consumer), gdzie producent generuje strumien danych, które sa wykorzystywane (konsumowane) przez konsumenta. Strumien ten stanowi wspólny zasób, watki musza byc zatem synchronizowane.

Niech Producent generuje liczby od 0 do 9, które sa nastepnie składowane w obiekcie typu Pudelko. Producent, po włozeniu do pudełka liczby i wydrukowaniu jej na ekranie, zostaje uspiony na losowo wybrany czas (0 a 100 msek.), zanim przejdzie do nastepnego cyklu produkcji liczby.

Producent - Konsument

```
class Producent extends Thread {
  private Pudelko pudelko;
  private int m_nLiczba;
  public Producent(Pudelko c, int liczba)
  { pudelko = c;
    this.m_nLiczba = liczba;
  }
  public void run()
  {
    for (int i = 0; i < 10; i++)
    { pudelko.wloz(i);
      System.out.println("Producent␣#" + this.m_nLiczba
      + "␣wlozyl:␣" + i);
      try
      { sleep((int)(Math.random() * 100)); }
      catch (InterruptedException e) { }
    }
  }
}
```

Producent - Konsument

Konsument podczas swego działania konsumuje wszystkie liczby złozone w pudełku, wyprodukowane przez Producenta, tak szybko, jak stana sie one dostepne.

```
class Konsument extends Thread
{ private Pudelko pudelko;
  private int m_nLiczba;
  public Konsument(Pudelko c, int Liczba)
  {
    pudelko = c;
    this.m_nLiczba = Liczba;
  }
  public void run()
  { int wartosc = 0;
    for (int i = 0; i < 10; i++)
    {
      wartosc = pudelko.wez();
      System.out.println("Konsument␣#" + this.m_nLiczba
      + "␣wyjal:␣" + wartosc);
    }
```

Pudełko

Producent i konsument w tym przykładzie współdziela dane przez wspólny obiekt typu Pudelko. Konsument ma prawo pobrac kazda wyprodukowana liczbe tylko raz. Synchronizacja miedzy tymi dwoma watkami wystepuje w metodach wez() i wloz() obiektu Pudelko.

Metody `wait` i `notifyAll` słuza do koordynacji wkładania i wyjmowania liczb z Pudełka. Watek Konsumenta woła metode wez i zajmuje monitor obiektu Pudelko na czas wykonania metody wez. Na koncu metody wez, wywołanie metody notifyAll budzi watek Producenta oczekujacy na monitor obiektu Pudelko. W tym momencie watek Producenta zajmuje monitor i wykonuje swoje zadanie.

```
class Pudelko {
  private int m_nZawartosc; //zmienna warunkowa,
  private boolean m_bDostepne = false;
  public synchronized int wez()
  { while (m_bDostepne == false)
    { try { wait(); }
      catch (InterruptedException e) { }
    }
    m_bDostepne = false;
    notifyAll(); // zawiadomienie Producenta
    return m_nZawartosc;
  }
  public synchronized void wloz(int wartosc)
  { while (m_bDostepne == true)
    { try { wait(); }
      catch (InterruptedException e) { }
    }
    m_nZawartosc = wartosc;
    m_bDostepne = true;
    notifyAll(); //zawiadomienie Konsumenta
  }
}
```

Pudełko - obiekt z monitorem

- W Javie kazdy obiekt, który ma metody synchroniczne posiada swój monitor.
- Metody wait i notifyAll naleza do klasy java.lang.Object i moga byc wywoływane tylko przez watki, które załozyły blokade.
- Metoda notifyAll informuje wszystkie watki oczekujace na monitor zajety przez biezacy watek o zwolnieniu tego monitora i budzi te watki. Przewaznie jeden z oczekujacych watków zajmuje monitor i wykonuje swoje zadanie
- Metoda wait powoduje zwolnienie posiadanego monitora i przejscie w stan oczekiwania do czasu, az inny watek powiadomi (ang. notify) go o zwolnieniu monitora obiektu.

Producent/Konsument Demo

```
class ProdKonsTest {
  public static void main(String[] args) throws Exception
  {
    Pudelko c = new Pudelko();
    Producent p1 = new Producent(c, 1);
    Konsument c1 = new Konsument(c, 1);
    p1.start();
    c1.start();
    pauza();
  }
  static void pauza() throws java.io.IOException
  {
    System.out.println("Nacisnij␣Enter...");
    System.in.read();
  }
}
```

Programowanie sieciowe w Javie

- Wprowadzenie
- Przegląd klas pakietu java.net
- Reprezentacje adresów
- Połączenia gniazdowe
- Datagramy

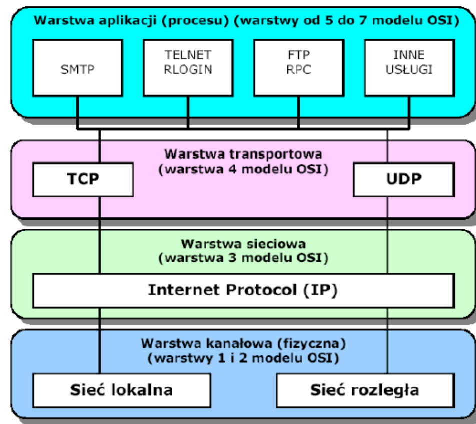
Protokoły sieciowe - podstawy

Protokół sieciowy jest to zbiór procedur oraz reguł rządzących komunikacją, między co najmniej dwoma urządzeniami sieciowymi. Istnieją różne protokoły, lecz nawiązujące w danym momencie połączenie urządzenia muszą używać tego samego protokołu, aby wymiana danych pomiędzy nimi była możliwa.

Podstawowe rodziny protokołów sieciowych:

- NetBEUI (IBM - Microsoft, opracowany w 1985) i IPX/SPX (Novell, lata 70) – obecnie stosowane niezbyt często lub wcale.
- **TCP/IP** – najczęściej stosowany zestaw protokołów sieciowych. Łączą komputery pracujące na różnych platformach sprzętowych i systemowych. Specyfikacje protokołów można znaleźć w dokumentach RFC (Request for Comments)

Teoretyczne modele sieciowe



Wprowadzenie do TCP/IP

- zestaw protokołów komunikacyjnych (RFC 793), którego nazwa pochodzi od dwóch najważniejszych (spośród wielu):
  - kontroli transmisji (Transmission Control Protocol, TCP),
  - protokołu internetowego (Internet Protocol, IP)
- TCP/IP udostępnia metody transmisji informacji pomiędzy poszczególnymi hostami w sieci, obsługując pojawiające się błędy oraz tworząc wymagane do transmisji informacje dodatkowe.
- otwarty standard, dostępny bezpłatnie i stworzony niezależnie od platformy sprzętowej czy programowej co służy integracji różnych sieci,
- IP zapewnia jednolity system adresowania, pozwalający w identyczny sposób zaadresować każde urządzenie w sieci, nawet tak dużej jak Internet,

Transmisja TCP

- Protokół połączeniowy,
- Niezawodne i wiarygodne przesyłanie danych (segmenty są dostarczone do miejsca przeznaczenia w kolejności, w jakiej zostały wysłane, bez uszkodzeń i strat)
- Oparty na sygnale ACK potwierdzenia z retransmisją,
- Kontrola przepływu, korekcja błędów,
- Skierowany na strumieniową (jednolitą) transmisję danych (przesyłanie dużych ilości ciągłych informacji – np. pliki)
- RFC 793.

Przesył UDP (User Datagram Protocol)

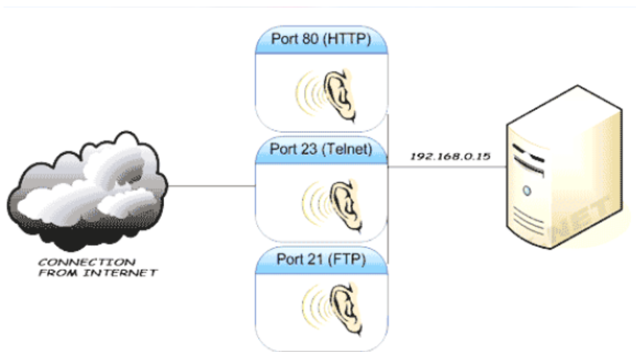
- Protokół bezpołączeniowy, w którym pojedynczy pakiet (datagram) integralną informację.
- Bez narzutu na nawiązywanie połączenia i śledzenie sesji (vide TCP),
- Nie korzysta z mechanizmów kontroli przepływu i retransmisji – zawodny
- Duża szybkość transmisji danych,
- Dla zastosowań gdzie dane muszą być przesyłane możliwie szybko, a poprawianiem błędów mogą zajmować się inne warstwy modelu OSI, np. wideokonferencje, strumienie dźwięku w Internecie, gry sieciowe, itp.
- RFC 768.

Adresy sieciowe

- **adres fizyczny (MAC)** - nadawany przez producentów kart sieciowych i należący do warstwy fizycznej DoD np. `00-50-56-C0-0A-01`,
- **adres domenowy** - łatwiejszy do zapamiętania słowny odpowiednik adresu IP, wykorzystujący serwery DNS i należący do warstwy aplikacji DoD np. `www.issi.uz.zgora.pl`
- **adres IP** - unikatowy identyfikator nadawany komputerowi w sieci IP, należący do warstwy komunikacyjnej DoD i pozwalający na lokalizację komputera w sieci, np. **84.10.191.2**

Port protokołu

- Adres wewnętrzny zapewniający interfejs pomiędzy aplikacjami sieciowymi, a warstwą transportową sieci,
- Numer portu źródłowego procesu, który wysłał dane oraz numer portu docelowego procesu, który ma dane otrzymać, są zawarte w pierwszym słowie nagłówka każdego segmentu TCP i UDP

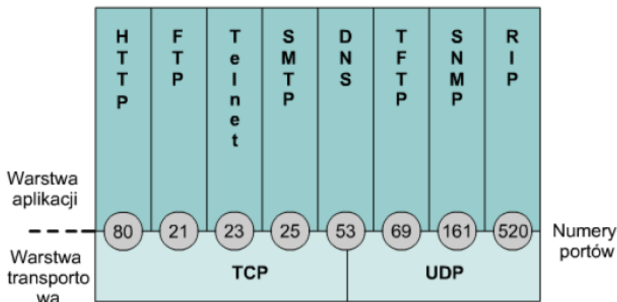


Reprezentacja i wartości portów

- Port reprezentowany jest przez liczbę z zakresu 0-65535 (16 bitów)
- Numery portów mają przydzielone następujące zakresy:
    - numery poniżej 1024 są uważane za dobrze znane numery portów,
    - 1024 – 49151 zarejestrowane porty. Mogą z nich korzystać programy i procesy zwykłych użytkowników,
    - 49152 – 65535 porty dynamiczne i/lub prywatne.
- Numery protokołów i portów przyporządkowane znanym usługom zdefiniowane są w RFC 1700
- Host źródłowy dynamicznie przydziela numery portów źródła rozpoczynającego transmisję. Numery te są zawsze większe od 1023.

