

Struktura programu:

```
package pl.mojastrona.mojpakiet;

import javax.swing.*;
import java.awt.Container;

class MojaKlasa extends JFrame
{
    public MojaKlasa()
    {
        ...
    }
}

class KlasaStartowa
{
    MojaKlasa ob1;
    MojaKlasa ob2 = new MojaKlasa();

    public static void main(String[] args)
    {
        ...
    }
}
```

Określenie pakietu, do którego należą klasy zdefiniowane w tym pliku (*opcjonalne...*).

Zewnętrzne pakiety (lub pojedyncze klasy, nterfejsy), z których korzystamy w naszym programie.
„odpowiednik” dyrektywy `#include` w C/C++.

Deklaracja klasy rozszerzającej inną klasę (dziedziczenie)

Konstruktor – taka sama nazwa jak klasa, może być kilka definicji konstruktorów dla jednej klasy, np.
`public MojaKlasa(int parametrPoczątkowy)`
{
}

Druga klasa, w której deklarowane są referencje do obiektów innej klasy, oraz tworzony jest nowy obiekt operator „new” + wywołanie konstruktora

Metoda *main* klasy startowej – od niej rozpoczyna się uruchamianie programu

Struktura klasy:

```
class NazwaKlasy {  
    //deklaracje pól  
    Typ pole1;  
    ...  
    Typ poleN;  
  
    //deklaracje metod i konstruktora/ów  
    Typ1 metoda1(lista-parametrów) {  
        //treść/zawartość metody1  
        return obiektTyp1;  
    }  
    ...  
    void metodaM(lista-parametrów) {  
        //treść/zawartość metodyM  
    }  
  
    NazwaKlasy(lista parametrów) {  
        //treść/zawartość konstruktora  
    }  
}
```

Tworzenie klasy

```
[modyfikator] class NazwaKlasy [extends  
    NazwaKlasyBazowej] [implements NazwaInterfejsu] {  
  
    // deklaracje pól  
  
    // deklaracje metod i konstruktorów  
  
}
```

Przykład:

```
public class Punkt {  
    //pola  
    double x = 0 , y = 0;  
  
    //konstruktor dwuargumentowy  
    Punkt (double parametr1, double parametr2){  
        x = parametr1;  
        y = parametr2;  
    }  
}
```

Nawet jeśli nie ma jawnego dziedziczenia, powstała klasa jest rozszerzeniem klasy Object - <http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html> .
Powyższa deklaracji klasy jest równoważna temu:

```
public class Punkt extends Object{  
    double x = 0 , y = 0;  
  
    Punkt (double parametr1, double parametr2){  
        x = parametr1;  
        y = parametr2;  
    }  
}
```

Klasa Object

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notifv() method or the notifvAll

metoda equals()

Służy ona do porównywania, czy dwa obiekty są sobie równe. Implementując metodę equals(), należy pamiętać o kilku zasadach

- metoda equals musi być zwrotna, czyli `a.equals(a)==true`.
- metoda equals musi być symetryczna, czyli `a.equals(b)==b.equals(a)`.
- metoda equals musi być przechodnia, czyli jeżeli `a.equals(b)==true` i `b.equals(c)==true` to `a.equals(c)==true`. (Jednak gdy: `a.equals(b)==false` i `b.equals(c)==false` to `a.equals(c)` może być `true`)
- metoda equals musi być konsekwenta, czyli gdy dwa razy (w różnych chwilach czasu) porównujemy te same obiekty, to wynik tego porównania powinien być taki sam.
- obiekt jest nie równy null, czyli `a.equals(null)==false` dla każdego `a` nie będącego null

`int hashCode() -> HashMap (...)`

Przykład przedefiniowania metody toString() odziedziczonej z klasy bazowej Object

```
public class Punkt extends Object{
    double x = 0 , y = 0;

    Punkt (double parametr1, double parametr2){
        x = parametr1;
        y = parametr2;
    }

    public String toString(){
        String text;
        text = "Wspolrzedne (" + x + ", " + y + ")";
        return text;
    }
}
```

Korzystanie z utworzonej klasy

```
public class PunktTest {  
  
    public static void main(String[] args) {  
  
        int rozmiarTablicy = 10;  
        Punkt[] tablicaPunktow = new Punkt[rozmiarTablicy];  
        //Tworzenie tablicy obiektów klasy Punkt  
  
        for (int i=0; i<tablicaPunktow.length; i++)  
            tablicaPunktow[i] = new Punkt (Math.random()*100,  
Math.random()*100);  
        //dodawanie do tablicy kolejno tworzonych obiektów  
        for (int j=0; j<tablicaPunktow.length; j++)  
            System.out.println(tablicaPunktow[j]);  
        //wypisywanie elementów - wykorzystanie metody Punkt.toString()  
  
    }  
}
```



```
public String toString(){
    String text;
    text = "Wspolrzedne (" + x + ", " + y + ")";

    //Tworzenie ciągu znakowego z formatowaniem liczb:
    text = "Wspolrzedne (" + String.format("%.2f",x) +
", " + String.format("%.2f",y) + ")";
    return text;
}
```

Formatowanie analogiczne do printf() znanego z C/C++:

```
System.out.printf("%d+%d=%d\n", 2, 2, 2 + 2);
String s = String.format("%d+%d=%d\n", 2, 2, 2 + 2);
System.out.print(s);
```

Więcej o formatowaniu:

<http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html>

Opis	Literał
New line (znak nowej linii)	\n
Horizontal tab (tabulacja pionowa)	\t
Backspace	\b
Carriage return (powrót karetki)	\r
From feed (znak nowej strony)	\f
Single quote (apostrof)	\'
Double quote (cudzysłów)	\"
Backslash (lewy ukośnik)	\\

Znak	Kod Unicode	Litera	Kod Unicode
Ą	0104	Ó	00D3
ą	0105	ó	00F3
Ć	0106	Ś	015A
ć	0107	ś	015B
Ę	0118	Ż	0179
ę	0119	ż	017A
Ł	0141	Ž	017B
ł	0142	ž	017C

```
System.out.println("Dzi\u0119kuj\u0119") ;
```

Definiowanie metody

```
[modyfikator] TypZwracanejWartości  
nazwaMetody (TypArgumentu1 zmienna1,  
..., TypArgumentuN zmiennaN) {  
  
    TypZwracanejWartości zwracanaWartość;  
    // kod metody, w którym można korzystać  
    ze zmiennych1..N  
  
    return zwracanaWartość  
    // jeśli metoda „void” – nic nie zwraca  
}
```

```
public class Punkt extends Object{
    double x = 0 , y = 0;

    Punkt (double parametr1, double parametr2){
        x = parametr1;
        y = parametr2;
    }

    void przesun(double parametr1, double parametr2){
        x += parametr1;
        y += parametr2;
    }

    double odlegloscOdPoczatkuUkladu(){
        return Math.sqrt(x*x+y*y);
    }

    public String toString(){
        String text;
        text = "Wspolrzedne (" + x + ", " + y + ")";
        return text;
    }
}
```

Korzystanie z utworzonych metod

```
public class PunktTest2 {  
    public static void main(String[] args) {  
  
        Punkt p = new Punkt(0,0);  
        int i=0;  
        while (i<10){  
            p.przesun(5, 10);  
            double d = p.odlegloscOdPoczatkuUkladu();  
            System.out.println(p + "    Odległość od (0, 0): " + d);  
            i++;  
        }  
    }  
}
```

Dziedziczenie/rozszerzanie

```
[modyfikator] class KlasaPochodna  
    extends KlasaBazowa {  
  
    ...  
  
    // pola i metody klasy pochodnej  
}
```

Każda KlasaPochodna (podklasa) posiada tylko jedną KlasęBazową (nadklasę).

```
public class Kolo extends Punkt {  
    double promien = 0;
```

`super(p1, p2)` jest równoważne
Z: `new Punkt(p1, p2)`

```
    public Kolo(double parametr1, double parametr2, double parametr3) {  
        super(parametr1, parametr2);  
        promien = parametr3;  
    }
```

```
    public Kolo(){  
        super(0,0);  
        promien = 10;  
    }
```

Konstruktor bezargumentowy
– „domyślne” parametry klasy

```
    public Kolo (Punkt p, double parametr ){  
        super (p.x, p.y);  
        promien = parametr;  
    }
```

```
    public Kolo(Kolo k){  
        super (k.x,k.y);  
        promien = k.promien;  
    }
```

Konstruktor kopiujący

```
}
```

Użycie „super” jako Konstruktora

- Wywołanie konstruktora nad-klasy:
`super(lista-parametrow)`
- Musi być pierwszą instrukcją konstruktora podklasy:

```
class NadKlasa { ... }  
class PodKlasa extends NadKlasa {  
    PodKlasa(...) {  
        super(...); ...  
    } ...  
}
```


Odwołanie do Nad-Klasy przez „super”