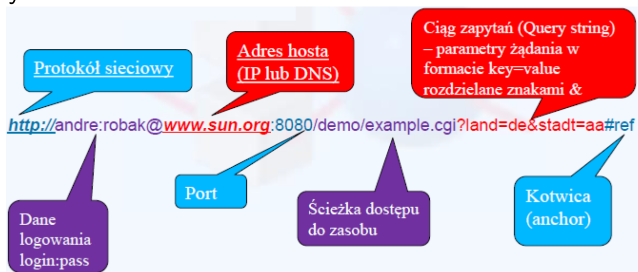
Reprezentacja i wartości portów

- Różne usługi mogą używać tego samego numeru portów, pod warunkiem że korzystają z innego protokołu (TCP lub UDP), chociaż istnieją także usługi korzystające jednocześnie z jednego numeru portu i obu protokołów, np. DNS - korzysta z portu 53 za pomocą TCP i UDP jednocześnie.
- Zdarza się także, że jedna usługa może korzystać z dwóch różnych portów używanych do innych zadań (np. FTP, SNMP).



Adres URL

URL (Uniform Resource Locator) jest adresem zasobów w Internecie opisanym w specyfikacji RFC1738. URL jest używany przez przeglądarki internetowe w celu lokalizacji i dostępu do informacji na stronach WWW. Przykładowy adres zasobu:



Java i programowanie sieciowe

- Javę zaprojektowano do tworzenia programów sieciowych
- Java dostarcza wiele interfejsów niskiego i wysokiego poziomu dostępnych w pakiecie java.net oraz pakietach uzupełniających (np. javax.servlet, javax.smtp, etc.) , m.in.:
  - Klasy reprezentujące adresy
    - InetAddress,
    - URL + URLConnection
  - Klasy połączeń opartych na TCP:
    - Socket
    - ServerSocket
  - Klasy połączeń opartych na UDP:
    - DatagramSocket
    - DatagramPacket

Klasa InetAddress

- reprezentuje istniejący adres (adresy) IP komputera (hosta) w sieci
- nie posiada publicznych konstruktorów - utworzenie obiektu możliwe jest dzięki statycznym metodom fabrycznym (factory methods) :

```
public static InetAddress getByName(String host) throws UnknownHostException
```

tworzy obiekt adresu hosta o podanej nazwie – IP lub DNS (jeśli istnieje)

```
public static InetAddress getByAddress(byte[] addr) throws UnknownHostException
```

tworzy obiekt adresu hosta na podstawie tablicy bajtów np. new byte[] 127, 0, 0, 1

```
public static InetAddress[] getAllByName(String host) throws
UnknownHostException
```

tworzy i inicjalizuje tablicę obiektów adresów hosta o podanej nazwie IP lub DNS

```
public static InetAddress getLocalHost() throws UnknownHostException
```

tworzy obiekt adresu lokalnego hosta

InetAddress – przykład użycia

```
try {
  InetAddress adresIP = InetAddress.getByName("www.issi.uz.zgora.pl");
  System.out.println("Nazwa␣hosta:␣" + adresIP.getHostName());
  System.out.println("Numer␣IP:␣" + adresIP.getHostAddress());
  System.out.println("?Grupowy␣(rozgloszeniowy):␣" + adresIP.isMulticastAddress());
  System.out.println("?Osia␣galne␣Echo:␣" + adresIP.isReachable(10));
  System.out.println("?Prywatny:␣" + adresIP.isSiteLocalAddress());
  System.out.println(adresIP);
} catch (UnknownHostException e) {
    System.err.println(e);
}
```

Klasa URL

- Reprezentuje abstrakcyjny adres zasobu sieciowego.
- Adres URL może składać się z następujących elementów (dostępnych przy użyciu metod dostępowych get):
  - protokół (wymagane) – getProtocol()
  - host (wymagane) – getHost()
  - port – getPort()
  - plik – getFile()
  - identyfikator fragmentu (np. ref, section, anchor) – getRef()
  - ciąg zapytania (query) – getQuery()
  - dane uwierzytelniające – getAuthority(),
- Obiekty tworzone są wyłącznie przy użyciu konstruktorów, ww. elementy adresu nie posiadają metod ustawiających set-

Konstruktory URL

- Konstruktor adresu bezwzględnego (absolute URL):
  `public URL(String spec) throws MalformedURLException np.`
  `URL urlAddress=new URL("http://www.issi.uz.zgora.pl");`
- Konstruktor adresu względnego (relative URL):
  `public URL(URL host, String spec) throws`
  `MalformedURLException np. new`
  `URL(urlAddress,"?action=111&id=6");`
  `new URL(urlAddress, "#bottom");`

Istnieją także konstruktory, umożliwiające podanie kolejnych komponentów adresu URL np. adres :

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.html");
```

jest równoznaczny z:
new URL ("http://www.gamelan.com/pages/Gamelan.html");

oraz

```
new URL("http", "www.gamelan.com", 80,
"pages/Gamelan.html");
```

i reprezentuje on następujący adres URL:

```
http://www.gamelan.com:80/pages/Gamelan.html
```

Każdy z konstruktorów może wyrzucić wyjątek MalformedURLException, gdy przekazane argumenty odnoszą się do złego protokołu:

```
try{
  URL myURL = new URL(...)
}
catch (MalformedURLException e) {
  ... //obsluga wyjatku
}
```

Elementy URL - przykład

```
import java.net.*; import java.io.*;
public class ParseURL {
  public static void main(String[] args) throws Exception {
    URL aURL = new URL("http://java.sun.com:80/docs/
␣␣␣␣+␣"tutorial/index.html#DOWNLOADING");
␣␣␣␣System.out.println("protocol = "␣+aURL.getProtocol());
␣␣␣␣System.out.println("host = "␣+␣aURL.getHost());
␣␣␣␣System.out.println("filename = "␣+␣aURL.getFile());
␣␣␣␣System.out.println("port = "␣+␣aURL.getPort());
␣␣␣␣System.out.println("ref = "␣+␣aURL.getRef());␣}}
```

Powyższy kod zwróci następujące wyniki:

protocol = http

host = java.sun.com

filename = /docs /tutorial/index.html

port = 80

ref = DOWNLOADING

URL – odczyt z zasobu sieciowego

Obiekt URL udostępnia obiekt binarnego strumienia wejściowego (metoda openStream() otwiera połączenie do zasobu).

```
import java.net.*;
import java.io.*;
public class URLReader{
  public static void main(String[] args) throws Exception{
    URL myURL = new URL("http://www.issi.uz.zgora.pl/");
    BufferedReader in = new BufferedReader(new
    InputStreamReader(myURL.openStream()));
    String inpuLine;
    while((inputLine= i.readLine())!=null)
      System.out.println(inputLine);
    in.close();
  }
}
```

Klasa URLConnection

Abstrakcyjna klasa URLConnection zapewnia metody komunikacji z zasobem URL - przede wszystkim pozwalające pobrać strumienie do zapisu i odczytu.
Podklasy abstrakcyjne: HttpURLConnection, JarURLConnection

Obiekt URLConnection tworzony jest metodą openConnection() klasy URL, której użycie inicjuje połączenie komunikacyjne (HTTP lub JAR). Gdy połączenie jest niemożliwe zwracany jest wyjątek IOException.

```
try {
  URL myURL = new URL("http://www.issi.zgora.pl␣/");
  URLConnection urlConnection = myURL.openConnection();
} catch (MalformedURLException e) { // blad tworzenia obiektu
  . . .
} catch (IOException e) { // blad metody openConnection()
  . . .
}
```

URLConnection - odczyt

Poniższy program wykonuje analogiczne zadania jak wcześniej omówiony odczyt z obiektu URL.

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
  public static void main(String[] args) throws Exception {
    URL myURL = new URL("http://www.issi.uz.zgora.pl/");
    URLConnection urlConn = myURL.openConnection();
    BufferedReader in = new BufferedReader(
    new InputStreamReader(
    urlConn.getInputStream()));
    String inputLine;
    while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);
    in.close();
}}
```

URLConnection - zapis

- Aplikacje internetowe zawierają formularze- pola tekstowe i inne elementu GUI, umożliwiające użytkownikowi wprowadzić dane i wysłać je na serwer. Po wypełnieniu formularza i potwierdzeniu wysłania przeglądarka wysyła dane do URL. Następnie dane te są przetwarzane po stronie serwera przez aplikacje (skrypty CGI, serwlety), w wyniku czego użytkownik otrzymuje odpowiedź w formie nowej strony.
- Programy Javy mogą współpracować ze skryptami CGI lub serwletami po stronie serwera poprzez zapis informacji do strumienia wyjściowego dostępnego z obiektu URLConnection.

Zapis do URLConnection - schemat

1. Utworzenie obiektu URL;
2. Otwarcie połączenia URLConnection;
3. Ustawienie w tryb zapisu;
4. Pobranie strumienia wyjściowego z połączenia. Jest on zazwyczaj połączony ze strumieniem wejściowym skryptu CGI, serwletu bądź strony PHP na serwerze;
5. Zapisanie do strumienia wyjściowego;
6. Zamknięcie strumienia wyjściowego.

URLConnection - zapis

```
public class Reverse {
  public static void main(String[] args) throws Exception {
    try {
      String stringToReverse = "ala␣ma␣kota";
      URL url =
        new URL("https://staff.uz.zgora.pl/aczajkow/backwards.php");
      URLConnection connection = url.openConnection();
      connection.setDoOutput(true);
      OutputStreamWriter out = new OutputStreamWriter(
        connection.getOutputStream());
      out.write("string=" + stringToReverse);
      out.close();
      BufferedReader in = new BufferedReader(new InputStreamReader(
        connection.getInputStream()));
      String decodedString;
      while ((decodedString = in.readLine()) != null) {
        System.out.println(decodedString);
      }
      in.close();
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Gniazdo (socket)

- URL i URLConnection dostarczają interfejs sieciowy względnie wysokiego poziomu. Czasami wymagany jest niższy poziom komunikacji sieciowej, np. połączenia klient-serwer.
- Gniazdo jest zdefiniowanym programowo jednym z końców dwu-kierunkowego połączenia pomiędzy dwoma programami pracującymi w sieci.
- Stanowi kombinację adresu IP oraz numeru portu (terminy gniazdo i numer portu często używane są zamiennie),
- Gniazdo jednoznacznie określa każdy program sieciowy w Internecie.

Rozróżniamy:

- Gniazda potokowe (TCP)
- Gniazda datagramowe (UDP)

Gniazdo (socket)

Klient wysyła w sieci TCP/IP żądanie połączenia z innym węzłem, przekazując numer gniazda (portu). Jeśli węzeł odbiorcy może przyjąć połączenie, zwraca numer gniazda (nowo tworzonego) zawierający adres IP odbiorcy i numer portu usługi, która będzie obsługiwać zadanie.
Zgodnie z RFC 793 połączenie gniazdowe definiuje para gniazd czyli czwórka:
(remoteAddress, remotePort, localAddress, localPort).

Połączenie klient - serwer

- Serwer świadczy usługi lub udostępnia zasoby czekając i nasłuchując na określonym porcie.
- Klient natomiast jest stroną żądającą dostępu do danej usługi lub zasobu. Klient musi znać adres gniazda serwera.
- Klient i serwer mogą pracować na tym samym komputerze, używając mechanizmów komunikacji lokalnej, lub na różnych komputerach, wykorzystując komunikację przez sieć.

1. Proces serwera tworzy gniazdo nasłuchiwania klientów
2. Proces klienta tworzy gniazdo klienta do połączenia
3. Klient inicjuje połączenie i przesyła dane swojego gniazda
4. Serwer akceptuje żądanie i tworzy nowe gniazdo do komunikacji z klientem, uwalniając gniazdo nasłuchu dla innych połączeń
5. Klient i serwer komunikują się z użyciem ustalonego protokołu

Implementacja klienta – zadania

1. utworzenie gniazda;
2. otwarcie strumieni IO dla gniazda;
3. odczyt i zapis z/do strumieni zgodnie z wykorzystywanym protokołem przez serwer (komunikacja synchroniczna lub asynchroniczna);
4. zamknięcie strumieni;
5. zamknięcie gniazda.

Implementacja klienta – klasa Socket

- konstruktor służy do nawiązywania połączenia - należy obowiązkowo podać stację zdalną i jej port;
  - `public Socket(String host, int port) throws UnknownHostException, IOException`
  - `public Socket(InetAddress address, int port) throws IOException`
  - `public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException`
- konstruktor pozwala wskazać także adres i port lokalny z którego będziemy się łączyć, ale uwaga - w danej chwili na jednym porcie może być realizowane tylko jedno połączenie
- obiekt Socket dostarcza referencje do podstawowych strumieni binarnych, które można dowolnie opakowywać (np. DataInputStream, ObjectOutputStream, BufferedReader, etc.)

Implementacja gniazda - wyjątki