- Odwołanie do elementu nad-klasy:

```
super.pole
```

```
super.metoda()
```

- Stosowane szczególnie gdy składowe pod-klasy przesłaniają składowe nad-klasy o tych samych nazwach.

```
public class KoloTest {
    public static void main(String[] args) {
        Kolo a,b,c,d,e;
        // Wykorzystanie różnych konstruktorów:
        a = new Kolo (10,10,5);
        b = new Kolo();
        Punkt p1 = new Punkt (30,30);
        c = new Kolo (p1, 30);
        d = new Kolo (new Punkt(40,40), 40 );
        e = new Kolo(c);

        System.out.println("" + a + "\n" + b + "\n"+ c + "\n" + d + "\n"+ e);
        // Jeśli klasa pochodna Kolo nie ma zdefiniowanej własnej metody
        // toString() wykorzystywana jest metoda toString klasy bazowej Punkt

        System.out.println(new Kolo(60,60,60)); // tworzenie "chwilowego"
        // obiektu bez referencji, na potrzeby wywołania metody


        e.przesun(-100, -100);
        // korzystanie z metody zdefiniowanej w klasie bazowej
        System.out.println(e);

    }
}
```

Szybkie tworzenie klasy pochodnej w Eclipse

New Java Class

Java Class

 The use of the default package is discouraged.

Source folder: Wyklad/src Browse...

Package: (default) Browse...

☐ Enclosing type: Browse...

Name: KoloroweKolo

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: pl.edu.pw.fizyka.pojava.wyklad.Kolo Browse...

Interfaces: Add...
Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☒ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

? Finish Cancel

[modyfikatory]

Modyfikatory widoczności ogólnie :

- **public** – widoczne/postępne dla wszystkich
- **private** – widoczne tylko wewnątrz danej klasy (niedziedziczone)
- **protected** – widoczne w obrębie pakietu oraz w klasach potomnych z innych pakietów
- **nienazwana** (friendly) – jak public w obrębie pakietu

Modyfikatory „dostępu”:

- **static** - dostęp bez uprzedniego utworzenia
- **final** - brak możliwości zmiany (pola mogą być inicjalizowane w konstruktorze)

Modyfikatory klas

- publiczne (`public`) są dostępne poza pakietem, w którym są zdefiniowane. Jeśli klasa `NazwaKlasy` zdefiniowana w pakiecie o pełnej nazwie `jakis.pakiet` - pełna nazwa klasy to `jakis.pakiet.Klasa`
- nienazwane (`friendly`) – widoczne całym pakiecie, w którym zostały zdefiniowane
- końcowe (`final`) - nie może posiadać podklas klasa nie może być jednocześnie abstrakcyjna i końcowa
- abstrakcyjne (`abstract`) - może zawierać metody abstrakcyjne (niezaimplementowane) przez dziedziczenie metod abstrakcyjnych od nadklas, interfejsów lub przez jawną deklarację metody abstrakcyjnej. Klasa nieabstrakcyjna nie może zawierać metod abstrakcyjnych. Nie może zostać stworzona instancja klasy abstrakcyjnej, powstaje wyjątek (`InstantiationException`), może być nadklasą dla innych klas (które już nie muszą być abstrakcyjne)

Składowe klas (pola, metody, klasy wewnętrzne...)

- mogą być zadeklarowane bezpośrednio w klasie lub odziedziczone po nadklasach lub interfejsach
- prywatne (`private`) nie są dziedziczone
- publiczne (`public`) i chronione (`protected`) są dostępne w podklasach klasy, w której zostały zadeklarowane
- konstruktory i statyczne inicjalizatory nie są składowymi i przez to nie są dziedziczone

Dodatkowe modyfikatory deklaracji pól

- pole statyczne (`static`) jest wspólne dla wszystkich instancji danej klasy, bez względu ich liczbę (również gdy nie ma żadnej)

Color `kolor` = Color.`WHITE`;

- pole niestatyczne jest tworzone dla każdej instancji klasy niezależnie
- końcowe (`final`) pola muszą zawierać inicjalizację w miejscu zadeklarowania i nie mogą być modyfikowane w czasie działania programu. Jeśli odnosi się do obiektu to jego stan może się zmieniać, ale zmienna będzie pokazywała zawsze na ten sam obiekt.
- pola zmienne (`volatile`) informują o możliwości asynchronicznej zmiany wartości przez inne wątki

Dodatkowe modyfikatory deklaracji metod

- metody abstrakcyjne (`abstract`) są identyfikowane jako składowe klas, ale nie zawierają implementacji. Mogą występować wyłącznie w klasach abstrakcyjnych.
- metody statyczne (`static`) istnieją niezależnie od instancji klasy i wywoływane są od niej niezależnie. Odwołanie do obiektu przez `this` powoduje błąd kompilacji.

```
double x = Math.sqrt(y);
```
- końcowe metody (`final`) nie mogą być przesłaniane w klasach pochodnych
- metody źródłowe (`native`) są implementowane w kodzie zależnym od platformy
- metoda synchronizowana (`synchronized`) może być wywołana tylko w przypadku, gdy nie jest jednocześnie wywoływana żadna inna metoda synchronizowana danej klasy

Elementy static

Zwykle aby móc używać elementów (pól, metod) zdefiniowanych w klasie, należy najpierw utworzyć obiekt będący instancją danej klasy. Istnieje jednak możliwość zdefiniowania elementów, do których nie musimy się odwoływać za pośrednictwem obiektów.

Do wyróżnienia takich elementów służy słowo kluczowe **static**.

- **Zmienne instancji:** są one wówczas *zmiennymi globalnymi* wszystkich obiektów danej klasy. Utworzenie obiektu danej klasy *nie powoduje powstania statycznej zmiennej instancji*.
- **Metody:** podlegają one następującym restrykcjom:
 - ✓ Mogą wywoływać tylko inne metody statyczne
 - ✓ Mogą korzystać wyłącznie ze zmiennych statycznych
 - ✓ Nie mogą odwoływać się do zmiennych **this** i **super**

Elementy final

Słowo kluczowe **final** służy do oznaczenia zmiennych, których wartość ma być *niezmienna we wszystkich instancjach*. Pełnią one rolę stałych i muszą być inicjalizowane przy deklaracji. Przykłady:

```
final int FILE_NEW = 1;
```

```
final int FILE_OPEN = 2;
```

```
final int FILE_SAVE = 3;
```

```
final int FILE_QUIT = 4;
```

Dalsza część programu może używać powyższych zmiennych jak stałych. Dodatkowo nie zajmują one miejsca w instancjach klasy.

Przyjętą konwencją jest pisanie zmiennych **final dużymi literami**.

Klasy finalne – nie mogą być rozszerzane

Metody finalne - nie mogą zostać zaimplementowana w klasie pochodnej a ich implementacja musi zostać podana w klasie w której zostały zadeklarowane

Stosowanie widoczności:

- klasa – public / friendly
- klasa wewnętrzna – public / private / protected / friendly
- konstruktor – public / protected / friendly
- metoda – public / private / protected / friendly
- pole – public / private / protected / friendly
- stała – public / private

Słowo kluczowe „this”

Czasem metoda obiektu potrzebuje odwołać się do obiektu wywołującego tę metodę. Umożliwia to słowo kluczowe **this**. Wewnątrz metody **this** oznacza obiekt, na rzecz którego wywołano metodę.

W Java nie można deklarować dwóch takich samych zmiennych lokalnych w obrębie jednego zakresu, jednak istnieje możliwość przesłonięcia zmiennych instancyjnych klasy przez parametry metody. Wewnątrz metody zmienne instancyjne byłyby niedostępne, gdyby nie możliwość użycia zmiennej **this**:

```
public class Punkt {  
    double x = 0 , y = 0;  
    Punkt (double x, double y){  
        this.x = x;  this.y = y;  
    }  
}
```

Hermetyzacja („enkapsulacja”) danych

```
public class PunktPriv {  
    private double x = 0 , y = 0;  
  
    PunktPriv (double x, double y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

✗ Remove 'x', keep assignments with side effects

➤ Create getter and setter for 'x'...

📄 Rename in file (Ctrl+2, R)

📄 Rename in workspace (Alt+Shift+R)

@ Add @SuppressWarnings 'unused' to 'x'

Press 'Ctrl+Enter' to fix 2 problems of same category in file

Encapsulate Field

Getter name: (new getter created)

Setter name: (new setter created)

[Configure naming conventions...](#)

Field access in declaring type: ☒ use setter and getter ☐ keep field reference

Insert new methods after:

☐ Generate method comments

Preview > OK Cancel

Hermetyzacja („enkapsulacja”) danych

```
public class PunktPriv {  
    private double x = 0 , y = 0;  
  
    PunktPriv (double x, double y){  
        this.setX(x);  
        this.setY(y);  
    }  
    public double getX() {  
        return x;  
    }  
    public void setX(double x) {  
        this.x = x;  
    }  
    public double getY() {  
        return y;  
    }  
    public void setY(double y) {  
        this.y = y;  
    }  
}
```

Zalety hermetyzacji

- uodparnia tworzony model na błędy,
- lepiej oddaje rzeczywistość,
- umożliwia rozbić model na mniejsze elementy

Np. metody `setNazwaPola(...)` można wykorzystać do kontroli przekazywanych argumentów