```
try {
  Socket gniazdo = new Socket("www.issi.uz.zgora.pl", 21);
  System.out.println("Nawiazano polaczenie");
  gniazdo.close();
}
catch (UnknownHostException e) {
  System.out.println("komputer nie jest znany");
}
catch (NoRouteToHostException e) {
  System.out.println("komputer jest niedostepny, firewall");
}
catch (ConnectException e) {
  System.out.println("komputer odrzucil polaczenie/nie nasluchuje na porcie");
}
catch (PortUnreachableException e) {
  System.out.println("nieosiagalny port");
}
catch (IOException e) {
  System.out.println("Wystapil blad IO"+e);
}
```

- BindException – próba utworzenia obiektu Socket lub ServerSocket na używanym porcie lokalnym lub brak uprawnień;
- ConnectException – zdalna stacja odmówiła połączenia np. z powodu zajętości stacji lub braku procesu nasłuchu na wskazanym porcie;
- NoRouteToHostException – połączenie przekroczyło przydzielony czas;
- SecurityException – wyjątek związany z próbą wykonania akcji naruszającej bezpieczeństwo;
- InterruptedIOException – wystąpienie „timeout'u" po ustawienie czasu oczekiwania.

Połączenia - przykład

```
import java.net.*;
import java.io.*;
public class Skaner {
  public static void main(String[] args) {
    for (int port = 1; port < 65536; port++) {
      try {
        Socket s = new Socket("localhost", port);
        System.out.println("otwarty␣port:"+port);
        s.close();
      } catch (IOException e) {
        System.out.println("zamkniety:"+port);
      }
}}}
```

Tworzenie strumieni IO z wykorzystaniem obiektu Socket

```
out = new PrintWriter(echoSocket.getOutputStream(), true);
in = new BufferedReader(new
InputStreamReader(echoSocket.getInputStream()));
```

Implementacja gniazd serwera

- serwer działa w nasłuchu na konkretnym porcie;
- nie jest znany adres klienta
- gniazdo nasłuchu implementowane jest przez obiekt klasy ServerSocket, która dostarcza:
  - konstruktory obiektów;
  - metody do nasłuchu oraz przyjęcia przychodzącego połączenia;
  - metody parametryzujące działanie np. podstawowy czas życia serwera.

Schemat implementacji serwera

1. utworzenie obiektu ServerSocket, związanego z konkretnym portem na lokalnej stacji;
2. ServerSocket nasłuchuje na połączenia za pomocą metody accept(); metoda ta blokuje się do wystąpienia próby połączenia; wówczas accept() zwraca obiekt Socket, łączący serwer z klientami;
3. W zależności od typu serwera, są wywoływane odpowiednie metody do utworzenia strumieni wejściowych i wyjściowych – komunikacja odbywa się już na zwykłych Socketach
4. Serwer i klient działają w oparciu o ustalony protokół aż do momentu zamknięcia połączenia;
5. Połączenie zamyka serwer, klient lub obie strony jednocześnie
6. Serwer wraca do pozycji 2.

Konstruktory ServerSocket

```
public ServerSocket(int port)
public ServerSocket(int port, int queue)
public ServerSocket(int port, int queue, InetAddress
bindAddr)
```

- tworzą obiekty gniazd nasłuchujących związane ze wskazanym portem lokalnego komputera;
- port=0 oznacza, że wybór portu należy do systemu (port anonimowy);
- mogą generować IOException, BindException;
- queue określa długość kolejki pamiętanych połączeń oczekujących; system operacyjny kolejkuje (FIFO) połączenia dla każdego portu, po zapełnieniu kolejki kolejne są odrzucane, chyba że zwolni się miejsce,
- konstruktor z argumentem bindAddr oznacza, że serwer jest dowiązany (binding) do danego portu; konstruktor przydatny dla hostów posiadających wiele adresów IP (interfejsów), pozwalający wskazać konkretny interfejs, do którego może się podłączyć klient.

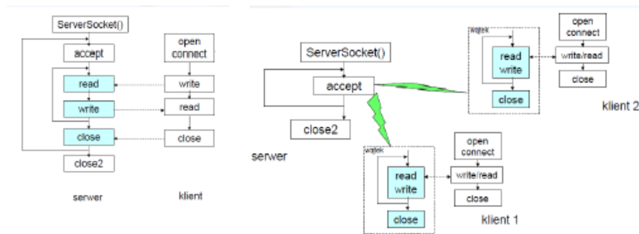
Serwer akceptuje połączenie poprzez zastosowanie metody accept():

- blokuje ona wątek serwera w oczekiwaniu na połączenie;
- po połączeniu metoda zwraca obiekt Socket, który służy do komunikacji.

```
try{ ServerSocket s = new ServerSocket(8877);
  while (true){
    Socket conn = s.accept();
    PrintStream p = new PrintStream (conn.getOutputStream());
    p.println("Polaczyles␣sie␣z␣serwerem."+"BYE");
    conn.close();
  }
} catch(IOException){....}
```

Schematy obsługi klienta

- iteracyjny – proces serwera zajmuje się obsługą jednego klienta, stosowany jest gdy czas obsługi klienta jest krótki lub kiedy używa go tylko jeden klient.
- współbieżny – dla obsługi klienta, proces serwera uruchamia nowy wątek



Datagramy

Istnieją dwa rodzaje konstruktorów klasy DatagramPacket: do odbioru:

```
public DatagramPacket(byte[] buf, int offset, int length)
public DatagramPacket(byte[] buf, int length)
```

buf – bufor do odbioru danych;
offset – początek bufora buf;
length – max. długość pakietu;

do nadawania:

```
public DatagramPacket(byte[] buf, int offset, int length,
InetAddress address, int port)
public DatagramPacket(byte[] buf, int length, InetAddress
address, int port)
```

buf – bufor z pakietem UDP;
offset – początek bufora buf;
length – długość danych;
address – adres danych;
port – numer portu adresata danych.

Datagram może mieć maksymalnie 65.535 bajtów, w tym 8 bajtów na nagłówek UDP i min 20 bajtów na nagłówek IP. Zastosowanie większych pakietów wiąże się ze wzrostem efektywności transferu, mniejsze datagramy są natomiast bardziej niezawodne.
Nagłówek UDP składa się z
:

- 2 bajty – unsigned int – numer portu źródłowego;
- 2 bajty – numer portu docelowego;
- trzecia para bajtów – rozmiar datagramu razem z nagłówkiem;
- ostatnia para – suma kontrolna.

Wysyłanie i odbiór datagramów:

- `public void send(DatagramPacket p)`
- `public void receive(DatagramPacket p)` – odbiera z sieci datagram UDP i konwertuje go do postaci obiektu DatagramPacket

Przykład wysyłania datagramu

```java
public static void main(String[] args) {
  try {
    InetAddress host = InetAddress.getByName("127.0.0.1");
    int port = 250;
    byte[] b = "Hello".getBytes();
    DatagramPacket dp =
    new DatagramPacket(b, b.length, host, port);
    DatagramSocket ds = new DatagramSocket();
    ds.send(dp);
  } catch (UnknownHostException ex) {
    System.err.println(ex);
  }
  catch (IOException e) {
    System.out.println(e);
  }
}
```

Przykład odbierania datagramu