```
public class PunktPriv {  
    private double x = 0 , y = 0;  
    private final int XMIN = -100, XMAX = 100;  
    private final int YMIN = -100, YMAX = 100;
```

```
    PunktPriv (double x, double y){  
        this.setX(x);  
        this.setY(y);  
    }
```

```
    public double getX() {  
        return x;  
    }
```

```
    public void setX(double x) {
```

```
        if (x < XMIN || x > XMAX)  
            throw new IllegalArgumentException();  
    }
```

```
        this.x = x;
```

```
    }  
    public double getY() {  
        return y;  
    }
```

```
    public void setY(double y) {
```

```
        if (y < YMIN || x > YMAX)  
            throw new IllegalArgumentException();  
    }
```

```
        this.y = y;
```

```
    }
```

```
}
```

Warunkowe
zgłoszenie
wyjątku

Wyjątki (Exceptions)

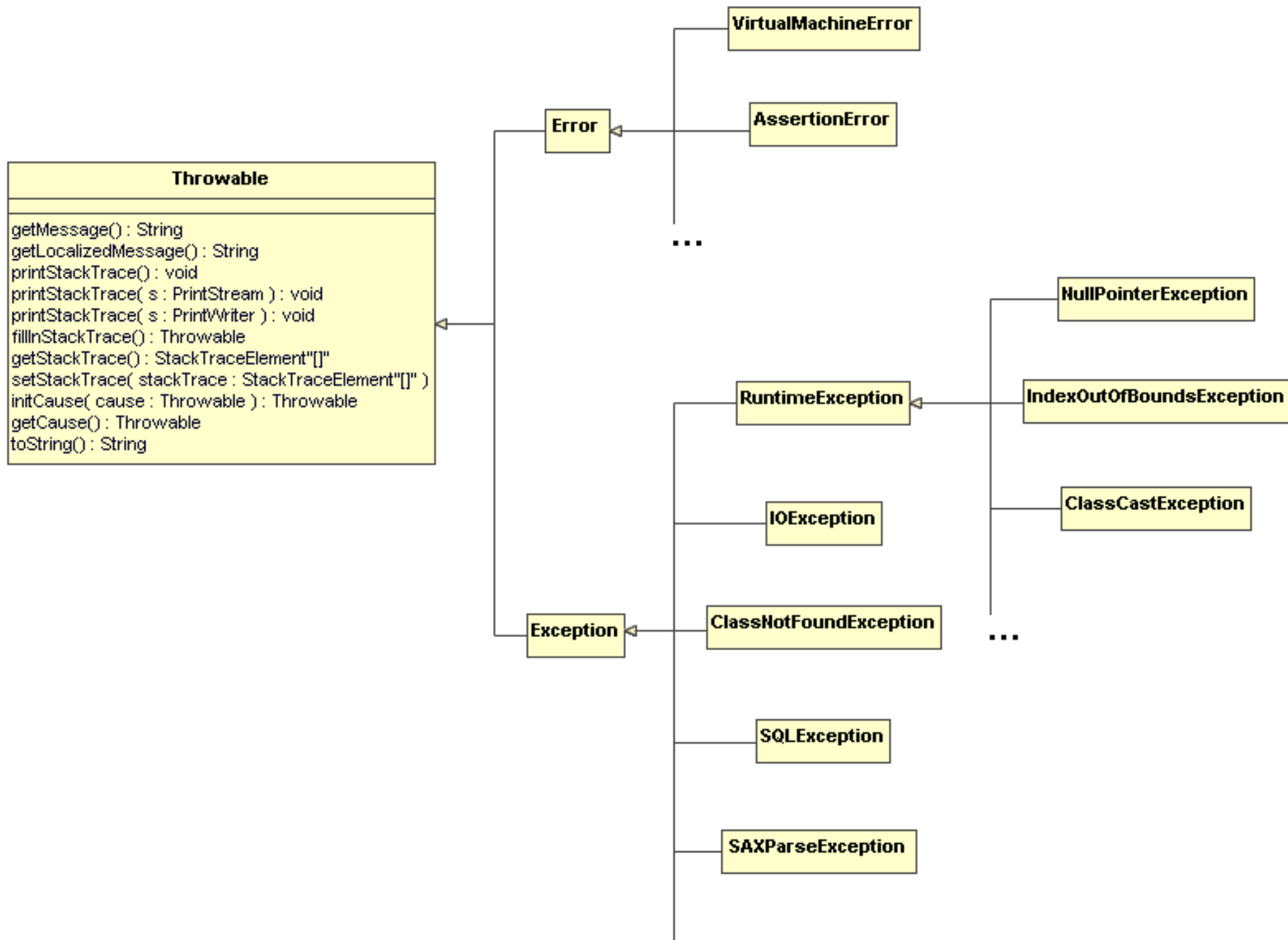
Wyjątek w Java jest obiektem, który opisuje sytuację błędną powstałą w kodzie. Zaistnienie sytuacji błędnej w metodzie powoduje utworzenie obiektu reprezentującego wyjątek i zgłoszenie go przez metodę, w której błąd wystąpił. Następnie metoda może sama obsłużyć wyjątek lub przesłać go do obsługi przez inne metody/obiekty.

Wyjątki mogą być zgłaszane przez maszynę wirtualną Javy lub przez kod użytkownika. Wyjątki zgłaszane przez maszynę wirtualną są związane z tzw. *błędami fatalnymi*, natomiast zgłaszane przez użytkownika z błędami związanymi z logiką programu.

Składnia programu obsługującego wyjątki bazuje na pięciu słowach kluczowych: **try**, **catch**, **throw**, **throws** i **finally**.

Monitorowany kod znajduje się w bloku **try**. **Zgłoszenie wyjątku** wewnątrz bloku **try** **powoduje powstanie obiektu-wyjątku** i **jego** ewentualne przejęcie przez odpowiedni blok **catch**.

- Wszystkie wyjątki i błędy są podklasami klasy **Throwable**.
- Klasa **Exception** i potomne, służą do opisywania sytuacji błędnych, które mogą być spowodowane przez kod użytkownika lub mogą być przez kod użytkownika wykryte i obsłużone.
- Klasa **Error** i potomne, są używane przez maszynę wirtualną do zgłaszania *błędów fatalnych, takich jak: przepełnienie stosu,*



Błędy - Error

Wyjątki dziedziczące po `Error` reprezentują poważne problemy, których aplikacja nie będzie w stanie rozwiązać. Przykładową podklasą jest *`VirtualMachineError`*. Wystąpienie takiego wyjątku oznacza, że maszyna wirtualna nie może dalej kontynuować pracy, np. z powodu wyczerpania się zasobów systemowych. Wyjątków rozszerzających *`Error`* nie należy przechwytywać, gdyż nie ma możliwości zaradzenia sytuacjom wyjątkowym, które je spowodowały. Z założenia te wyjątki mogą wystąpić praktycznie w każdej instrukcji kodu i nie muszą być wymieniane w klauzulach `throws`.

Wyjątki dziedziczące po `Exception` reprezentują sytuacje, na które dobrze napisana aplikacja powinna być przygotowana. To właśnie tę klasę rozszerza się tworząc własne rodzaje wyjątków. Jej przykładowe podklasy to:

`IOException` - sytuacje wyjątkowe związane z wejściem/wyjściem,
`ClassNotFoundException` - maszyna wirtualna nie odnalazła klasy o nazwie podanej w opisie wyjątku,
`SQLException` - wyjątki związane z dostępem do bazy danych
`SAXParseException`, która wskazuje, że podczas parsowania dokumentu XML wystąpił błąd.

Klasa RuntimeException

Bardzo ciekawą podklasą `Exception` jest `RuntimeException`, która sama posiada wiele podklas. Wyjątki rozszerzające `RuntimeException` mogą wystąpić podczas typowych operacji, jak rzutowanie zmiennej, odwołanie się do elementu tablicy lub odwołanie się do składowej obiektu. Ich wystąpienie zazwyczaj oznacza, że programista popełnił błąd w swoim kodzie lub nieumiejętnie korzystał z kodu napisanego przez innych. Maszyna wirtualna wykrywa wystąpienie takich błędów w trakcie działania programu i informuje o tym, zgłaszając odpowiedni wyjątek. Przykładowymi podklasami `RuntimeException` są:

`ClassCastException` - oznacza próbę rzutowania zmiennej na niepasujący typ,

`IndexOutOfBoundsException` - oznacza odwołanie się do indeksu z poza zakresu

`NullPointerException` - oznacza że zamiast referencji wskazującej na obiekt pojawiła się wartość null (np. obiekt nie utworzony)

`IllegalArgumentException` - oznacza, że do metody przekazany został niewłaściwy argument

Obsługa wyjątków: `try` - `catch`

W przypadku wystąpienia nieobsługiwanego wyjątku program kończy pracę.

Aby samemu obsłużyć błąd powodujący przerwanie programu należy umieścić go w bloku **`try {}`**, a następnie w bloku **`catch{} umieścić typy`** wyjątków, na które chcemy reagować oraz z wiązać z nimi kod obsługujący zgłoszony wyjątek.

Należy również pamiętać, że po obsłudze wyjątku przez blok **`try/catch`** program nie wraca do komendy następnej w bloku **`try`** lecz **przechodzi** do wykonania pierwszej instrukcji za blokiem **`try/catch`**.

Przechwycenie wyjątku dla klasy PunktPriv zdefiniowanej wcześniej:

```
public class PunktPrivTest {  
    public static void main(String[] args) {  
        PunktPriv p = null;  
        try{  
            p = new PunktPriv(120,100);  
            // zbyt duża wartość pierwszego argumentu  
        }  
        catch(IllegalArgumentException e) {  
            System.out.println("Argument poza zakresem");  
            e.printStackTrace();  
            return;  
        }  
    }  
}
```


Obsługa kilku wyjątków jednocześnie

```
try {  
    //kod który może zgłosić wyjątki  
}  
catch (TypWyjatk1 w) {  
    //obsługa wyjątków Typ1  
}  
catch (TypWyjatk2 w) {  
    //obsługa wyjątków Typ2  
}  
catch (TypWyjatk3 w) {  
    //obsługa wyjątków Typ3  
}  
finally{  
    //instrukcje – wykonane niezależnie od tego czy  
    wyjątek wystąpił, czy nie...  
}
```

```

public class Kilkawyjatkow {
    public static void main(String[] args) {
        int[] licznik = {1, 2, 3 , 0 };
        int mianownik[] = {2, 0, 1, 0, 5};
        double ulamek = 0.0;

        for (int i=0; i<6; i++){
            try{
                ulamek = (double) ( licznik[i]/mianownik[i] );
                //ulamek = (double) licznik[i]/mianownik[i]; // brak wyjatku
dzielenia przez zero
            }
            catch (IndexOutOfBoundsException e) {
                System.out.println("Indeks tablicy poza zakresem");
            }
            catch (ArithmeticException e) {
                System.out.println("Proba dzielenia przez zero");
            }
            finally {
                System.out.println("Blok finally wykonany zawsze");
                System.out.println("wartosc zmiennej ulamek: " + ulamek);
            }
        } //koniec petli for
    } // koniec metody main()
}

```

Jednoczesne przechwytywanie kilku wyjątków jednym blokiem **catch**

```
try {  
    //kod który może zgłosić wyjątki  
}  
catch (TypWyjatk1 | TypWyjatk2 w) {  
    //obsługa wyjątków Typ1 lub Typ2  
}
```

Jednoczesne przechwytywanie kilku wyjątków jednym blokiem **catch**

```
public class KilkaWyjatkow2 {  
    public static void main(String[] args) {  
        int[] licznik = {1, 2, 3 , 0 };  
        int mianownik[] = {2, 0, 1, 0, 5};  
        double ulamek = 0.0;  
  
        for (int i=0; i<6; i++){  
            try{  
                ulamek = (double)(licznik[i]/mianownik[i]);  
                //ulamek = (double) licznik[i]/mianownik[i];  
            }  
            catch (IndexOutOfBoundsException | ArithmeticException e) {  
                System.out.println("Przechwycony wyjatek: " +  
                    e.getClass().getName());  
            }  
  
        } //koniec petli for  
    } // koniec metody main()  
}
```

Zagnieżdżanie bloków try-catch

Instrukcja `try` może występować w bloku instrukcji innej instrukcji `try`.

Konstrukcja taka powoduje, że wyjątki zgłaszane przez wewnętrzny blok `try` będą posiadały swój kontekst wywołania, inny niż wyjątki bloku zewnętrznego.

Jeżeli wewnętrzny blok `try` zgłosi wyjątek, dla którego nie posiada odp. sekcji `catch`, będzie przeszukiwany kontekst bloku zewnętrznego w poszukiwaniu odp. sekcji `catch`.

```

public class ZagniezdzenieWyjatkow1{
    public static void main(String[] args) {

        int[] licznik = {1, 2, 3 , 0 };
        int mianownik[] = {2, 0, 1, 0, 5};
        double ulamek = 0.0;