```
public static void main(String[] args) {
  try{
    DatagramSocket ds = new DatagramSocket(250);
    byte buf[] = new byte[10];
    DatagramPacket dp = new DatagramPacket(buf,buf.length);
    ds.receive(dp);
    byte[] data = dp.getData();
    String s = new String(data, 0, data.length);
    System.out.println(s);
    System.out.println(dp.getPort());
  } catch(IOException e){
    System.err.println(e);
  }
}
```

Podsumowanie TCP vs UDP

- TCP posiada oddzielną klasę do nasłuchu (serwera);
- UDP nie stosuje pojęcia gniazda serwera - to samo gniazdo może przyjmować połączenia i wysyłać dane;
- TCP wysyła dane przez strumień skojarzony z gniazdem; a UDP nie stosuje strumieni, wysyła pakiety (datagramy);
- Konkretne gniazdo typu DatagramSocket może odbierać dane od wielu niezależnych stacji i nie jest dedykowane jednemu konkretnemu połączeniu,
- Metody strumieni łącza: read i write (TCP) mogą blokować swój wątek, np. przy zapisie write do strumienia wyjściowego gdy zostanie już osiągnięty rozmiar bufora gniazda, a odbiorca nie odbiera danych (domyślnie bufor gniazda ma poj. 8KB, ale klasa Socket pozwala ustawić inny).
- W UDP metoda zapisu nie blokuje bieżącego wątku, ale pakiety mogą zostać utracone gdy wystąpi awaria sieci

Java Foundation Classes

Java Foundation Classes do tworzenia interfejsu użytkownika programu napisanego w Javie zawiera dwa podzbiory obiektów

- AWT
- Swing

Ponadto JFC obsługuje:

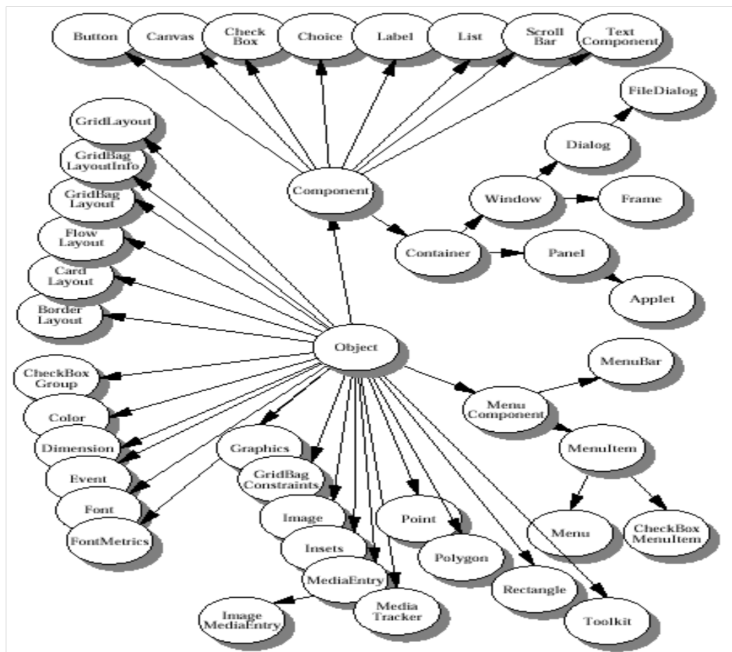
- delegacyjny model zdarzeń
- wymianę danych przy pomocy schowka
- udoskonalone powiązanie z kolorami systemu
- definiowanie skrótów klawiszowych
- działania "drag-and-drop"
- technologie ułatwień dla niepełnosprawnych

Abstract Window Toolkit - AWT

- jeden z podstawowych zbiorów klas zapewniających użytkownikowi możliwość tworzenia prostego interfejsu graficznego
- wszystkie komponenty pochodzą od abstrakcyjnej klasy **Component**

AWT: Pros and Cons

- Pros
  - Speed: native components speed performance.
  - Look and feel: AWT components more closely reflect the look and feel of the OS they run on.
- Cons
  - Portability: use of native peers creates platform specific limitations.
  - Features: AWT supports only the lowest common denominator—e.g. no tool tips or icons.


Button
Canvas
Check Box
Choice
Label
List
Scroll Bar
Text Component
FileDialog
GridLayout
GridBag LayoutInfo
Dialog
GridBag Layout
Component
Window
Frame
Flow Layout
Container
Panel
Card Layout
Applet
Border Layout
Object
MenuBar
CheckBox Group
Menu Component
Color
Dimension
Graphics
MenuItem
Event
GridBag Constraints
Font
Image
Point
Menu
CheckBox MenuItem
FontMetrics
Insets
Polygon
MediaEntry
Rectangle
Image MediaEntry
Media Tracker
Toolkit

Abstract Window Toolkit – AWT

- abstrakcyjna klasa `Component` definiuje grupę metod wspólnych dla wszystkich komponentów AWT
- rozróżniamy dwa rodzaje komponentów: zwykłe (*terminalne*) oraz mogące zawierać inne komponenty (tzw. *kontenery*)
- obsługa poszczególnych komponentów jest prosta - zadajemy właściwości komponentu oraz jego reakcje na generowane zdarzenia.

Wspólne właściwości komponentów

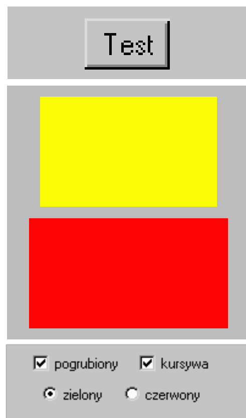
- Size - rozmiar
  - pobieranie getSize() getSize(Dimension rozmiar)
  - ustalanie setSize(int szerokosc, int wysokosc)
- Location - położenie
  - pobieranie getLocation() getLocation(Point p)
  - ustalanie setLocation(Point p)
- Bounds - rozmiar i położenie
  - pobieranie getBounds() getBounds(Rectangle prost)
  - ustalanie setBounds(int x, int y, int szerokosc, int wysokosc) setBounds(Rectangle prost)
- Font - czcionka
  - pobieranie getFont()
  - ustalanie setFont(Font f)
- Background - kolor tła
  - pobieranie getBackground()
  - ustalanie setBackground(Color c)

Wspólne właściwości komponentów

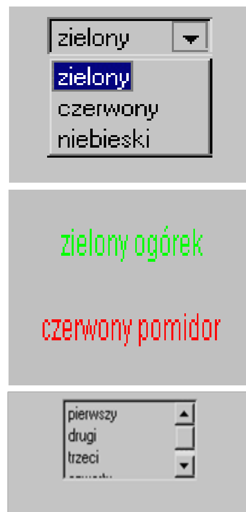
- Foreground - kolor pierwszego planu
  - pobieranie getForeground()
  - ustalanie setForeground(Color c)
- Visible - widzialność
  - pobieranie isVisible()
  - ustalanie setVisible(boolean logiczna)
- Enabled - dostępność
  - pobieranie isEnabled()
  - ustalanie setEnabled(boolean logiczna)

Przegląd komponentów

- `Button` – reprezentuje przycisk,
- `Canvas` – reprezentuje prostokątny obszar ekranu, na którym program może rysować lub z którego może przechwytywać zdarzenia, aby w sposób sensowny wykorzystać ten komponent (np. przy tworzeniu nowych komponentów) powinniśmy tworzyć klasy potomne, w których nadpisując metodę paint() określimy własną zawartość płótna
- `Checkbox` – wykorzystywany, gdy wymagane jest dopuszczenie możliwość wyboru, możliwe jest również tworzenie grup opcji (wzajemnie się wykluczających), korzystamy wówczas z klasy `CheckboxGroup`.



- Choice – jeśli w programie pragniemy dopuścić możliwość wyboru spośród większej liczby możliwości, wygodniej jest użyć listę rozwijalną, można ją również traktować jak swego rodzaju menu - wybrana opcja jest tytułem menu.
- Label – wykorzystywany do opisywania innych elementów okna, tekst na nim wyświetlany możemy wyrównywać za pomocą odpowiednich stałych
- List – komponent prezentujący przewijalną listę elementów do wyboru, można pozwolić wybierać jedynie jeden element naraz lub dopuścić równoczesne zaznaczanie wielu elementów.



- Scrollbar – stosowany jeśli pragniemy umożliwić użytkownikowi wybieranie wartości z pewnego zakresu, możemy również wykorzystać go do ilustrowania przewijania danych, postępu pracu, itp.
- TextArea – komponent jest to wielowierszowy obszar do wyświetlania tekstu.
- TextField – pozwala tworzyć pola edycyjne o podanej liczbie widocznych kolumn tekstu, jest to wygodny sposób komunikacji z użytkownikiem programu - miejsce gdzie może on podać własne parametry, dane do programu.

JFC – Kontenery

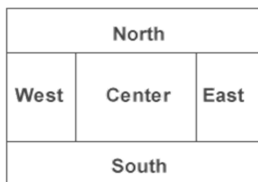
- komponenty, które mogą zawierać inne komponenty,
- rozkładem komponentów steruje Menedżer Rozkładu,
- standardowo dostępnych jest 5 głównych rozkładów: FlowLayout, GridLayout, BorderLayout, CardLayout, GridBagLayout,
- rozkłady są podstawowym sposobem rozmieszczania komponentów,
- można zrezygnować z menedżera rozkładu na rzecz ręcznego rozmieszczenia i ustawienia wielkości komponentów (rozkład null).

Rozkłady

- GridLayout
  - obszar kontenera dzielony jest na siatkę (o jednakowych oczkach), każdy komponent dostaje jeden obszar i wypełnia go całkowicie
  - komponenty dodawane są do kolejnych oczek siatki od lewej do prawej
  - przy zmianie wielkości obszaru kontenera zmienia się rozmiar oczek siatki
- FlowLayout
  - stosowany domyślnie przez kontener Panel
  - komponenty są rozkładane od lewej do prawej, od góry do dołu
  - komponenty mają swoje standardowe rozmiary
  - przy zmianie wielkości kontenera komponenty będą się przesuwać

Rozkłady

- BorderLayout
  - stosowany domyślnie przez kontener Window
  - obszar dzielony jest na pięć pól zgodnie z kierunkami:



- CardLayout
  - wyświetla na raz tylko jeden komponent
  - przejście między komponentami uzyskuje się przez wywołanie metody next(),previous(), first(), last (), show()
  - wyświetlany komponent wypełnia cały dostępny obszar

Rozkłady

- GridBagLayout
  - każdy komponent ma narzucone ograniczenia GridBagConstraints
  - można zadać mu wymagania rozmiaru i położenia względem innych komponentów oraz kontenera
  - dostępne są pola:
    - fill NONE, BOTH, HORIZONTAL, VERTICAL - jak może wzrastać rozmiar komponentu
    - gridwidth - określa liczbę obszarów w wierszu,
    - wartość REMINDER określa, że ma to być ostatni komponent w wierszu
- null – programista sam definiuje rozkład i rozmieszczenie poszczególnych elementów

Przegląd kontenerów

- Window – okna jakie uzyskujemy korzystając bezpośrednio z tej klasy:
  - nie posiadają ramki, menu i tytułu,
  - mają inne okno jako właściciela,
  - nie są przesuwane wraz z właścicielem,
  - nie są modalne.
- Dialog – za pomocą tej klasy oraz jej klas pochodnych możemy tworzyć różnego typu okienka dialogowe do komunikacji z użytkownikiem,
  - Dialog jest oknem posiadającym ramkę oraz tytuł, ma właściciela i nie może zawierać menu,
  - za pomocą tej klasy możemy tworzyć okna modalne i niemodalne,
  - właścicielem okna dialogowego może być inne okno dialogowe lub ramka.

Przegląd kontenerów

- FileDialog
  - okno dialogowe do wyboru plików,
  - pochodzi od klasy Dialog,
  - przykład okna modalnego odwzorowującego standardowe okno dialogowe wyboru plików platformy.
- Frame
  - główne okno tworzonych przez nas aplikacji jest zawsze obiektem klasy pochodnej od Frame,
  - okna ramowe Frame mają ramkę, pasek tytułowy, ikony sterujące, mogą również posiadać pasek menu,
  - nie mają natomiast właściciela oraz, podobnie jak klasa Window, nie mogą tworzyć okien modalnych.
- Panel
  - jest wykorzystywana przede wszystkim do grupowania elementów,
  - Panele są umieszczane w innych kontenerach,

Komponenty ciężkie i lekkie

- ciężkie (np. „gotowe” do użycia Menu, ScrollPane oraz pochodzące od Canvas i Panel)
  - związane są z równorzędnymi obiektami systemu
  - ich wygląd zależy od systemu
  - mają nieprzezroczyste okno
  - każdy komponent wywodzący się z komponentu ciężkiego jest ciężki
  - kombinowany z lekkim, znajdzie się zawsze na górze
- lekkie
  - wprowadzone w JDK 1.1
  - wywodzą się z abstrakcyjnych klas Component lub Container
  - ich wygląd zależy wyłącznie od programisty
- komponenty lekkie można dowolnie łączyć z ciężkimi
- ciężki kontener może zawierać lekkie komponenty i odwrotnie

Zdarzenia

Wraz z wykonywanymi poleceniami, akcjami ze strony użytkownika, procedurami systemu generowane są zdarzenia (wywodzące się z klasy awt.event):

- niskiego poziomu - związane są z fizycznymi aspektami interfejsu (kliknięcie myszy, wejście kursora na dany obszar...)
- semantyczne - są sekwencjami zdarzeń niskiego poziomu (wybranie konkretnej pozycji menu...).

Zdarzenia

Pochodne klasy `java.awt.event.ComponentEvent` to zdarzenia niskiego poziomu:

- **ContainerEvent** - wysyłane po dodaniu i usunięciu komponentu
- **FocusEvent** - uzyskanie lub strata sterowania
- **InputEvent** (KeyEvent, MouseEvent) - związany z wejściem dla GUI, klawiaturą i myszką.
- **WindowEvent** - użytkownik wykonał jedno z poleceń systemu operacyjnego (np. zminimalizował okno)
- **PaintEvent** - służy do serializacji zdarzeń malowania komponentu. Nie powinno być używane (stosuje się przeciążanie metody paint).

Zdarzenia

Zdarzenia semantyczne:

- **ActionEvent** - powiadamia o działaniach specyficznych dla komponentu
- **AdjustmentEvent** - zmiana stanu paska przewijania
- **ItemEvent** - informuje, że użytkownik wykonał działanie na liście lub na polu wyboru
- **TextEvent** - użytkownik zmienił tekst (dotyczy TextArea, TextField)

przykład

```
import java.awt.*;
import java.awt.event.*;

public class AWTGUI extends Frame{
  public AWTGUI() {
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        dispose();
        System.exit(0);
          }});
          add( new Label("Java!") );
        }
    public static void main(String args[]) {
          AWTGUI f = new AWTGUI();
          f.setSize(100,100);
          f.setVisible(true);
   }
}
```

Pojemnikowy model obsługi zdarzeń

- zdarzenia są przekazywane do metod action() i handleEvent() komponentu
- aby je obsługiwać należy je przesłonić w podklasie komponentu
- zwracając wartość true można zapobiec dalszemu przekazywaniu zdarzenia
- minusy:
  - mnogość tworzonych komponentów
  - do metod obsługi trafiają wszystkie zdarzenia, co wymaga tworzenia skomplikowanego systemu ich rozpoznawania wewnątrz procedury
  - obsługa zdarzenia jest ściśle powiązana z obiektem GUI
  - brak filtrowania zdarzeń
- nie polecany i w zasadzie nie używany

Delegacyjny model obsługi zdarzeń

- pozwala oddzielić komponent GUI od obsługi zdarzenia
- zdarzenie jest przekazywane od źródła (komponentu) do procedury obsługi (słuchacza/listener)
- słuchacz otrzymuje jedynie zdarzenia, które go interesują (pozostałe są filtrowane)
- komponent generuje tylko specyficzne dla siebie zdarzenia i może mieć tylko słuchaczy określonych typów
- wydelegowanie obiektu do obsługi zdarzenia
  `component.addKindOfListener( objKindOfListener )`
- obiekt nasłuchujący musi implementować interfejs KindOfListener
- każdy komponent udostępnia metody rejestrujące słuchaczy dla właściwych dla siebie zdarzeń
- klasy nasłuchujące mogą być w klasie głównej, zewnętrzne, wewnętrzne i anonimowe

Klasa główna jako nasłuchująca

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyFrame extends Frame implements ActionListener {
  private JButton button_ok = new JButton ("Ok");
  public MyFrame(){
    button_ok.addActionListener(this);
  }
  // przeladowanie wywolywanej metody
  public void actionPerformed(ActionEvent e){...}
}
```

Wada: niedobre do przechwytywanie zdarzeń z wielu obiektów i obsługi ich w różny sposób.

Wewnętrzna klasa nasłuchująca

```
// import jak wczesniej
public class MyFrame extends Frame{

  private JTextField barcode = new JTextField(15);

  public MyFrame(){
    button_ok.addActionListener( new MyListener() );
  }

  class MyListener implements ActionListener{
    public void actionPerformed(ActionEvent e){...}
  }
}
```

Zewnętrzna klasa nasłuchująca

```
class MyListener implements ActionListener {
  MyListener(Frame owner){
    this.owner=owner;
  }
 private Frame owner;
 // przeladowanie wywolywanej metody
 public void actionPerformed(ActionEvent e){...}
}

public class MyFrame extends Frame{
  private JButton button_ok = new JButton ("Ok");
  public elementy_sterowane;
  public MyFrame(){
    button_ok.addActionListener(new MyListener(this));
  }
}
```

Anonimowa klasa nasłuchująca

```
button_ok.addActionListener( new ActionListener(){
  // przeladowanie wywolywanej metody
  public void actionPerformed(ActionEvent e){... }
});
```

Nie pojawia się słowo kluczowe implements. W momencie rejestracji zdarzenia powołujemy do życia anonimowy obiekt i w tym miejscu w kodzie implementujemy metodę ActionPerformed.

Tabela interfejsów obiektów nasłuchujących i metod rejestrujących

Zdarzenie	interfejs metody	rejestracja komponenty
ComponentEvent	ComponentListener componentHidden componentMoved componentResized componentShown	addComponentListener Component
ContainerEvent	ContainerListener containerAdded containerRemoved	addContainerListener Container
WindowEvent	WindowListener windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened	addWindowListener Frame Dialog

Tabela interfejsów obiektów nasłuchujących i metod rejestrujących

FocusEvent	FocusListener focusGained focusLost	addFocusListener Component
KeyEvent	KeyListener keyPressed keyReleased keyTyped	addKeyListener Component
MouseEvent	MouseListener mouseClicked mouseEntered mouseExited mousePressed mouseReleased MouseMotionListener mouseDragged mouseMoved	addMouseListener Component addMouseMotionListener Component
ActionEvent	ActionListener actionPerformed	addActionListener Button List TextField MenuItem

Tabela interfejsów obiektów nasłuchujących i metod rejestrujących

AdjustmentEvent	AdjustmentListener adjustmentValueChanged	addAdjustmentListener Scrollbar
ItemEvent	ItemListener itemStateChanged	addItemListener Choice List Checkbox CheckboxMenuItem
TextEvent	TextListener textValueChanged	addTextListener TextField TextArea

Klasy adaptacyjne

- często istnieje potrzeba obsłużenia tylko jednej akcji związanej z danym zdarzeniem
- jest to uciążliwe jeśli dany interfejs wymaga zaimplementowania wielu metod (jak np. `WindowListener`)
- AWT udostępnia klasy adaptacyjne, które definiują puste metody dla każdego z interfejsów
- są zdefiniowane tylko dla zdarzeń niskiego poziomu
- tworząc podklasę klasy adaptacyjnej wystarczy przeciążyć tylko potrzebną metodę

Tabela klas adaptacyjnych

interfejs	klasa adaptacyjna
ComponentListener	ComponentAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

```
public class DynFrame extends Frame {
  int counter = 0;
  Frame mainFrame = null;
  Menu buttonMenu = null;
  DynMenuButton focus = null;

  public static void main(String args[]) {
    DynFrame myApp = new DynFrame();
    myApp.setSize(300, 300);
    myApp.setVisible(true);;
  }

  public DynFrame() {
    super("Main␣Frame");
    mainFrame = this;
    buttonMenu = new Menu("Button");
    MenuBar mb = new MenuBar();
    Menu mainMenu = new Menu("Main");
    mainMenu.add("New");
    mainMenu.addSeparator();
    mainMenu.add("Remove");

    mb.add(mainMenu);
    mb.add(buttonMenu);
```

```
    mainMenu.addActionListener( new MenuHandler() );
    setMenuBar(mb);
    setLayout(new FlowLayout());
    addWindowListener( new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);
}});}
class MenuHandler implements ActionListener{
  public void actionPerformed(ActionEvent event) {
    String s = event.getActionCommand();
    if( "New".equals(s)) {
      counter++;
      CheckboxMenuItem enabled = new CheckboxMenuItem("Enabled" + counter, true);
      DynMenuButton button = new DynMenuButton(counter+"", enabled);
      buttonMenu.add(enabled);
      enabled.addItemListener(button);
      add(button);
    } else if("Remove".equals( s )) {
        if(focus != null)     {
          buttonMenu.remove( focus.menuOption );
          remove(focus);
          focus = null;
      }
    }
    validate();
  }
}
```

```
class DynMenuButton extends Button implements ItemListener, ActionListener, Focu
  MenuItem menuOption;
  DynMenuButton (String caption, MenuItem menuOption) {
    super(caption);
    addActionListener(this);
    addFocusListener(this);
    this.menuOption = menuOption;
  }
  public void itemStateChanged(ItemEvent event){
    CheckboxMenuItem enabled = (CheckboxMenuItem)event.getItemSelectable();
    setEnabled(enabled.getState());
    validate();
  }
  public void actionPerformed(ActionEvent event){
    Dialog dialog = new Dialog(mainFrame, "Przycisk");
    dialog.add( "Center", new Label( "Button␣:␣" + getLabel() ));
    dialog.setSize(100, 50);
    dialog.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent event) {
        event.getWindow().dispose();
    }});
    dialog.setVisible(true);;
  }
  public void focusGained(FocusEvent event) {
    focus = (DynMenuButton)event.getComponent();
  }
  public void focusLost(FocusEvent event){}
 }
}
```

JFC - Projekt Swing

- posiada ponad dwa razy więcej podstawowych komponentów niż AWT
- nazwy podstawowych komponentów zaczynają się od J...
- większość komponentów jest kompatybilna z AWT, to znaczy, że można ich używać dokładnie tak samo jak komponenty AWT
- pozwala na formatowanie za pomocą HTML np:

```
button = new JButton("<html><b><u>T</u>wo</b><br>lines</html>");
```

JFC - Projekt Swing

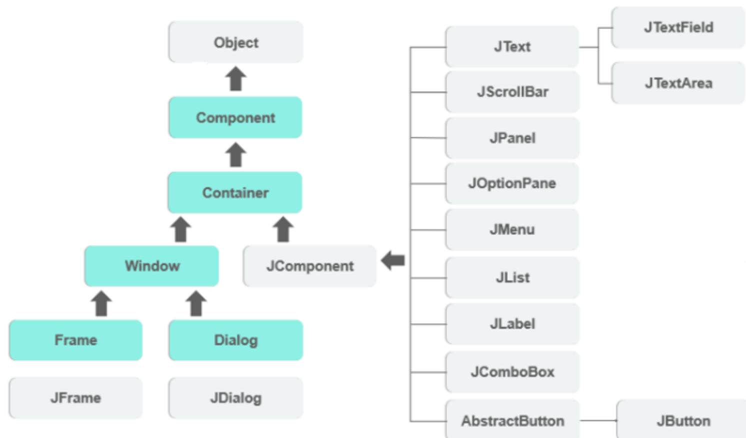
Swing vs AWT

Swing	AWT
OS independent	OS dedpendent
Light weight	Heavy weight
base on Write once use anywhere	Not consistent as compared to Swing
feel and look	change behaviour due to os
rich set of object	less as compared to swing

Przegląd wybranych pakietów Swing

- javax.swing - zbiór podstawowych obiektów
- javax.swing.accessibility - pozwala na pełną współpracę aplikacji z systemem wspomagania dla niepełnosprawnych
- javax.swing.border - definiuje style obrazowania krawędzi i pozwala je dowolnie modyfikować
- javax.swing.event - zdarzenia i słuchacze
- javax.swing.plaf - interfejs do definiowania własnych wyglądów (w jego skład wchodzą obecnie metal, motif, basic)
- javax.swing.table - udostępnia klasę JTable do prezentacji danych
- javax.swing.text - pomaga w manipulacji na tekstach w tym wycinanie i wstawianie
- javax.swing.text.html - wprowadza klasy do edycji i manipulacji na tekstach w HTML
- javax.swing.text.rtf - definiuje klasy do pracy na tekstach formatowanych w RTF
- javax.swing.tree - pozwala na definiowanie drzew i wpływanie na ich własności
- javax.swing.undo - dla programistów chcących wyposażyć aplikacje w możliwości undo/redo

Swing



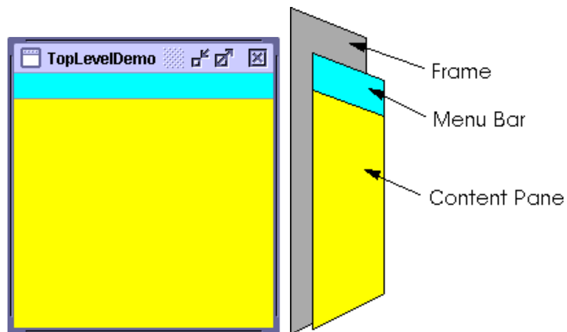
klasa JComponent

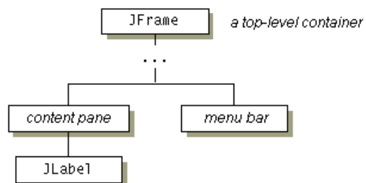
- baza dla komponentów Swing
- wymienny wygląd
- projektowanie nowych komponentów
- skróty klawiszowe
- definiowalna ramka
- możliwość ustawienia preferowanego, minimalnego i maksymalnego rozmiaru komponentu
- prosty system podpowiedzi (ToolTips)
- automatyczne przewijanie
- wspomaganie wielu wersji językowych

Swing - przykład

```
public class FirstGUI extends JFrame {
  JButton abort = new JButton("Abort");
  JButton retry = new JButton("Retry");
  JButton fail = new JButton("Fail");

  public FirstGUI() {
    super("Buttons");
    setSize(80, 140);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel pane = new JPanel();
    pane.add(abort);
    pane.add(retry);
    pane.add(fail);
    setContentPane(pane);
  }
  public static void main(String[] arguments) {
    FirstGUI rb = new FirstGUI();
    rb.setVisible(true);;
  }
}
```

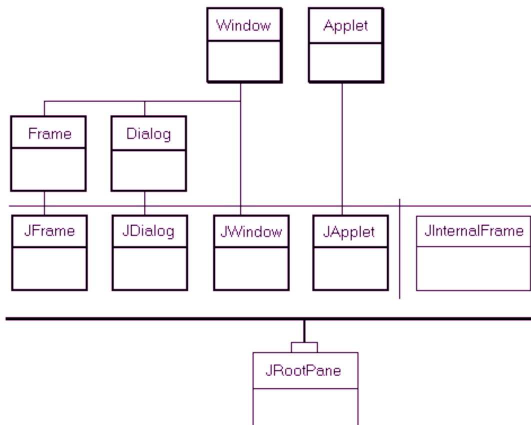
Kontener JFrame





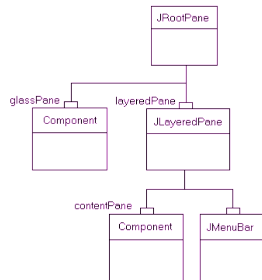
Przegląd klas Swing

JRootPane – nie wchodzi w skład hierarchii ale jest obiektem obsługującym zadania dla kontenerów (delegują zadania). Klasa która chce z niego korzystać musi implementować interfejs RootPaneContainer.



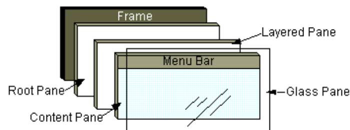
Przegląd klas Swing

- menuBar – jest opcjonalne
- glassPane jest umieszczone ponad wszystkim
- contentPane zawiera wszystkie komponenty danego kontenera
- dodawanie komponentów odbywa się przez metodę getContentPane().add(...)
- analogicznie ustawia się menedżer rozkładu
- metoda getRootPane() komponentu może służyć do uzyskania dostępu do obiektu JRootPane zawierającego komponent



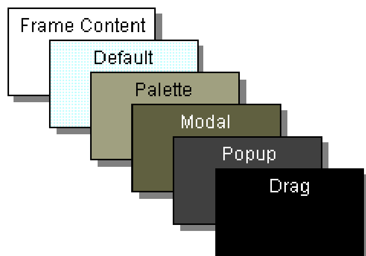
Przegląd klas Swing

- domyślny menedżer rozkładu dla JRootPane
  - umieszcza glassPane na całym widocznym obszarze (domyślnie glassPane jest przezroczysty)
  - layeredPane wypełnia cały obszar
  - menuBar jest umieszczany na górze obszaru zajmowanego przez layeredPane contentPane wypełnia cały pozostały obszar minus ramka i odstępy
- można zmienić menedżer rozkładu ale wtedy wszystkie okna trzeba pozycjonować ręcznie
- wystarczy zmieniać menedżer rozkładu dla contentPane przez getContentPane().setLayout(...);



klasa JLayeredPane – na poszczególnych warstwach dzieją się odmienne rzeczy

- FRAME_CONTENT_LAYER (-30000)
- DEFAULT_LAYER (0) zawiera większość zwykłych komponentów
- PALETTE_LAYER (100) przeznaczona dla pasków narzędzi, można je ustawiać ponad normalną zawartością okna
- MODAL_LAYER (200) dla modalnych dialogów, będą się pojawiały ponad paskami narzędzi
- POPUP_LAYER (300) wyświetla wyskakujące menu, podpowiedzi...
- DRAG_LAYER (400) zawiera przeciągany komponent. Po zakończeniu przeciągania komponent trafia do swojej normalnej warstwy



Java FX

JavaFX jest to framework (zestaw narzędzi) do budowy desktopowych jak i rozbudowanych aplikacji WEB (Rich Internet Applications - RIA). W zamyśle miał zastąpić Swing'a jako podstawowa biblioteka GUI dla Java SE ale w najbliższej przyszłości będą wspierane oba rozwiązania. Od Javy 8 jest rekomendowanym rozwiązaniem do tworzenia graficznego interfejsu użytkownika.

Pierwsze wydanie 2008. Nowe wersje wydawane są aktualnie dwa razy do roku (marzec, wrzesień).

Moduły JavaFX SDK

- `javafx.base` –podstawowe API dla JavaFX UI toolkit, (m.in. bindings, properties, collections, and events).
- `javafx.controls` – API do kontrolowania UI, wizualizacji danych (charts) czy skórek (skins).
- `javafx.graphics` – podstawowe API do tworzenia i obsługi elementów graficznych.
- `javafx.media` – API do obsługi odtwarzania mediów (m.in. MediaView czy MediaPlayer).
- `javafx.swing` – API do realizacji współdziałania pomiędzy JavaFX i Swing (SwingNode, JFXPanel).
- `javafx.web` – API do obsługi funkcjonalności internetowych w aplikacji (m.in., WebView czy WebHistory).

JavaFX vs Swing

	Java Swing	Java FX
Popularność	Ugruntowana pozycja	Systematyczny progres
Komponenty	więcej gotowych komponentów	mniej komponentów
User Interface	Standardowe elementy graficzne	Zaawansowane elementy UI (Rich GUI components)
Implementacja	Swing API	FXML,CSS i Scene builder
Rozwój	brak planów rozwoju	ciągły rozwój
MVC Support	niespójne wsparcie MVC	odseparowanie widoku od modelu i kontrolera
Multimedia	dodatkowe biblioteki (np. JMF)	pełne wsparcie audio, video, grafika 3D

JavaFX-podstawowe klasy

- `Application` – główna klasa po której dziedziczą wszystkie aplikacji JavaFX (można traktować jako odpowiednik JFrame dla Swing'a)
- `Stage` – klasa odzwierciedlająca najwyższy poziom kontenera w JavaFX. Podstawowy obiekt klasy Stage jest tworzony podczas uruchamiania aplikacji JavaFX i przekazywany do metody *`start()`*. Może odzwierciedlać okno jak i cały obszar wyświetlania (zależnie od urządzenia).
- `Scene` – Obiekt klasy Scene zawiera "scene graph" czyli kolekcje wszystkich elementów które tworzą interfejs użytkownika. Aplikacja może zawierać wiele "scen" ale tylko jedna może być przedstawiona z pomocą obiektu typu `Stage`.
- `Node` – klasa bazowa dla wszystkich elementów graficznych interfejsu użytkownika które są dodawane do sceny.

JavaFX – klasy narzędziowe

- `javafx.scene.paint.Color` – Klasa reprezentująca kolory wraz z kanałem alpha.
- `javafx.scene.Group` – Klasa grupująca obiekty typu Node.
- `javafx.scene.Camera` – Klasa bazowa dla obiektów pozwalających na renderowanie sceny. Jej podklasy to `ParallelCamera` i `PerspectiveCamera`.
- `javafx.scene.input.Event` – Klasa pozwalająca na wykrycie interakcji użytkownika z wybranym Node'em (przykładowe podklasy ActionEvent, DialogEvent, KeyEvent, MouseEvent, TouchEvent, GestureEvent)
- `javafx.event.EventHandler` –interfejs pozwalający na dodanie obsługi wybranego zdarzenia do danego Node'a.

JavaFX-podstawowe metody

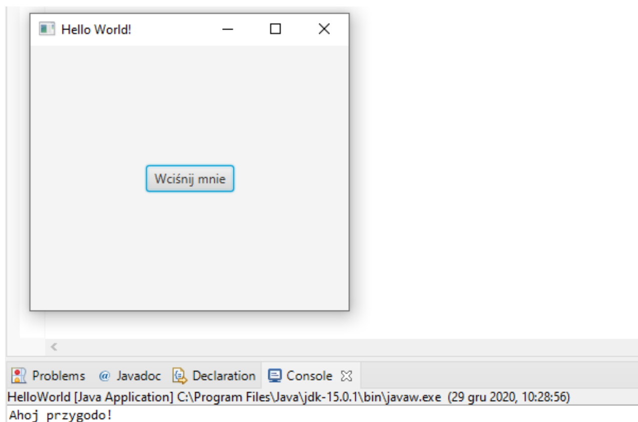
- *start()* – metoda z klasy Application która zostaje uruchomiona na początku działania aplikacji. W jej ciele zostaje zdefiniowany początkowy wygląd interfejsu użytkownika.
- *launch()* – metoda statyczna uruchamiająca aplikację w metodzie main.
- *show()* – metoda klasy Stage. Wyświetla daną scenę po jej przygotowaniu.
- *handle()* – metoda pochodząca z interfejsu `EventHandler`. Musi zostać zaimplementowana w celu dodania obsługi danego zdarzenia.
- *runLater()* – Metoda kolejkująca wątki modyfikujące wygląd UI w celu ich bezpiecznego uruchomienia dla JavaFX.
- *invokeLater()* – Metoda kolejkująca wątki modyfikujące wygląd UI w celu ich bezpiecznego uruchomienia dla Swing.

JavaFX-tworzenie projektu

W przypadku Eclipse IDE z zainstalowaną najnowszą wersją Javy (od wersji 11) aby uruchomić projekt JavaFX należy:

- Pobrać ze strony `https://gluonhq.com/products/javafx/` najnowsze JavaFX SDK
- Dodać pobrane biblioteki do Build Path w projekcie (najwygodniej stworzyć User Library).
- zmodyfikować plik `module-info.java` o linijkę:
  `opens nazwa_pakietu to javafx.graphics;`

JavaFX – pierwsza aplikacja



JavaFX – komponenty

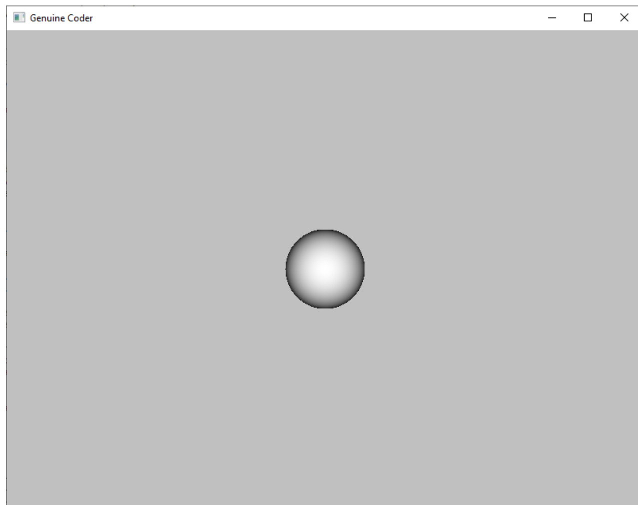
- Label
- Button
- RadioButton
- ToggleButton
- Checkbox
- ChoiceBox
- TextField
- PasswordField
- ScrollBar
- ScrollPane
- ListView
- TableView
- TreeView
- TreeTableView
- ComboBox
- Separator
- Slider
- ProgressBar
- ProgressIndicator
- Hyperlink
- Tooltip
- HTMLEditor
- TitledPane
- Accordion
- Menu
- ColorPicker
- DatePicker
- PaginationControl
- FileChooser

https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm

JavaFX – Layouty

- `BorderPane` – udostępnia 5 stref na komponenty: top, bottom, left, right, center.
- `HBox` – ułożenie komponentów w jednym wierszu,
- `VBox` – ułożenie komponentów w jednej kolumnie,
- `StackPane` – ułożenie komponentów jeden na drugim,
- `GridPane` – ułożenie komponentów w formie siatki o określonej liczbie wierszy i kolumn,
- `FlowPane` – ułożenie elementów w kolumnach i następnie w kolejnych wierszach,
- `TilePane` – ułożenie komponentów w formie siatki komórek o tej samej wielkości, z wykorzystaniem `prefColumns` i `prefRows` można wymusić żądaną liczbę wierszy lub kolumn,
- `AnchorPane` – układ pozwala "przypiąć" dany element w żądanej pozycji (relatywnie do top, bottom, left, right), przy zmianie rozmiaru okna elementy dopasowują pozycję do tych zmian.

JavaFX – 3D



https://www.genuinecoder.com/javafx-3d-tutorial-introduction/

FXML

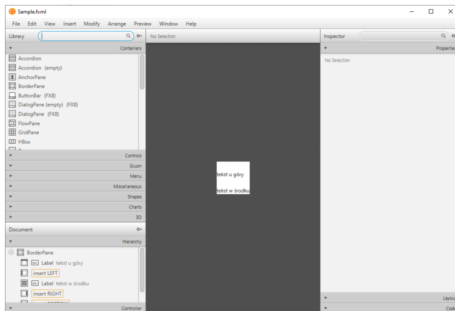
JavaFX pozwala na wykorzystanie języka skryptowego FXML do tworzenia aplikacji (FX XML-eXtra Markup Language).
FXML pozwala na:

- dużo łatwiejszą edycję UI w porównaniu do kodu Java,
- dużą szybkość tworzenia kodu,
- możliwość wykorzystania CSS do ujednolicenia stylu kontrolek,
- łatwiejsze rozdzielenie roli w zespole,
- możliwość wykorzystania Scene Buildera

Eclipse aby wykorzystać FXML musi mieć doinstalowany plugin E(FX)clipse. IntelliJ jest gotowe do pracy od razu po zainstalowaniu.

Scene Builder

Dzięki wykorzystaniu języka FXML, możliwe jest wykorzystanie graficznego narzędzia do projektowania UI. Scene Builder musi być doinstalowany ze strony `gluonhq.com/open-source/scene-builder/`.
W Eclipse należy dodać ścieżkę do Scene Buildera w `Window>Preferences>JavaFX`.



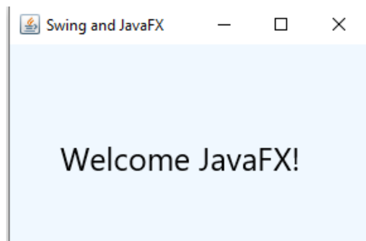
Współdziałanie JavaFX i Swing

Ze względu na ugruntowaną pozycję Swing'a i jego rozbudowane możliwości, w ramach JavaFX zdecydowano się na realizację klas pozwalających na współpracę tych dwóch środowisk.
Istnieją dwie możliwości realizacji takiego współdziałania:

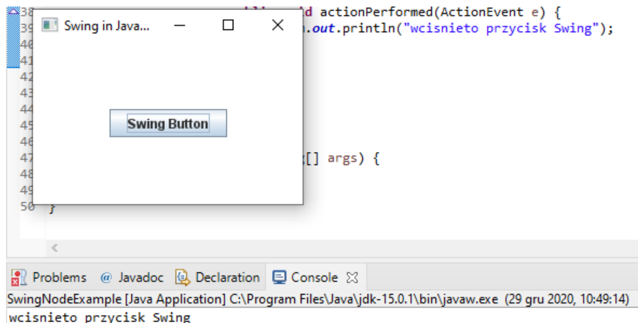
- Integracja komponentów JavaFX w aplikacjach Swing. W tym celu powstała klasa JFXPanel, która dziedziczy po klasie `javax.swing.JComponent` dzięki czemu możliwe jest jej dodanie do typowych kontenerów Swing takich jak JFrame czy JPanel.
- Integracja komponentów z biblioteki Swing wewnątrz aplikacji JavaFX. W tym celu powstała klasa SwingNode dziedzicząca po klasie JavaFX `Node`, która udsotępnia metodę *`setContent()`*, która jako argument przyjmuję obiekty pochodzące od `JComponent` z biblioteki Swing.

https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/fx_swing.htm

JFXPanel



SwingNode



Wyrażenia Lambda – Programowanie Funkcyjne

W programowaniu funkcyjnym, w odróżnieniu od programowania obiektowego, najważniejszym i zarazem jedynym narzędziem są funkcje. Funkcje mogą przyjmować na wejściu także funkcje oraz zwracać funkcje jako wynik.

Lambda Expressions - wprowadzenie

Rachunek lambda – system formalny używany do badania zagadnień związanych z podstawami matematyki jak rekurencja, definiowalność funkcji, obliczalność, podstawy matematyki np. definicja liczb naturalnych, wartości logicznych itd. Rachunek lambda został wprowadzony przez Alonzo Churcha i Stephena Cole'a Kleene'ego w 1930 roku. (prog. obiektowe w latach 60)

W rachunku lambda każde wyrażenie określa funkcję jednoargumentową. Z kolei argumentem tej funkcji jest również funkcja jednoargumentowa, wartością funkcji jest znów funkcja jednoargumentowa. Funkcja jest definiowana anonimowo przez wyrażenie lambda, które opisuje, co funkcja robi ze swoim argumentem.

Pomijając matematyczne wywody, lambda to tak naprawdę kawałek kodu, który można przekazać do innego kawałka programu w celu wykonania. Czyli parametrem nie jest obiekt czy wartość, tylko sam kod.

Czym jest lambda w Javie? - definicja

Lambda jest instancją interfejsu funkcyjnego. Interfejs funkcyjny to interfejs, który definiuje tylko jedną abstrakcyjną metodę. Metoda abstrakcyjna to taka metoda, która nie posiada implementacji.

Lambdy można również rozumieć jako skróconą formę dla anonimowej klasy implementującej z interfejsu pojedynczą metodę abstrakcyjną. Dlatego też nie są one sposobem na wprowadzenie paradygmatu programowania funkcyjnego do Javy (i odstąpienie od pełnej obiektowości Javy) a jedynie wykorzystują ten koncept w sposób obiektowy.

Boilerplate code

“Boilerplate code” to kod, który nie wnosi do programu niczego użytecznego, a którego obecność jest wymagana do jego działania. Przez niego kod aplikacji puchnie (wielokrotnie powtarzane są sekcje identycznego lub prawie identycznego kodu), co potrafi utrudnić lub spowolnić pracę nad nim. W językach programowania takich jak Java jest to niezwykle często występujące zjawisko gdy wymagana jest stosunkowo duża ilość takiego kodu do zapewnienia niewspółmiernie mniejszej funkcjonalności.

Lambda - jak to działa?

W przypadku wyrażeń lambda to kompilator określą jednoznacznie typ dla zmiennych w wyrażeniu lambda z kontekstu wywołanej metody. Działa to wyłącznie przy braku jakiejkolwiek niejednoznaczności czyli w sytuacjach, które są oczywiste. np.:

```
btn.addActionListener( new ActionListener(){
        public void actionPerformed(ActionEvent e){
                System.out.println("btn␣pressed");
        }
});
```

W powyższym przykładzie klasa implementująca intefejs ActionListener i zmienna typu ActionEvent moga bez problemu zostać jednoznacznie wyznaczone z metody *addActionListener* dla obiektu przycisku.

Moja pierwsza Lambda :)

Lambdy przejawiają się niezwykle skróconą formą. Dla kodu omówionego na poprzednim slajdzie:

```
btn.addActionListener( new ActionListener(){
public void actionPerformed(ActionEvent e){
System.out.println("btn␣pressed");
}
});
```

Lambda przyjmie następującą formę:

```
btn.addActionListener(e -> System.out.println("btn␣pressed"));
```

Działanie w obu przypadkach jest dokładnie takie same.

Przykłady Lambd

- Lambda bez argumentów:

```
Runnable noArguments = () -> System.out.println("Hello");
```

- Lambda z blokiem kodu:

```
Runnable multiStatement = () -> {
  System.out.print("Hello");
  System.out.println(" World");
};
```

- Lambda z wieloma argumentami:

```
BinaryOperator<Long> add = (x, y) -> x + y;
```

- Lambda z określonymi typami dla argumentów:

```
BinaryOperator<Long> add = (long x, long y) -> x + y;
```

Przykłady Lambd

- W wyrażeniach Lambda możliwe jest wykorzystanie zmiennych lokalnych (muszą być finalne lub efektywnie finalne):

```
final string id="Andrzej";
btn.addActionListener(e -> System.out.println("Hi␣"+ id));
```

Interfejsy funkcyjne

Interfejs funkcyjny to interfejs, który definiuje tylko jedną abstrakcyjną metodę. np.:

```
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent event);
}
```

Wbudowane interfejsy funkcyjne w Javie (najczęściej używane):

Interface name	Arguments	Returns	Example
Predicate<T>	T	boolean	Has this album been released yet?
Consumer<T>	T	void	Printing out a value
Function<T,R>	T	R	Get the name from an Artist object
Supplier<T>	None	T	A factory method
UnaryOperator<T>	T	T	Logical not (!)
BinaryOperator<T>	(T, T)	T	Multiplying two numbers (*)

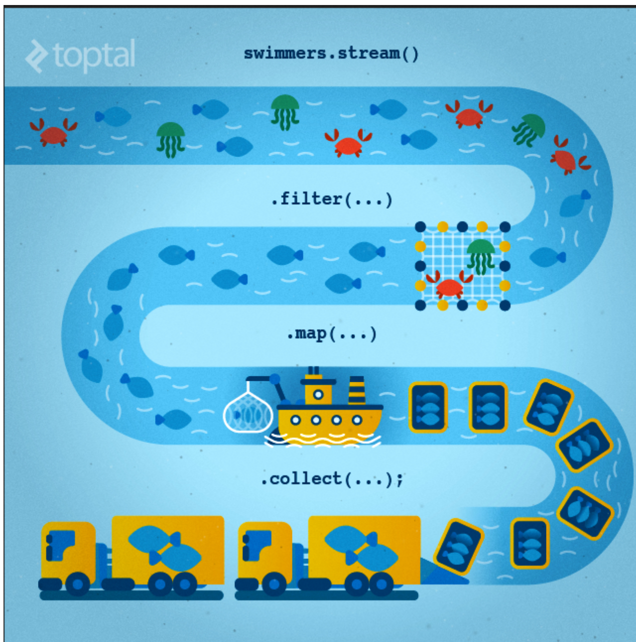
Przykłady użycia interfejsów funkcyjnych

```
Predicate<Integer> lesserthan = i -> (i < 18);
System.out.println(lesserthan.test(10));

Consumer<Integer> display = a -> System.out.println(a);
display.accept(10);

Function<Integer, Double> half = a -> a / 2.0;
System.out.println(half.apply(10));

Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());

UnaryOperator<Integer> func2 = x -> x * 2;
Integer result2 = func2.apply(2);

BinaryOperator<Integer> op = BinaryOperator.maxBy((a, b) ->
    (a > b) ? 1 : ((a == b) ? 0 : -1));
System.out.println(op.apply(98, 11));
```

Stream API

Java Stream API jest jedną z głównych funkcjonalności dodanych w Java 8. Stream API nie należy mylić ze strumieniami z pakietów Java IO. Stream API służy do "opakowywania" źródła danych oraz wykonywanie na nim żądanych operacji. Stream API dostarcza metody do przetwarzania poszczególnych elementów które wchodzą w skład kolekcji czy tablicy.

W strumień można opakować praktycznie dowolny zestaw danych. Strumienie pozwalają również w łatwy sposób zrównoleglić pracę na danych. Dzięki temu przetwarzanie dużych zbiorów danych może być dużo szybsze. Strumienie kładą nacisk na operacje jakie należy przeprowadzić na danych.


toptal
swimmers.stream()
.filter(...)
.map(...)
.collect(...);

Feature	Stream	Collection
Version	Stream API was introduced in version 8.	Collection API was introduced in Java 1.2.
Usage	Stream API is used for computation of data on a large set of Objects.	Collection API is used for storing data in different kinds of data structures.
Finite	With Stream API, we can handle streams of data that can contain infinite number of elements.	With Collection API we can store a finite number of elements in a data structure.
Eager vs. Lazy	Stream API creates objects in a lazy manner.	Collection API constructs objects in an eager manner.
Multiple consumption	With Stream API we can consume or iterate elements only once.	Most of the Collection APIs support iteration and consumption of elements multiple times.

Metody w Stream API

- `filter` – zwraca strumień obiektów, które spełniają żądane warunki

```
myCarsCollection.stream()
.filter(car -> car.colorCompare(Color.blue));
```

- `map` – zwraca strumień obiektów zwróconych przez zadaną funkcję

```
myCarsCollection.stream().map(car -> {
return new Car(Color.blue, car.getName());
```

- `sorted` – Zwraca posortowany strumień (wymaga implementacji Comparable)

```
myCarsCollection.stream().sorted()
.forEach(System.out::println);
```

- `limit` i `skip` – zwraca strumień `n` pierwszych/ominiętych elementów
- `forEach` – wykonuje daną funkcję dla wszystkich elementów w strumieniu

```
myCarsCollection.stream()
.filter(car -> car.colorComapre(Color.blue));
```

Metody w Stream API

- toArray – Zamienia strumień na tablicę obiektów

```
Object[] carArray=myCarsCollection
.stream().toArray();
```

- reduce – pozwala zredukować strumień do pojedynczej wartości

```
String allNames=myCarsCollection.stream()
.reduce("",(partialString, car) ->
partialString+" "+car.getName(), String::concat);
```

- collect – tworzy kolekcję na podstawie strumienia:

```
myCarsCollection.stream()
.collect(Collectors.toList());
```

- min max– redukcja do najmniejszego/najwiekszego elementu

```
myCarsCollection.stream().min(Comparator
.comparing(car -> car.getPower())).get();
```

- count – Zlicza elementy w strumieniu

```
myCarsCollection.stream().count();
```

Java 8 - new features

Kluczowe zmiany/aktualizacje:

- Metody domyślne w interfejsach
- Wyrażenia Lambda
- Stream API java.util.stream

https://www.oracle.com/java/technologies/javase/8-whats-new.html

Java 9 - new features

Kluczowe zmiany/aktualizacje:

- Java Platform Module System (JPMS) – Project `Jigsaw` – to największa zmiana w organizacji kodu źródłowego jaka do tej pory miała miejsce. Rolę modułu można określić nieformalnie jako „paczka pakietów Java” pozwalający lepiej organizować źródła kodu.
- JShell – narzędzie pozwalające na uruchamianie kodu javy poprzez linię komend. Lista dostępnych poleceń /help np /vars czy /types.
- prywatne metody w interfejsach

`https://docs.oracle.com/javase/9/whatsnew/`

Java 10 - new features

Kluczowe zmiany/aktualizacje:

- local variable type inference – słowo kluczowe `var` – kompilator jest w stanie wywnioskować typ na podstawie prawej strony przypisania. Słowo `var` może być używane tylko dla zmiennych lokalnych.

https://www.oracle.com/java/technologies/javase/10-relnote-issues.html

Java 11 - new features

Kluczowe zmiany/aktualizacje:

- Nowe metody dla klas String i Files:

```
"Marco".isBlank();
"Mar\nco".lines();
"Marco␣␣".strip();
Path path = Files.writeString(
  Files.createTempFile("helloworld", ".txt"),
  "Hi,␣my␣name␣is!");
String s = Files.readString(path);
```

- Możliwość wykorzystania var w lambdach:

```
(var firstName, var lastName) -> firstName + lastName
```

- Run Source Files (bez kompilacji):
  `c:\java MyScript.java`

https://www.oracle.com/java/technologies/javase/jdk-11-relnote.html#NewFeature

Java 12 - new features

Kluczowe zmiany/aktualizacje:

- Pełne wsparcie dla Unicode 11 (Unicode 11.0 adds 684 characters, for a total of 137,374 characters. These additions include 7 new scripts, for a total of 146 scripts, as well as 66 new emoji characters.)

https://www.oracle.com/java/technologies/javase/12-relnote-issues.html#NewFeature

Java 13 - new features

Kluczowe zmiany/aktualizacje:

- Text Blocks – Preview Feature – w bloku tekstu można dowolnie wykorzystywać łamanie linii, cudzysłowia jak i fragmentów kodu w postaci HTML, JSON czy SQL.

```
String block="""
some␣text""";
```

- Wsparcie dla Unicode 12.1

https://www.oracle.com/java/technologies/javase/13-relnote-issues.html

Java 14 - new features

Kluczowe zmiany/aktualizacje:

- Nowy format instrukcji switch (brak `brake`):

```
int x=switch (day) {
case MONDAY, FRIDAY, SUNDAY -> 6;
case TUESDAY                -> 7;
case THURSDAY, SATURDAY     -> 8;
case WEDNESDAY              -> 9;
default -> 0;
};
```

"Feature" wprowadzony jako preview w wersji 12 i uzupełniony w 13.

Java 15 - new features

Kluczowe zmiany/aktualizacje:

- Text Blocks wersja release.
- Sealed classes - preview

```
public abstract sealed class Shape
permits Circle, Rectangle {...}
```

- Records - preview

```
record Point(int x, int y) { }
```

https://www.oracle.com/java/technologies/javase/15-relnote-issues.html#NewFeature

Java 16 - new features

Kluczowe zmiany/aktualizacje:

- Records - finalize
- Pattern Matching for instanceof

```
if (obj instanceof String s) {
    // Let pattern matching do the work!
   ...
}
```

https://www.oracle.com/java/technologies/javase/16-relnote-issues.html#NewFeature

Java 17 (LTS) - new features

Java 17 jest dostępna na nowej licencji Oracle No-Fee Terms and Conditions License (w skrócie NFTC). Kolejne wersje Javy również będą wydawane na tej licencji. Licencja NFTC pozwala na użycie komercyjne i produkcyjne dystrybucji Oracle JDK za darmo.
Kluczowe zmiany/aktualizacje:

- Sealed classes - finalize
- Pattern matching for switch - preview

```
String formatted = switch (o) {
  case Integer i -> String.format("int %d", i);
  case Long l    -> String.format("long %d", l);
  case Double d  -> String.format("double %f", d);
  case String s  -> String.format("String %s", s);
  default        -> o.toString();
};
```

https://www.oracle.com/java/technologies/javase/17-relnote-issues.html#NewFeature

Java 18 - new features

Kluczowe zmiany/aktualizacje:

- UTF-8 by Default – Starting with JDK 18, UTF-8 is the default charset for the Java SE APIs
- Simple Web Server – jwebserver, a command-line tool to start a minimal static web server
- Internet-Address Resolution SPI – InetAddress may use resolvers other than the built-in resolver of the platform.
  https://www.oracle.com/java/technologies/javase/18-relnote-issues.html#NewFeature

Java 19 - new features

Kluczowe zmiany/aktualizacje:

- Support Unicode 14.0
- Additional Date-Time Formats np. `DateTimeFormatter.ofLocalizedPattern("yMMM")`
- Nowe funkcjonalności w wersji Preview/Incubator – Virtual Threads, Structured Concurrency, Record Patterns, Pattern Matching for switch, Vector API
  https://www.oracle.com/java/technologies/javase/19-relnote-issues.html#NewFeature

LEARN JAVA, THEY SAID
IT'LL BE FUN, THEY SAID
quickmeme.com