        try{
            for (int i=0; i<6; i++){
                int l=0, m =0;
                try{
                    l = licznik[i];
                    m = mianownik[i];
                }
                catch (IndexOutOfBoundsException e) {
                    System.out.println("Indeks tablicy poza zakresem");
                    System.out.println("Wyjątek z bloku wewnętrznego -
przejdzie do kolejnej iteracji petli");
                }
                ulamek = (double) (l/m); // zgłosi wyjątek zewnętrzny
                //ulamek = (double) l / m; // nie zgłosi wyjątku zewn
                System.out.println(ulamek);
            } //koniec petli for
        }
        catch(ArithmeticException e){
            System.out.println("Proba dzielenia przez zero");
            System.out.println("Wyjątek z bloku zewnętrznego - koniec petli");
        }
    }
}

```

Zagnieżdżanie bloków try-catch

Zagnieżdżone instrukcje **try/catch** nie muszą występować w tak jawny sposób jak w poprzednim przykładzie.

Zagnieżdżenia takie otrzymamy również, jeśli wewnątrz bloku **try** wywołamy metodę zawierającą swoją własną instrukcję **try/catch**.

Instrukcja `throw`

Służy do zgłaszania wyjątków przez nasz program:

```
throw Obiekt_klasy_Throwable;
```

Obiekt ***Obiekt_klasy_Throwable musi być klasy **Throwable lub***** potomnej. Wykonanie komendy **throw** powoduje natychmiastowe przerwanie sekwencyjnego wykonania programu.

Wykonanie programu przenosi się do najbliższej sekcji obsługi zgłoszonego wyjątku.

Jeżeli takiej sekcji nie ma, to program zostanie zatrzymany, a domyślny program obsługi wypisze ścieżkę wywołań metod aż do zgłoszonego wyjątku.

```
public void setY(double y) {  
    if (y < YMIN || x > YMAX)  
        throw new IllegalArgumentException();  
    this.y = y;  
}
```


throws

Jeśli metoda zgłasza wyjątek, którego sama nie obsługuje, to deklaracja metody musi zawierać informację o tym.

Służy do tego słowo kluczowe **throws**, umieszczane po deklaracji metody, a po nim wymieniane są typy wszystkich wyjątków zgłaszanych przez metodę - za wyjątkiem **Error** i **RuntimeException** i ich podklas.

Ogólna postać definicji metody zgłaszającej nieobsługiwane wyjątki :

```
typ nazwa_metody(lista-parametrów) throws  
lista-wyjątków  
{  
// ciało metody  
}
```

Tworzenie własnych wyjątków

Java posiada wbudowane wyjątki obsługujące najczęściej spotykane błędy. Jednak często zachodzi potrzeba zdefiniowania nowych wyjątków specyficznych dla naszego programu.

Aby utworzyć nową klasę wyjątku należy zdefiniować klasę dziedziczącą po klasie **Exception**. Nowo zdefiniowana klasa nie musi nawet niczego implementować...

Najczęściej implementowane zmiany w klasach nowych wyjątków, to:

- dodatkowe zmienne instancyjne przechowujące stan sytuacji błędnej
- pokrywanie standardowych metod klasy **Throwable** takich, jak: **getLocalizedMessage()**, **getMessage()**, **printStackTrace()** i **toString()**

//deklaracja klasy własnego wyjątku

```

class LiczbaNieparzystaException extends Exception{
    int n;
    LiczbaNieparzystaException(int liczba){
        n = liczba;
    }
    public String toString(){
        return "Wyjątek! Liczba " + n + " jest nieparzysta";
    }
}

```

Usunąć `static` przy metodzie
`sprawdzParzystosc()` i

```

public class WlasnyWyjatek {

```

//deklaracja metody zgłaszającej wyjątek

```

static void sprawdzParzystosc(int liczba) throws LiczbaNieparzystaException{
    if (liczba %2 != 0 )
        throw new LiczbaNieparzystaException(liczba);
}

```

```

public static void main(String[] args) {

```

```

    for (int i = 1; i<10; i++){
        try {
            sprawdzParzystosc(i);
        } catch (LiczbaNieparzystaException e) {
            System.out.println(e);
        }
    }
}

```

Usunąć blok `try-catch` i
sprawdzić podpowiedzi Eclipse

```

}
}

```

```
public class WlasnyWyjatek {  
  
    //deklaracja metody zgłaszającej wyjątek  
    static void sprawdzParzystosc(int liczba) throws LiczbaNieparzystaException{  
        if (liczba %2 != 0 )  
            throw new LiczbaNieparzystaException(liczba);  
    }  
  
    public static void main(String[] args) throws LiczbaNieparzystaException {  
  
        for (int i = 1; i<10; i++){  
  
            sprawdzParzystosc(i);  
  
        }  
    }  
}
```

W tym przykładzie, w przypadku wystąpienia wyjątku w metodzie sprawdzParzystosc() zostanie on przekazany do metody main(), a co ca tym idzie obsługa będzie przez JVM (koniec programu)

Importowanie projektu z przykładami do Eclipse:
File -> Import -> General – Existing Projects into
Workspace -> archive file:

