



**Faculty
of Physics**

WARSAW UNIVERSITY OF TECHNOLOGY



Programowanie Obiektowe Java

Małgorzata Janik

Zakład Fizyki Jądrowej
malgorzata.janik@pw.edu.pl
<http://java.fizyka.pw.edu.pl/>



Modyfikatory

dostępu: (public, private, ...)
inne: (static, final, abstract, ...)

Interfejsy

co to jest?
nasłuchiwanie (listeners)
kolekcje

Struktura klasy

```
//deklaracje klas
[modyfikatory] class NazwaKlasy [extends
NazwaKlasyBazowej] [implements NazwaInterfejsu] {

    //deklaracje metod
    [modyfikatory] Typ1 metodaN(lista parametrów) {
        ...
        return obiektTypN;
    }

    //deklaracje pól
    [modyfikatory] Typ pole1;
    ...
    [modyfikatory] Typ poleN;
}
```



Modyfikatory dostępu

default

public

private

protected



Modyfikatory dostępu

- ang. Access (Control) Modifiers
- 4 modyfikatory które służą ustawianiu poziomu dostępności danej klasy / metody / zmiennej
 - **public** – widoczny i dostępny dla wszystkich
 - **private** – widoczny i dostępny tylko w obrębie danej klasy
 - **protected** – widoczny i dostępny w obrębie pakietu oraz dla wszystkich podklas
 - **domyślny (brak słowa kluczowego)** – widoczny i dostępny dla wszystkich tylko w obrębie pakietu



Modyfikatory dostępu

- ang. Access (Control) Modifiers
- 4 modyfikatory które służą ustawianiu poziomu dostępności danej klasy / metody / zmiennej
 - **public** – widoczny i dostępny dla wszystkich
 - dotyczy klas, metod, zmiennych
 - **private** – widoczny i dostępny tylko w obrębie danej klasy
 - dotyczy tylko składników klasy (np. metod i zmiennych)
 - klasy i interfejsy nie mogą być prywatne
 - **protected** – widoczny i dostępny w obrębie pakietu oraz dla wszystkich podklas (inaczej niż C++)
 - dotyczy metod, zmiennych; nie klas
 - **domyślny (brak słowa kluczowego)** – widoczny i dostępny dla wszystkich w obrębie pakietu



Modyfikatory dostępu

- Czy dany modyfikator dotyczy...?

Modyfikator	Klasa	Składnik klasy
default	tak	tak
public	tak	tak
private	-	tak
protected	-	tak



Modyfikatory dostępu

- Czy daną klasę / składnik widać w... ?

Modyfikator	Klasa	Pakiet	Podklasa (ten sam pakiet)	Podklasa (inny pakiet)	Wszyscy
public	tak	tak	tak	tak	tak
protected	tak	tak	tak	tak	-
default	tak	tak	tak	-	-
private	tak	-	-	-	-



Stosowanie widoczności

Zawsze używaj najbardziej ograniczającego modyfikatora, jaki nadal ma sens.

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

Więcej przydatnych sugestii można znaleźć w dyskusji:

<https://stackoverflow.com/questions/215497/in-java-difference-between-package-private-public-protected-and-private>



Pozostałe modyfikatory

- ang. Non-Access Modifiers lub *specifiers*
- 7 innych modyfikatorów
 - **static** – pola / metody statyczne, nie zależą od konkretnej instancji klasy
 - **final** – brak możliwości zmiany
 - **abstract** – klasy / metody abstrakcyjne – niedokończone, trzeba je uzupełnić w podklasie
 - **strictfp** – używane dla obliczeń zmiennoprzecinkowych; zapewnia, żeby wyniki się zgadzały na różnych platformach
 - **transient** – używane w przypadku serializacji obiektów, obiekty transient nie będą serializowane
 - **synchronized** i **volatile** – używane dla wątków



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.

```
double x = Math.sqrt(y);  
Color kolor = Color.WHITE;
```

- Do zdefiniowania takich elementów służy słowo kluczowe **static**.



Static

- **static** – statyczne

- Służy do tworzenia zmiennych i metod które nie zależą od konkretnej instancji klasy (konkretnego obiektu)
- Zawsze istnieje tylko jedna kopia danej **zmiennej statycznej** – jeśli dowolna instancja klasy ją zmodyfikuje, taką modyfikację będą widzieć wszystkie pozostałe instancje.
- **Metody statyczne** mogą wykorzystywać do obliczeń jedynie argumenty które przyjmują, oraz występujące w klasie pola statyczne. Nigdy nie wykorzystują “zwykłych” pól klasy, które mogą być różne dla różnych instancji.
- Do takich metod i zmiennych możemy odwoływać się poprzez nazwę klasy (nie konkretnej instancji) + nazwę zmiennej / metody:

- **Klasa.metoda()**
- **Klasa.zmienna**

```
double x = Math.sqrt(y);  
Color kolor = Color.WHITE;
```



Final

- **final** – końcowe - brak możliwości zmiany
 - Ogranicza modyfikacje danej zmiennej / metody / klasy
 - Zmienna z modyfikatorem final nie może zostać zmodyfikowana po tym, jak pierwszy raz uzyska konkretną wartość.
 - int, double, itp...
 - dobrą praktyką jest nazywać takie zmienne wielkimi literami
 - Referencja nie może się zmienić w ten sposób, by wskazywała na inny obiekt
 - **ALE** nadal możemy zmieniać wartości atrybutów danego obiektu,
 - Referencja = zmienne które inicjalizujemy przez użycie **new**.
 - Metoda z modyfikatorem final nie może zostać przeciążona (nadpisana) w podklasie
 - Klasa z modyfikatorem final nie może zostać rozszerzona (nie można z niej stworzyć podklasy)
- **final** często używany razem ze **static** by w danej klasie stworzyć stałą:
 - **public static final double PI = 3.14159265359;**



Abstract

- **abstract** – abstrakcyjne
- **Klasa abstrakcyjna jest to klasa niedokończona**
- **Obiekty** takiej klasy **nie mogą być tworzone**, może być natomiast dziedziczona – **klasa pochodna** “**uzupełnia**” definicję danej klasy.
- Jeśli klasa dziedziczy po klasie abstrakcyjnej i nie zapewni implementacji wszystkich metod z klasy nadrzędnej → musi również zostać oznaczona jako klasa abstrakcyjna.
- Może posiadać konstruktor, może on być jednak wywołany tylko przez klasy pochodne. Może również posiadać zwykłe metody i pola.
- Dodatkowo klasa abstrakcyjna może posiadać **metody abstrakcyjne**, metody takie posiadają listę argumentów, jednak **nie posiadają ciała**.

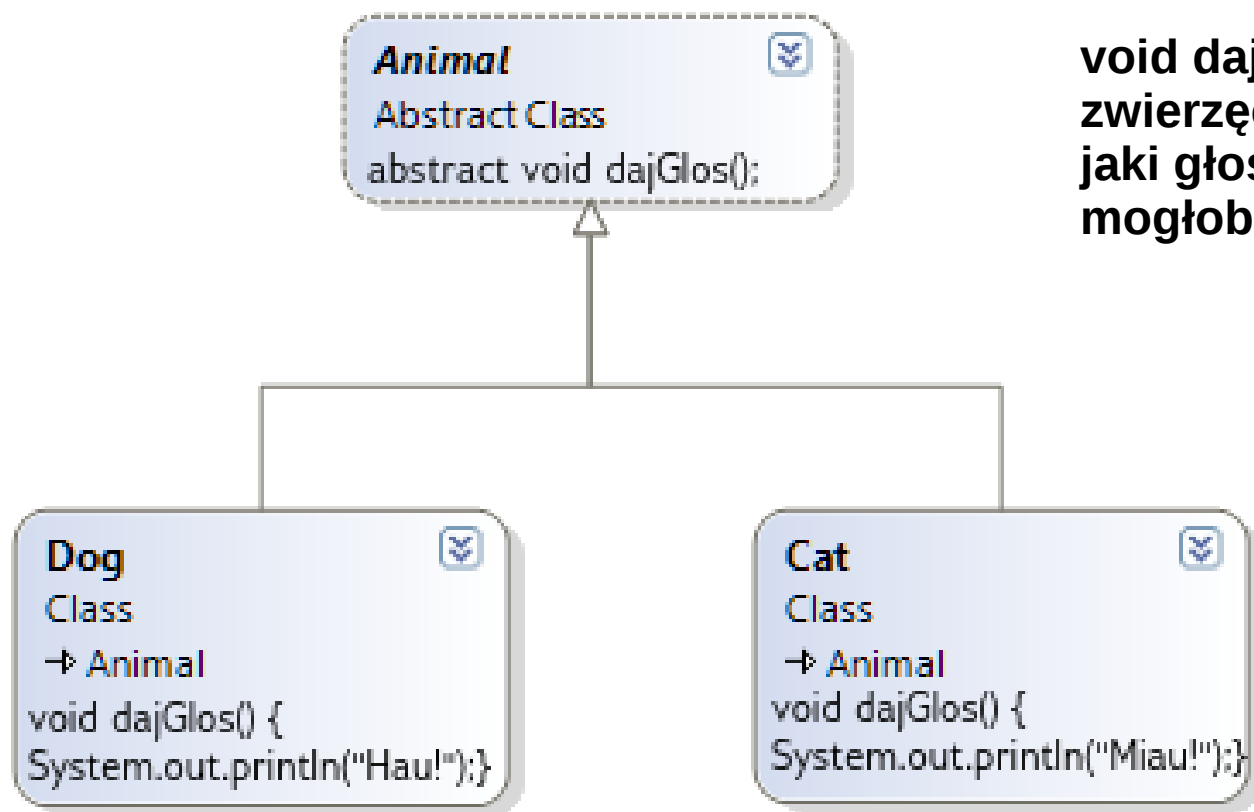


Abstract

- **abstract** – abstrakcyjne
- **Klasa abstrakcyjna jest to klasa niedokończona**
- **Obiekty** takiej klasy **nie mogą być tworzone**, może być natomiast dziedziczona – **klasa pochodna “uzupełnia” definicję danej klasy.**
- Jeśli klasa dziedziczy po klasie abstrakcyjnej i nie zapewni implementacji wszystkich metod z klasy nadrzędnej → musi również zostać oznaczona jako klasa abstrakcyjna.
- Może posiadać konstruktor, może on być jednak wywołany tylko przez klasy pochodne. Może również posiadać zwykłe metody i pola.
- Dodatkowo klasa abstrakcyjna może posiadać **metody abstrakcyjne**, metody takie posiadają listę argumentów, jednak **nie posiadają ciała.**

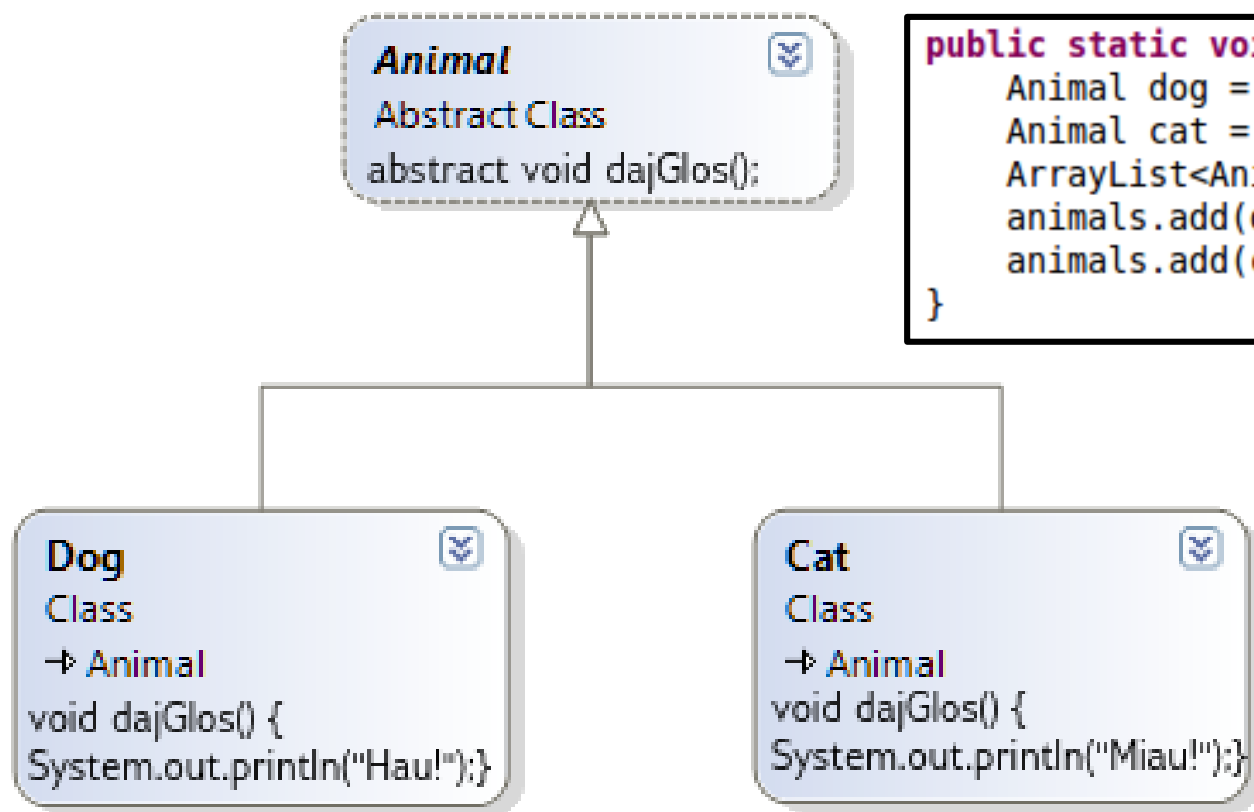


Klasy abstrakcyjne – po co?



`void dajGlos()` dla zwierzęcia nie ma sensu: jaki głos takie zwierze mogłoby wydawać?

Klasy abstrakcyjne – po co?



```
public static void main(String[] args) {
    Animal dog = new Dog();
    Animal cat = new Cat();
    ArrayList<Animal> animals = new ArrayList<Animal>()
    animals.add(dog);
    animals.add(cat);
}
```

Ale dzięki istnieniu klasy abstrakcyjnej możemy kotki i pieski łączyć:

- we wspólne kolekcje
- tak samo przekazywać do funkcji

Co więcej:

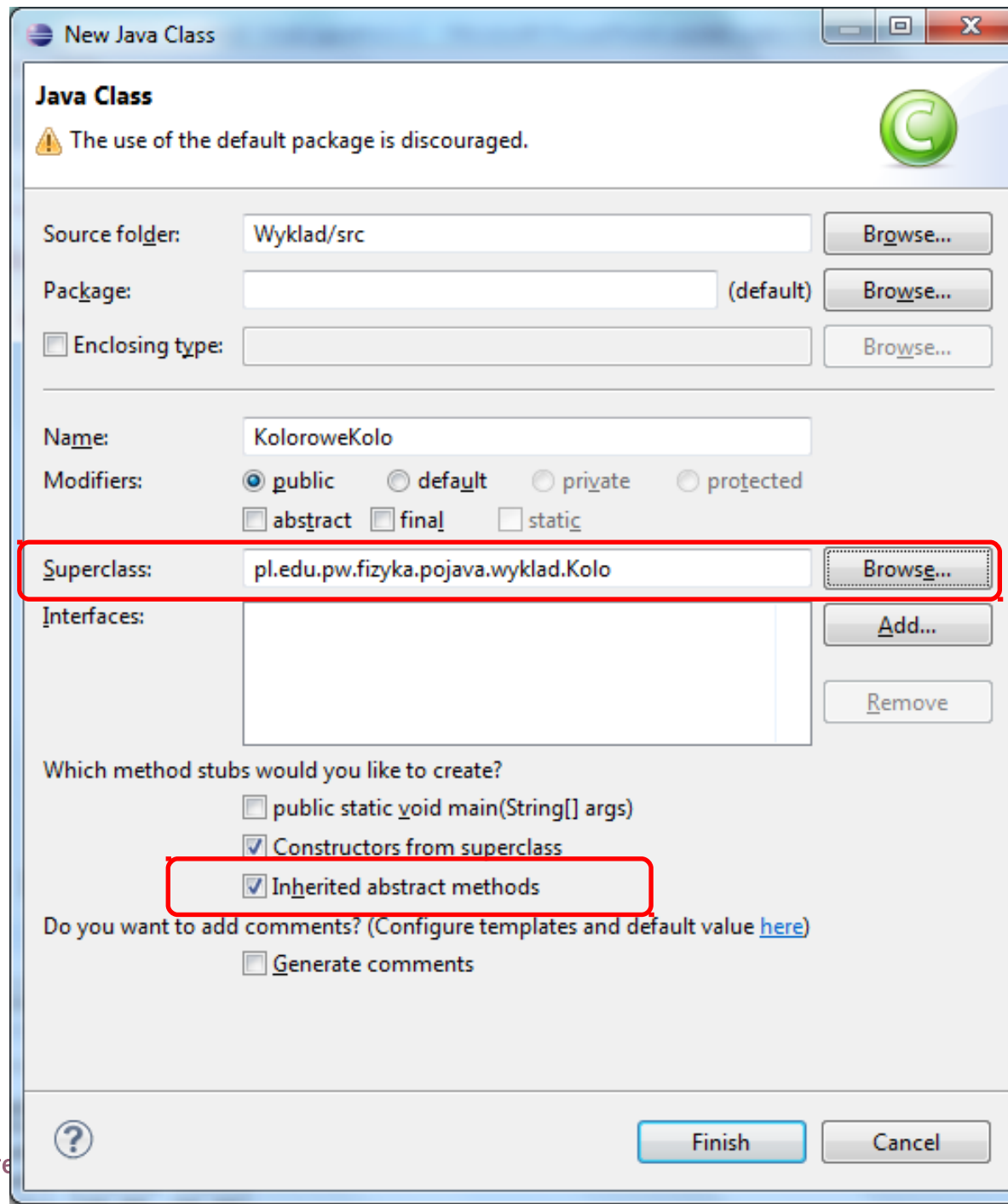
- jesteśmy pewni, że każde utworzone zwierze będzie umiało dać głos, bo metoda `dajGlos` na pewno będzie przeciążona

Abstract

- **abstract** – abstrakcyjne
 - klasa / metoda niedokończona - trzeba je uzupełnić w podklasie
 - **Klasy abstrakcyjne**
 - nie można stworzyć instancji (konkretnego obiektu) klasy abstrakcyjnej
 - powstaje wyjątek (InstantiationException)
 - klasa nie może być równocześnie abstract i final
 - Klasę abstrakcyjną trzeba dokończyć (rozszerzyć), aby jej użyć, a modyfikator final blokuje możliwość rozszerzania klasy
 - Jeśli klasa ma chociaż jedną metodę abstrakcyjną, sama też powinna zostać zadeklarowana jako abstrakcyjna
 - Odwrotność nie musi być prawdziwa: klasa abstrakcyjna nie musi zawierać metod abstrakcyjnych.
 - **Metody abstrakcyjne**
 - Metoda abstrakcyjna to metoda bez implementacji
 - Implementacja metody abstrakcyjnej powinna się znaleźć w podklasie.
 - Przykład: **public abstract exampleMethod();**



Szybkie tworzenie klasy pochodnej w Eclipse



abstract - przykład

```
public abstract class Zwierze {  
    protected String imie;  
  
    public Zwierze (String nazwij){  
        imie = nazwij;  
    }  
  
    String podajImie(){  
        return imie;  
    }  
    abstract String wydajGlos();  
}
```

```
public class Kot extends Zwierze {  
    public Kot(String nazwij) {  
        super(nazwij);  
    }  
    @Override  
    String wydajGlos() {  
        return "Miau";  
    }  
}
```



Synchronized i Volatile

- Związane z używaniem wielu wątków
- Synchronized – synchronizowana
 - Metoda może zostać wywołana w tym samym czasie tylko przez jeden wątek
- Volatile – zmienna
 - Zmienna może zostać asynchronicznie zmieniona przez inne wątki



Non-access modifiers

Modyfikator	Klasa	Metoda	Zmienna (pole)
static	-	tak	tak
final	tak	tak	tak
abstract	tak	tak	-
transient	-	-	tak
volatile	-	-	tak
synchronized	-	tak	-



Quiz time



WU

Pytanie 1

- Jeśli nie napiszemy **żadnego modyfikatora przed nazwą klasy** to wtedy domyślnie:
 - (a) Klasa będzie prywatna
 - (b) Klasa będzie publiczna
 - (c) Klasa będzie widoczna tylko w obrębie pakietu



Pytanie 1

- Jeśli nie napiszemy **żadnego modyfikatora przed nazwą klasy** to wtedy domyślnie:
 - (a) Klasa będzie prywatna
 - (b) Klasa będzie publiczna
 - (c) **Klasa będzie widoczna tylko w obrębie pakietu**



Pytanie 2

- Czy można stworzyć taką klasę:

```
abstract final class Test{  
    int pole;  
}
```

(a) Tak

(b) Nie



Pytanie 2

- Czy można stworzyć taką klasę:

```
abstract final class Test{  
    int pole;  
}
```

(a) Tak

(b) Nie



Pytanie 3

- A taką?

```
final class Test{  
    abstract int pole;  
}
```

(a) Tak

(b) Nie



Pytanie 3

- A taką?

```
final class Test{  
    abstract int pole;  
}
```

(a) Tak

(b) Nie



Pytanie 4

- A taką?

```
abstract class Test{  
    final int pole;  
}
```

(a) Tak

(b) Nie



Pytanie 4

- A taką?

```
abstract class Test{  
    final int pole;  
}
```

(a) **Tak**

(b) **Nie**



Pytanie 5

- A coś takiego:

```
abstract class Test{  
    abstract void metoda1();  
    static final stala = 10;  
    abstract static int metoda2();  
}
```

(a) Tak

(b) Nie



Pytanie 5

- A coś takiego:

```
abstract class Test{  
    abstract void metoda1();  
    static final stala = 10;  
    abstract static int metoda2();  
}
```

(a) **Tak**

(b) **Nie**



Nagroda?



Interfejsy



Interfejsy

- Interfejs jest zbiorem wymagań dotyczącym klas, które chcą się dostosować do danego interfejsu.
- Po co?
 - Zazwyczaj dostawca pewnych usług stwierdza “jeśli twoja klasa jest dopasowana do danego interfejsu, to wykonam usługę”.



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {  
    int compareTo(Object inny);  
}
```



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {  
    int compareTo(Object inny);  
}
```

- Oznacza to, że każda klasa implementująca interfejs `Comparable` musi:
 - Posiadać metodę `compareTo`
 - Ta metoda ma przyjmować argument typu `Object`
 - Ma zwracać liczbę całkowitą `int`



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {  
    int compareTo(Object inny);  
}
```

- Oznacza to, że każda klasa implementująca interfejs `Comparable` musi:

- Posiadać metodę `compareTo`

- Ta metoda ma przyjmować argument typu `Object`
- Ma zwracać liczbę całkowitą `int`



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {  
    int compareTo(Object inny);  
}
```

- Oznacza to, że każda klasa implementująca interfejs `Comparable` musi:

- **Posiadać metodę `compareTo`**

- Ta metoda ma przyjmować argument typu `Object`
- Ma zwracać liczbę całkowitą `int`



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {  
    int compareTo(Object inny);  
}
```

- Oznacza to, że każda klasa implementująca interfejs `Comparable` musi:

- Posiadać metodę `compareTo`

- Ta metoda ma przyjmować argument typu `Object`
- Ma zwracać liczbę całkowitą `int`



Interfejsy

- Konkretny przykład

- Metoda `sort` klasy `Arrays` posortuje obiekty tablicy, pod warunkiem, że te obiekty implementują interfejs `Comparable`.
- Jak wygląda interfejs `comparable`?

```
public interface Comparable {
```

```
    int compareTo(Object o);
```

```
}
```

- Oznacza to, że każda klasa implementująca interfejs `Comparable` musi posiadać metodę `compareTo`.

- Posiadać metodę

- Ta metoda musi być publiczna.
- Ma zwracać

To wymogi formalne.

Jeśli chcemy, żeby metoda działała poprawnie (faktycznie porównywała liczby)

to dla wywołania `x.compareTo(y)`, metoda powinna

- Jeśli $x < y$: zwracać liczbę < 0
- Jeśli $x = y$: zwracać liczbę 0
- Jeśli $x > y$: zwracać liczbę > 0

Interfejsy - przykład

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



Interfejsy - przykład

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



Interfejsy - przykład

```
public class Punkt implements Comparable{  
    double x = 0 , y = 0;
```

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

```
    int compareTo (Object inny) {
```

- Jeśli $x < y$: zwracać liczbę < 0
- Jeśli $x = y$: zwracać liczbę 0
- Jeśli $x > y$: zwracać liczbę > 0

```
        return 0;
```

```
    }
```

```
}
```



Interfejsy - przykład

```
public class Punkt implements Comparable{  
    double x = 0 , y = 0;
```

```
Punkt (double parameter1, double parameter2){  
    x = parameter1;  
    y = parameter2;  
}
```

- Jeśli $x < y$: zwracać liczbę < 0
- Jeśli $x = y$: zwracać liczbę 0
- Jeśli $x > y$: zwracać liczbę > 0

```
int compareTo(Object inny) {  
    Punkt innyPunkt = (Punkt) inny;  
    if (x*x+y*y < inny.x*inny.x+inny.y*inny.y)  
        return -1;  
    if (x*x+y*y > inny.x*inny.x+inny.y*inny.y)  
        return 1;  
    return 0;  
}
```



Interfejsy – przykład 2

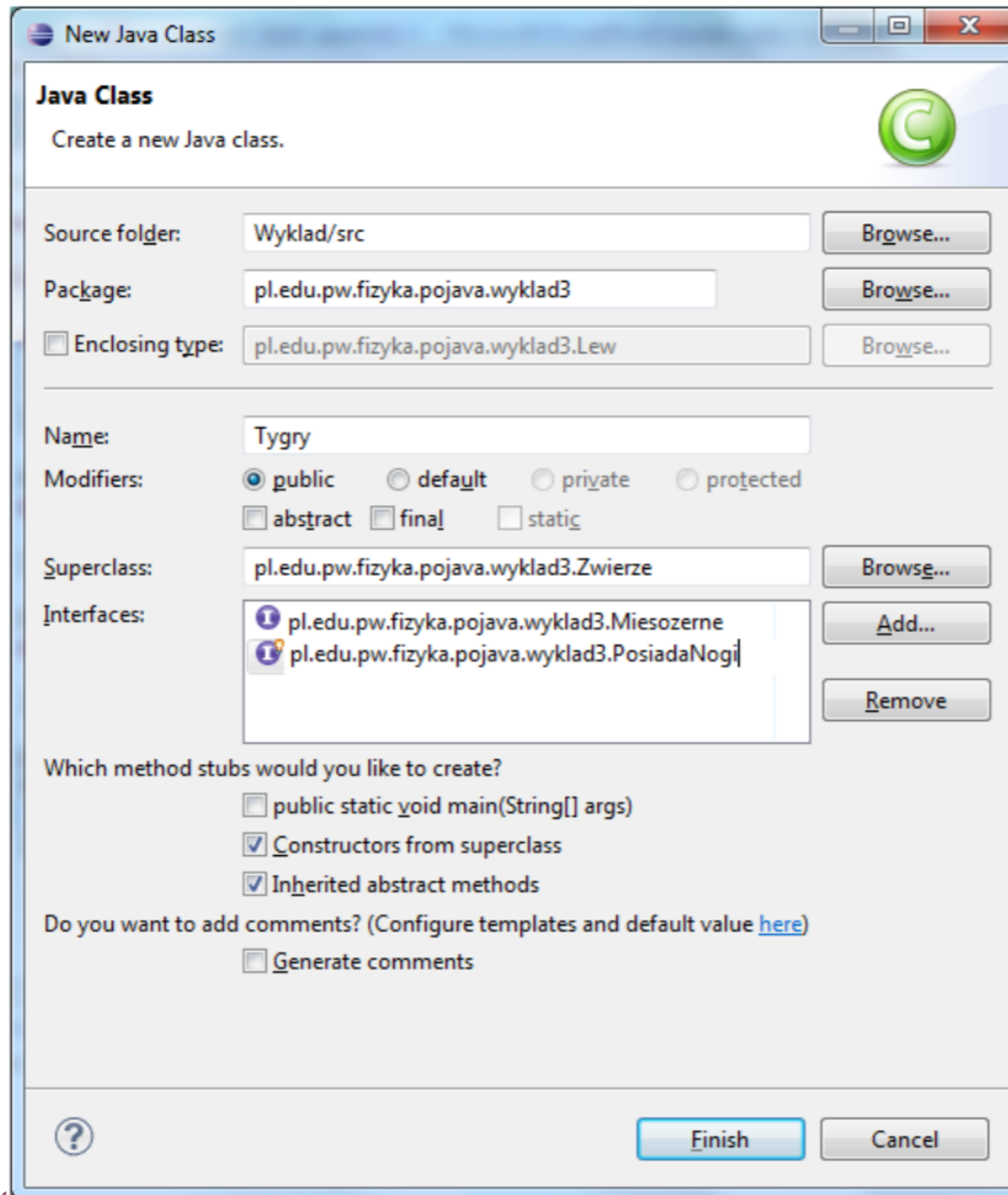
```
public interface Miesozerne {  
    String zjedzMieso();  
}
```

```
public interface PosiadaNogi {  
    int podajIloscNog();  
}
```

```
public class Lew extends Zwierze implements  
Miesozerne, PosiadaNogi {  
    public Lew(String nazwij) {  
        super(nazwij);  
    }  
    @Override  
    public int podajIloscNog() {  
        return 4;  
    }  
    @Override  
    public String zjedzMieso() {  
        return "Lew zjadł mięso";  
    }  
    @Override  
    String wydajGlos() {  
        return "Lew ryczy";  
    }  
}
```



Szybkie dodawanie interfejsów w Eclipse



Automatycznie wygenerowany kod

```
package pl.edu.pw.fizyka.pojojava.wyklad3;
public class Tygrys extends Zwierze implements Miesozerne, PosiadaNogi {
    public Tygrys(String nazwij) {
        super(nazwij);
        // TODO Auto-generated constructor stub
    }
    @Override
    public int podajIloscNog() {
        // TODO Auto-generated method stub
        return 0;
    }
    @Override
    public String zjedzMieso() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    String wydajGlos() {
        // TODO Auto-generated method stub
        return null;
    }
}
```



Interfejsy

- [Definicja] Interfejs w Javie to (deklarowany za pomocą słowa kluczowego **interface**) nazwany zbiór deklaracji zawierający:
 - publiczne abstrakcyjne metody (bez implementacji),
 - publiczne statyczne zmienne finalne (stałe) o ustalonych typach i wartościach.
- Implementacja interfejsu w klasie polega na zdefiniowaniu w tej klasie wszystkich metod zadeklarowanych w implementowanym interfejsie.



Interfejsy

- Ogólna postać definicji interfejsu w języku Java:

```
public interface NazwaInterfejsu {  
    typ nazwaZmiennej = wartosc;  
    ...  
    typ nazwaMetody(lista_parametrów);  
    ...  
}
```

- Uwagi:
 - modyfikator dostępu `public` przed słowem `interface` może nie występować (wówczas interfejs jest dostępny tylko w bieżącym pakiecie),
 - ewentualne **zmienne są zawsze typu `static final`** i mają przypisaną wartość stałą,
 - metody są ~~zawsze~~ abstrakcyjne (**bez implementacji**).



Metody domyślne

- Od Javy 8 istnieje możliwość zdefiniowania **metod domyślnych**.
- Zostały stworzone głównie z myślą zachowania kompatybilności
 - Jeśli chcemy rozszerzyć interfejs o nową metodę, ale nie chcemy modyfikować wszystkich klas które dotychczas implementowały dany interfejs
- Metody te mogą mieć właściwą implementację w ciele interfejsu i są one poprzedzone słowem kluczowym **default**:

```
public interface Pojazd {  
    public void start();  
    public void stop();  
    public void  
    ustawPredkosc(double[] v);  
    default String nazwa() {  
        return "Pojazd";  
    }  
}
```



Interfejsy a klasy abstrakcyjne

- Ten sam cel:
przygotować deklaracje metod które muszą zostać
dodane do klasy która
 - Rozszerza klasę (klasy abstrakcyjne)
 - Implementuje interfejs (interfejsy)
- Różnica:
 - Nie istnieje dziedziczenie wielokrotne w Javie.
ALE
 - Każda klasa może implementować tyle interfejsów, ile tylko chce.



Interfejsy

```
public interface Miesozerne {  
    String zjedzMieso();  
}
```

```
public interface PosiadaNogi {  
    int podajIloscNog();  
}
```

```
public class Lew extends Zwierze implements  
Miesozerne, PosiadaNogi {  
    public Lew(String nazwij) {  
        super(nazwij);  
    }  
    @Override  
    public int podajIloscNog() {  
        return 4;  
    }  
    @Override  
    public String zjedzMieso() {  
        return "Lew zjadł mięso";  
    }  
    @Override  
    String wydajGlos() {  
        return "Lew ryczy";  
    }  
}
```



Interfejsy a klasy abstrakcyjne

- **Różnica:**

- Nie istnieje dziedziczenie wielokrotne w Javie.

ALE

- Każda klasa może implementować tyle interfejsów, ile tylko chce.

- **Ponadto:**

- W interfejsach wszystkie metody są abstrakcyjne, natomiast w klasie abstrakcyjnej można stworzyć metody posiadające ciało, jak i abstrakcyjne.
- Klasa abstrakcyjna zazwyczaj jest mocno związana z klasami dziedziczącymi w sensie logicznym, interfejs natomiast nie musi być już tak mocno związany z daną klasą (określa jej cechy).



Interfejsy a klasy abstrakcyjne

Klasa abstrakcyjna	Interfejs
Klasy abstrakcyjne nie wspierają dziedziczenia wielokrotnego	Jedna klasa może implementować kilka interfejsów
Klasy abstrakcyjne mogą posiadać zarówno abstrakcyjne jak i zwykłe metody	Interfejsy zawierają tylko metody abstrakcyjne (w specjalnych przypadkach można użyć słowa default)
Klasa abstrakcyjna może mieć zmienne final bądź nie, statyczne bądź nie	Interfejsy posiadają tylko zmienne statyczne i finalne
Klasa abstrakcyjna może posiadać metody statyczne, metodę main oraz konstruktor.	Interfejsy nie mogą mieć metod statycznych, metody main ani konstruktorów
Klasa abstrakcyjna może implementować interfejs	Interfejs nie może zawierać implementacji klasy abstrakcyjnej
<pre>public abstract class Shape(){ public abstract void draw(); }</pre>	<pre>public interface Drawable{ void draw(); }</pre>



Interfejsy a klasy abstrakcyjne

- **Różnica:**

- Nie istnieje dziedziczenie wielokrotne w Javie.

ALE

- Każda klasa może implementować tyle interfejsów, ile tylko chce.

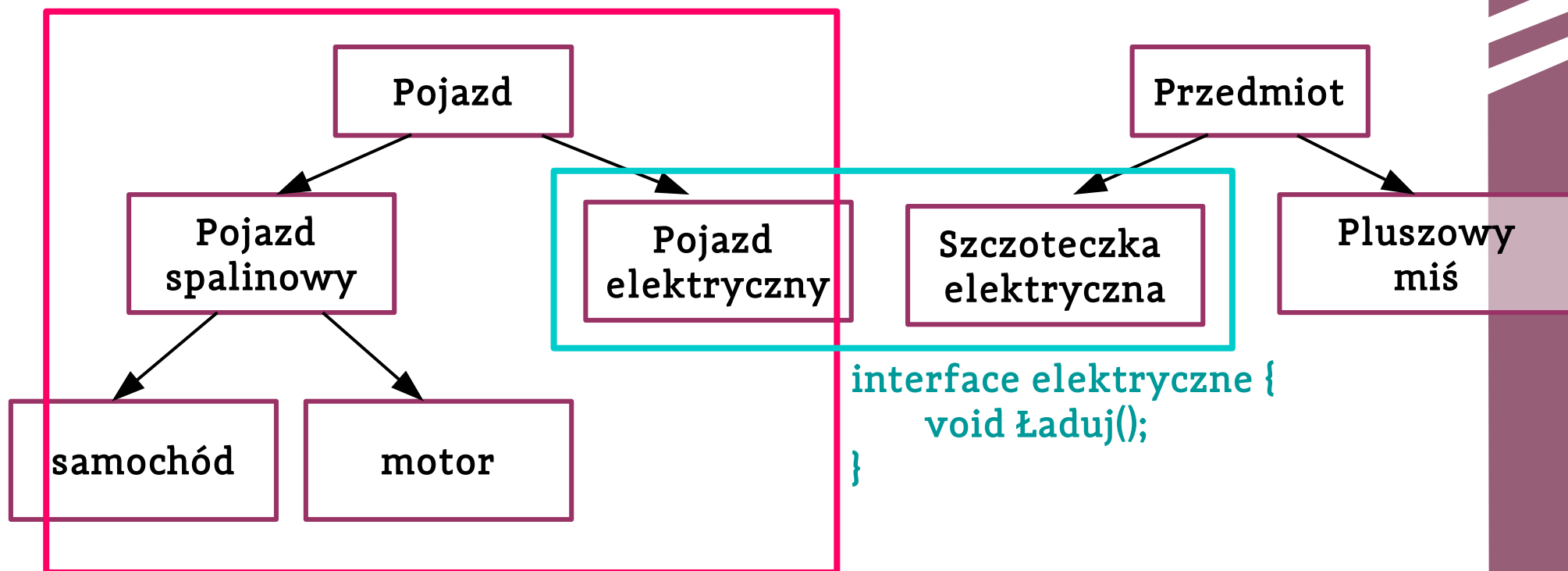
- **Ponadto:**

- W interfejsach wszystkie metody są abstrakcyjne, natomiast w klasie abstrakcyjnej można stworzyć metody posiadające ciało, jak i abstrakcyjne.
- Klasa abstrakcyjna zazwyczaj jest mocno związana z klasami dziedziczącymi w sensie logicznym, interfejs natomiast nie musi być już tak mocno związany z daną klasą (określa jej cechy).



Hierarchia klas

- Dziedziczenie – “pion”
- Interfejsy – “poziom”
(dwie klasy mogą implementować ten sam interfejs nie mając nic więcej wspólnego)



```
abstract class Pojazd {  
    void Jedź();  
}
```

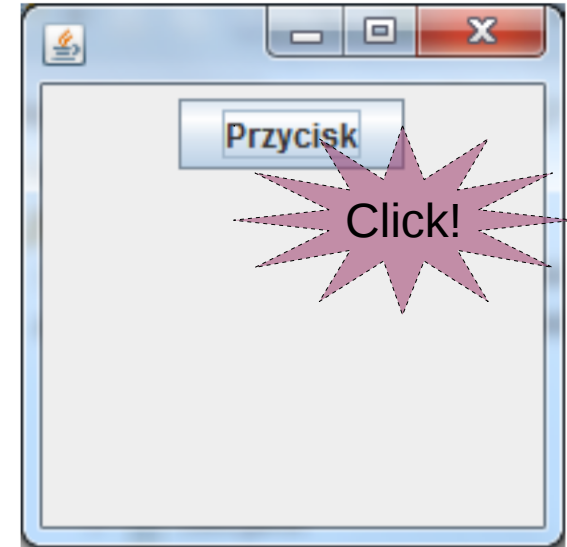
```
interface elektryczne {  
    void Ładuj();  
}
```



Nasłuchiwacze
(Listeners)
Obsługa zdarzeń



Nasłuchiwanie



- Dodawanie obsługi zdarzeń do komponentu
 - Programowanie GUI jest programowaniem zdarzeniowym.
 - Zdarzenia są generowane np. w momencie naciśnięcia klawisza lub kliknięcia myszą.
 - **W celu obsługi zdarzenia wykorzystywane są obiekty-słuchacze (ang. Listeners).**
 - Aby móc generować obiekty-słuchacze **klasa musi implementować interfejs nasłuchu**, który zawiera abstrakcyjne metody do obsługi zdarzeń.



Listenery

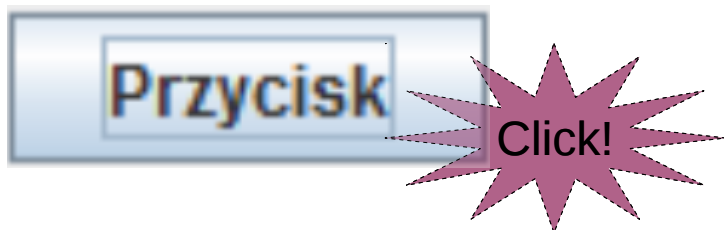
- Java dostarcza bogaty zestaw interfejsów nasłuchujących.
- Metody każdego z tych interfejsów umożliwiają reakcję na zdarzenie określonego typu.
- Klasy-słuchacze mogą implementować jeden lub kilka z tych interfejsów, zyskując w ten sposób zdolność do obsługi wybranych zestawów zdarzeń.



Przykłady nasłuchiwały

- **ActionListener**

- Kliknięcie przycisku



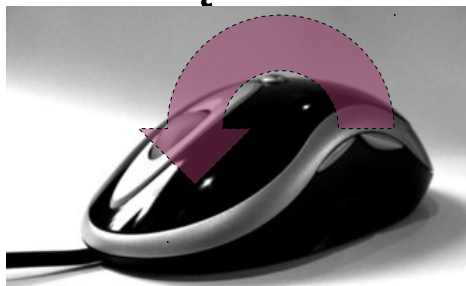
- **KeyListener**

- Naciśnięcie klawisza na klawiaturze



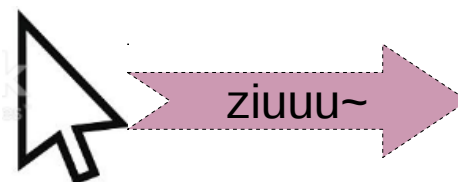
- **MouseListener**

- Naciśnięcie klawisza myszy



- **MouseMotionListener**

- przesunięcie wskaźnika myszy nad czymś



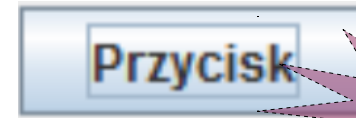
Dodawanie obsługi zdarzeń do komponentu

- ActionListener

- Kliknięcie przycisku

- Interfejs

```
public interface ActionListener{  
    void actionPerformed(ActionEvent e);  
}
```



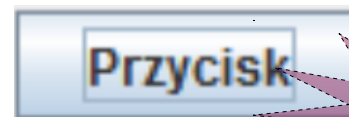
Dodawanie obsługi zdarzeń do komponentu

- ActionListener

– Kliknięcie przycisku

- Interfejs

```
public interface ActionListener{  
    void actionPerformed(ActionEvent e);  
}
```



- Należy dodać do klasy obsługującej zdarzenie

```
public class ObslugiwaczZdarzen implements ActionListener {  
    void actionPerformed(ActionEvent e)  
    {  
        //co ma zrobić naciśnięcie przycisku, np.  
        System.exit(0);  
    }  
}
```



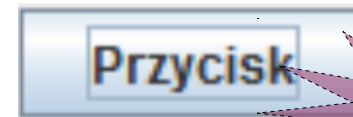
Dodawanie obsługi zdarzeń do komponentu

- ActionListener

– Kliknięcie przycisku

- Interfejs

```
public interface ActionListener{  
    void actionPerformed(ActionEvent e);  
}
```



- Należy dodać do klasy obsługującej zdarzenie

```
public class ObslugiwaczZdarzen implements ActionListener {  
    void actionPerformed(ActionEvent e)  
    {  
        //co ma zrobić naciśnięcie przycisku, np.  
        System.exit(0);  
    }  
}
```

- i dodać zdolność do nasłuchiwania komponentowi

button.addActionListener(obslugiwacz);



Dodawanie obsługi zdarzeń do komponentu

- Najważniejsze - do obsługi zdarzeń komponentu konieczna jest rejestracja dla danego komponentu obiektu klasy nasłuchującej:

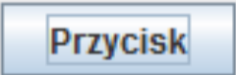
`źródłoZdarzeń.addRodzajListener (obiekt KlasyNasłuchującej);`

- Ta linijka oznacza, że
 - dla obsługi zdarzeń generowanych przez komponent **źródłoZdarzeń**,
 - zarejestrowano obiekt **obiektKlasyNasłuchującej**
 - implementujący interfejs nasłuchujący **RodzajListener**.



Dodawanie obsługi zdarzeń do komponentu

- Najważniejsze - do obsługi zdarzeń komponentu konieczna jest rejestracja dla danego komponentu obiektu klasy nasłuchującej:

```
źródłoZdarzeń.addRodzajListener (obiekt KlasyNasłuchującej);  
 .addActionListener (klasa implementująca interfejs  
ActionListener)
```

- Ta linijka oznacza, że
 - dla obsługi zdarzeń generowanych przez komponent **źródłoZdarzeń**,
 - zarejestrowano obiekt **obiektKlasyNasłuchującej**
 - implementujący interfejs nasłuchujący **RodzajListener**.



Dodawanie obsługi zdarzeń do komponentu

- Co jest czym (przykłady):

`źródłoZdarzeń.addRodzajListener (obiekt KlasyNasłuchującej);`

- Przycisk

 `.addActionListener` (klasa implementująca interfejs `ActionListener`)

- ScrollBar

 `.addAdjustmentListener` (klasa implementująca interfejs `AdjustmentListener`)

- Panel

 `.addMouseListener` (klasa implementująca interfejs `MouseListener`)

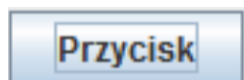


Dodawanie obsługi zdarzeń do komponentu

- Co jest czym (przykłady):

`źródłoZdarzeń.addRodzajListener (obiekt KlasyNasłuchującej);`

- Przycisk



`.addActionListener`

*A czym są klasy
implementujące interfejsy?*
(klasa implementująca interfejs
ActionListener)

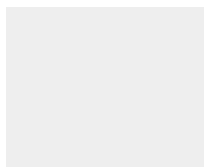
- ScrollBar



`.addAdjustmentListener`

(klasa implementująca interfejs
AdjustmentListener)

- Panel



`.addMouseListener`

(klasa implementująca interfejs
MouseListener)



Klasy obsługujące interfejsy

- **Wiele możliwości**

- Dowolna klasa może obsługiwać nasze zdarzenie pod warunkiem że implementuje odpowiedni interfejs

- Czyli np:

- Posiada metodę `void actionPerformed(ActionEvent e);`

```
class MoaNasluchiwacz implements ActionListener{  
    public void actionPerformed(ActionEvent arg0) {  
        System.out.println("Zewnętrzna");  
    }  
};
```

Taka klasa może powstać w tym samym pliku obok naszej klasy głównej, albo w nowym, oddzielnym pliku

```
Button b = new Button("Wypisz");  
MojNasluchiwacz nasluchiwacz1 = new MoaNasluchiwacz();  
b.addActionListener(nasluchiwacz1);
```



Klasy obsługujące interfejsy

- **Wiele możliwości**

- Dowolna klasa może obsługiwać nasze zdarzenie pod warunkiem że implementuje odpowiedni interfejs

- Czyli np:

- Posiada metodę `void actionPerformed(ActionEvent e);`

```
class MojaNasluchiwacz implements ActionListener{  
    public void actionPerformed(ActionEvent arg0) {  
        System.out.println("Zewnętrzna");  
    }  
};
```

Taka klasa może powstać w tym samym pliku obok naszej klasy głównej, albo w nowym, oddzielnym pliku

- **Brzmi jak dużo roboty**

- Za każdym razem tworzyć nową klasę jeśli chce się obsłużyć przycisk...



Anonimowa klasa wewnętrzna

- Przecież tak naprawdę nie potrzebujemy ani nazwy tej klasy...

```
ChangeListener nasluchiwacz1 = new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};  
  
b.addActionListener(nasluchiwacz1);
```

Anonimowa klasa wewnętrzna:
sposób z Laboratorium 2

- Ani nawet specjalnej zmiennej na nią!

```
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```

To samo, tylko jeszcze mniej rozwlekle



Klasy anonimowe wewnętrzne

- Klasy anonimowe

- ang. *anonymus inner class*

sprawiają, że można skrócić pisany kod

- w tym samym czasie można zadeklarować oraz stworzyć instancję danej klasy.

- Są jak klasy wewnętrzne **bez nazwy**.

- Używane wtedy, jeśli potrzeba stworzyć instancję klasy lokalnej **tylko jeden raz**.



Klasy anonimowe wewnętrzne

- Klasy anonimowe można stworzyć na dwa sposoby:

- rozszerzenie **klasy**

```
Klasa cos = new Klasa(){  
    //rozszerzenie klasy  
};
```

```
class MyThread  
{  
    public static void main(String[] args)  
    {  
        Thread t = new Thread()  
        {  
            public void run()  
            {  
                System.out.println("Child Thread");  
            }  
        };  
        t.start();  
        System.out.println("Main Thread");  
    }  
}
```

- użycie **interfejsu**

```
ActionListener nasluchiwacz = new ActionListener()  
{  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wnetrzna");  
    }  
};  
b.addActionListener(nasluchiwacz);
```

```
Interfejs cos = new Interfejs(){  
    //implementacja interfejsu  
};
```



Dodawanie nasłuchu

- Wiele możliwości, czyli...
- Można zrobić to samo na 5 różnych sposobów:
 - 1) Klasa wewnętrzna (nazwana)
 - 2) Anonimowa klasa wewnętrzna
 - 3) Klasa zewnętrzna – zmienne przez konstruktor
 - 4) Klasa zewnętrzna – zmienne przez metody set
 - 5) Interfejs dodawany do klasy w której jest komponent
– (...i jeszcze kilka innych)
- Gdzie te linijki umieszczać w kodzie?



(1) Klasa wewnętrzna

```
public class ObslugaZdarzen1 extends JFrame {

    //Zmienna wewnetrzna klasy
    int i = 10;
    // definicja klasy wewnętrznej - zwróć uwagę na "widoczność" zmiennych
    class MojInterfejs implements ActionListener{
        public void actionPerformed(ActionEvent arg0) {
            setTitle("Klasa wewnetrzna - wartość zmiennej i: " + i );
        }
    }

    public ObslugaZdarzen1() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        MojInterfejs mI = new MojInterfejs();
        b.addActionListener(mI);
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen1();
        f.setVisible(true);
    }
}
```



(1) Klasa wewnętrzna

```
public class ObslugaZdarzen1 extends JFrame {

    //Zmienna wewnętrzna klasy
    int i = 10;

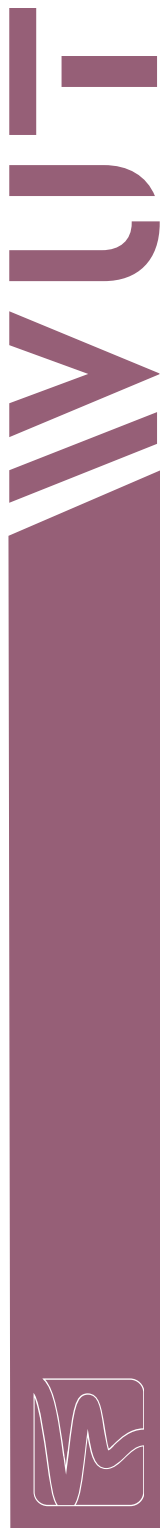
    // definicja klasy wewnętrznej - zwróć uwagę na "widoczność" zmiennych
    class MojInterfejs implements ActionListener{
        public void actionPerformed(ActionEvent arg0) {
            setTitle("Klasa wewnętrzna - wartość zmiennej i: " + i );
        }
    }

    public ObslugaZdarzen1() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener(new MojInterfejs());
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen1();
        f.setVisible(true);
    }
}
```



(2) Anonimowa klasa wewnętrzna



```
public class ObslugaZdarzen2 extends JFrame {

    public ObslugaZdarzen2() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                setTitle("Anonimowa klasa wewnetrzna");
            }
        });
        add(b);
    }

    public static void main(String[] args) {
        JFrame f = new ObslugaZdarzen2();
        f.setVisible(true);
    }
}
```

Klasa wewnętrzna → zewnętrzna

```
public class ObslugaZdarzen1 extends JFrame {
```

```
    //Zmienna wewnętrzna klasy  
    int i = 10;
```

```
    // definicja klasy wewnętrznej — zwróć uwagę na "widoczność" zmiennych
```

```
    class MojInterfejs implements ActionListener{  
        public void actionPerformed(ActionEvent arg0) {  
            setTitle("Klasa wewnętrzna - wartość zmiennej i: " + i );  
        }  
    }
```

```
    public ObslugaZdarzen1() throws HeadlessException {  
        super();  
        setSize(600,200);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());  
        JButton b = new JButton("Przycisk");  
        b.addActionListener(new MojInterfejs());  
        add(b);
```

```
    }  
    public static void main(String[] args) {  
        JFrame f = new ObslugaZdarzen1();  
        f.setVisible(true);  
    }
```

```
}
```

→ do zewnętrznej klasy

(w tym samym bądź
innym pliku)



(3) Klasa zewnętrzna – zmienna przekazywana przez konstruktor

```
public class MojInterfejsPubliczny implements ActionListener {  
    JFrame referencjaDoOkna;  
  
    MojInterfejsPubliczny(JFrame zmiennaPrzekazanaWKonstruktorze){  
        referencjaDoOkna = zmiennaPrzekazanaWKonstruktorze;  
    }  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        referencjaDoOkna.setTitle("Interfejs w innej klasie  
                                publicznej");  
    }  
}
```



(3) Klasa zewnętrzna – zmienna przekazywana przez konstruktor

```
public class MojInterfejsPubliczny implements ActionListener {  
    JFrame referencjaDoOkna;  
  
    MojInterfejsPubliczny(JFrame zmiennaPrzekazanaWKonstruktorze){  
        referencjaDoOkna = zmiennaPrzekazanaWKonstruktorze;  
    }  
}
```

```
public class ObslugaZdarzen3 extends JFrame {  
  
    public ObslugaZdarzen3() throws HeadlessException {  
        super();  
        setSize(600,200);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());  
        JButton b = new JButton("Przycisk");  
        MojInterfejsPubliczny mIP = new MojInterfejsPubliczny(this);  
        b.addActionListener(mIP);  
        add(b);  
    }  
    public static void main(String[] args) {  
        JFrame f = new ObslugaZdarzen3();  
        f.setVisible(true);  
    }  
}
```



(4) Klasa zewnętrzna – zmienne przekazywane przez metody set, get

```
public class ObslugaZdarzen4 extends JFrame {  
  
    public ObslugaZdarzen4() throws HeadlessException {  
        super();  
        setSize(600,200);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        setLayout(new FlowLayout());  
        JButton b = new JButton("Przycisk");  
        MojInterfejsPubliczny2 mIP2 = new MojInterfejsPubliczny2();  
        mIP2.setReferencjaDoOkna(this);  
        b.addActionListener(mIP2);  
        add(b);  
    }  
    public static void main(String[] args) {  
        JFrame f = new ObslugaZdarzen4();  
        f.setVisible(true);  
    }  
}
```



(4) Klasa zewnętrzna – zmienne przekazywane przez metody set, get

```
public class MojInterfejsPubliczny2 implements ActionListener {
    JFrame referencjaDoOkna;
    String nowyTytul = "Inna klasa publiczna - przekazywanie zmiennych przez metody
set... i get...";

    @Override
    public void actionPerformed(ActionEvent e) {
        referencjaDoOkna.setTitle(nowyTytul);
    }

    // przekazanie referencji do okna, ktorego nazwa ma byc zmieniona, metoda set...
    void setReferencjaDoOkna(JFrame zmiennaPrzekazana){
        referencjaDoOkna = zmiennaPrzekazana;
    }

    //dodatkowe metody get... i set... definiuje się w razie potrzeby w podobny sposób:
    void setTextInterfejsu(String przekazanyTekst){
        nowyTytul = przekazanyTekst;
    }

    String getTekstInterfejsu(){
        return nowyTytul;
    }
}
```



(5) Interfejs w klasie do której dodawany jest komponent

```
public class ObsługaZdarzen5 extends JFrame implements ActionListener {

    public ObsługaZdarzen5() throws HeadlessException {
        super();
        setSize(600,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JButton b = new JButton("Przycisk");
        b.addActionListener(this);
        add(b);
    }
    public static void main(String[] args) {
        JFrame f = new ObsługaZdarzen5();
        f.setVisible(true);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        setTitle("Interfejs implementowany w tej samej klasie...");
    }
}
```





Więcej na laboratorium 3

Obsługa kontrolek będzie się
przewijać do końca zajęć.

Dodawanie obsługi zdarzeń do komponentu

- Przykłady:

- Przycisk

 `.addActionListener` (klasa implementująca interfejs `ActionListener`)

- ScrollBar

 `.addAdjustmentListener` (klasa implementująca interfejs `AdjustmentListener`)

- Panel

 `.addMouseListener` (klasa implementująca interfejs `MouseListener`)

Wybrane interfejsy-zarządcy obsługi zdarzeń

- **ActionListener**
 - obsługuje zdarzenia **generowane przez użytkownika** na rzecz danego składnika interfejsu (np. kliknięcie przycisku)
- **AdjustmentListener**
 - obsługuje zdarzenie jako **zmianę stanu składnika** (np. przesuwanie suwaka w polu tekstowym)
- **KeyListener**
 - obsługuje zdarzenie np. od **wpisywania tekstu z klawiatury**
- **MouseListener**
 - obsługuje zdarzenie od **nacisnięcia klawiszy myszy**
- **MouseMotionListener**
 - obsługuje zdarzenie od **przesuwania wskaźnika myszy** nad danym składnikiem
- **ItemListener**
 - obsługuje zdarzenie od np. zaznaczenia pola wyboru
- **WindowListener**
 - obsługuje zdarzenie od okna np. minimalizacja, maksymalizacja, przesunięcie, zamknięcie
- **FocusListener**
 - obsługuje zdarzenie od przejścia składnika w stan aktywny/nieaktywny



Wiązanie wybranych składników z obsługą zdarzeń

- `addActionListener()`
 - dla `JButton`, `JCheckBox`, `JComboBox`, `TextField`, `JRadioButton`
- `addAdjustmentListene()`
 - dla `JScrollBar`
- `addItemListener()`
 - dla `JCheckBox`, `JComboBox`, `TextField`, `JRadioButton`
- `addFocusListener()`
 - dla **wszystkich** składników Swing
- `addKeyListener()`
 - dla **wszystkich** składników Swing
- `addMouseListener()`
 - dla **wszystkich** składników Swing
- `addMouseMotionListener()`
 - dla **wszystkich** składników Swing
- `addWindowListener()`
 - dla wszystkich obiektów typu **JFrame** oraz **JWindow**
- Metody te muszą być zastosowane przed wstawieniem składnika do kontenera (`JFrame`, `JPanel`, ...)



Ogólny schemat dodawania i obsługi zdarzeń komponentów

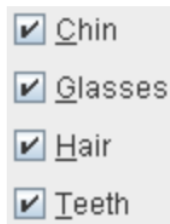
- deklaracja zmiennej (Klasa obiekt;)
- tworzenie nowego obiektu
(obiekt = new Klasa(parametry_kons);)
- ustawianie właściwości komponentu (preferowany rozmiar, tekst, kolor...)
- dodawanie interfejsu do obiektu:
obiekt.addNazwaInterfejsu(obiekt_implementujący_interfejs);
- dodawanie obiektu do okna lub np. panelu :
add(obiekt), panel.add(obiekt), ...
- odpowiednia modyfikacja metody obsługującej zdarzenia generowane przez obiekt w obiekcie_implementującym_interfejs



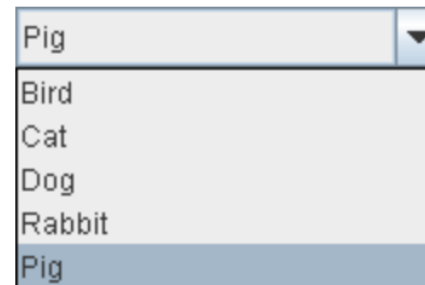
Wybrane kontrolki Swing (podstawowe)



[JButton](#)



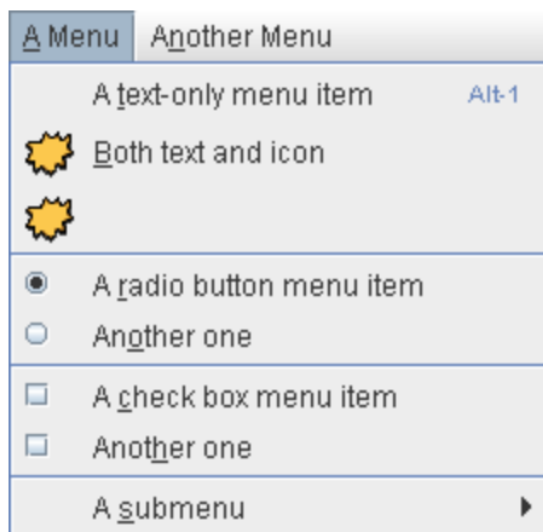
[JCheckBox](#)



[JComboBox](#)



[JList](#)



[JMenu](#)



[JRadioButton](#)



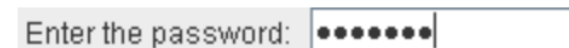
[JSlider](#)



[JSpinner](#)



[JTextField](#)



[JPasswordField](#)

Więcej na stronie:

<http://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html>



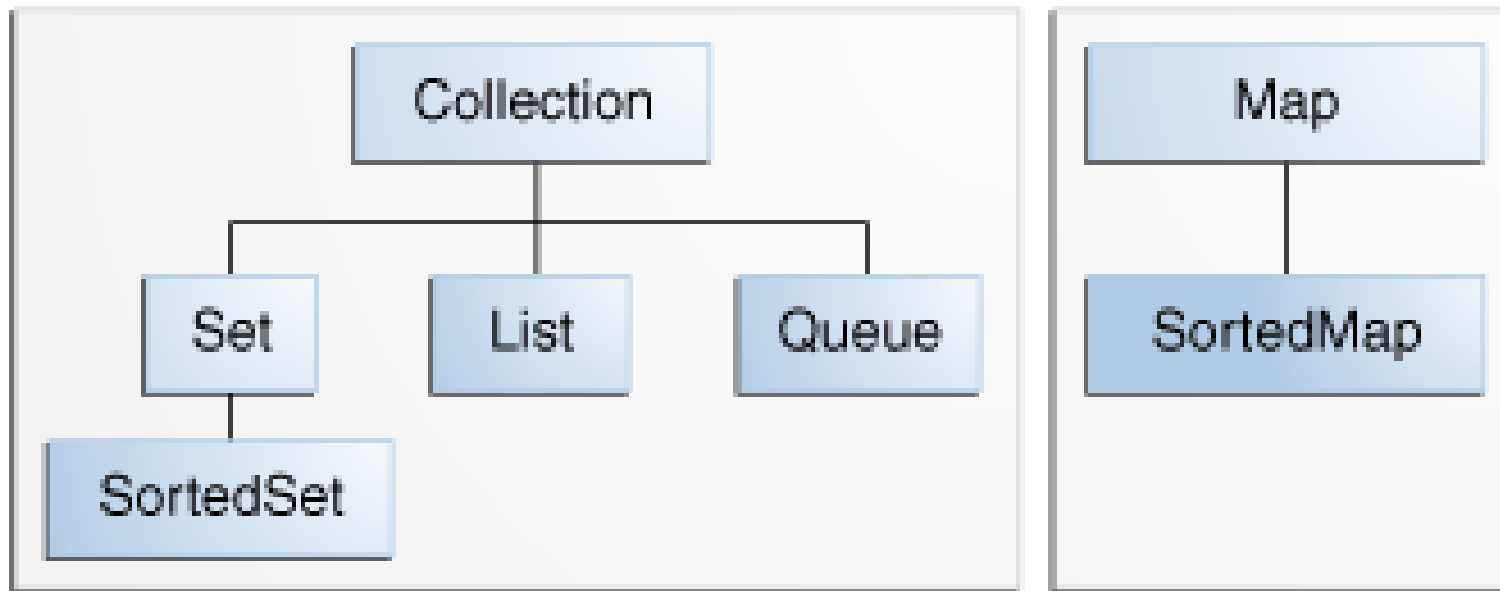
Kolekcje

Krótko o kolekcjach



Dlaczego kolekcje?

- Przechowywanie obiektów w tablicach ma zasadniczą wadę: konieczność deklarowania rozmiaru tablicy w momencie jej tworzenia
- Java zapewnia zestaw interfejsów do efektywnego „przechowywania” obiektów



Przykład

Stwórzmy dwie proste klasy dziedziczące z innej:

```
class Zwierze {  
    private static long counter;  
    private final long id = counter++;  
    public long id(){return id;}  
}  
  
class Kot extends Zwierze{  
    public String miaucz(){return "Miauuu";}  
}  
  
class Pies extends Zwierze {  
    public String szczekaj(){return „Hau!";}  
}
```



Przykład Kolekcje1.java

```
ArrayList zwierzeta = new ArrayList();

for (int i =0; i<5; i ++ ) zwierzeta.add(new Kot());

for (int j =0; j<5; j ++ ) zwierzeta.add(new Pies());

for (int k = 0; k< zwierzeta.size(); k++){
    System.out.println( ((Zwierze)zwierzeta.get(k)).id());
}
// rzutowanie (Zwierze) pozwala na skorzystanie z metody id -
// jednak "efekt" rzutowania musi być ujęty w dodatkowy
// nawias:
// ( (Zwierze)zwierzeta.get(k) ).id()
```



Przykład Kolekcje2.java

// Dla kolekcji można zdefiniować typ przechowywanych obiektów:

```
ArrayList<Zwierze> zwierzeta = new ArrayList<Zwierze>();
```

```
for (int i =0; i<5; i ++ ) zwierzeta.add(new Kot());
```

```
for (int j =0; j<5; j ++ ) zwierzeta.add(new Pies());
```

```
for (int k = 0; k< zwierzeta.size(); k++){  
    System.out.println( zwierzeta.get(k).id());  
}
```

// w tym wypadku dostęp do metody id nie wymaga rzutowania

//powyższą pętlę można zapisać "bardziej elegancko":

```
for(Zwierze z : zwierzeta){  
    System.out.println( z.id() );  
}
```



Pętla for each - ForEach.java

służy do iteracji po kolejnych elementach tablicy lub kolekcji,
ogólna składnia:

```
for ({deklaracja zmiennej pętli} : {kolekcja lub tablica}) {  
    {ciało pętli}  
}
```

Np.:

```
String[] teksty = {"jeden", "dwa", "trzy" };  
for (String s : teksty) System.out.println(s);
```

```
ArrayList<Kot> koty = new ArrayList<Kot>();  
koty.add(new Kot() ); koty.add(new Kot() );  
koty.add(new Kot() ); koty.add(new Kot() );
```

```
for(Kot k : koty) System.out.println("Kot id: " + k.id() );
```



Przykład Kolekcje3.java

```
ArrayList<Kot> koty = new ArrayList<Kot>();  
// czasem deklarując zmienne stosuje się bardziej ogólne  
// interfejsy  
// co może m.in. ułatwiać ewentualne zmiany implementacji,  
// np.:  
List<Pies> psy = new ArrayList<Pies>();  
Collection<Zwierze> zwierzeta = new ArrayList<Zwierze>();  
  
for (int i = 0; i < 5; i++) {  
    koty.add(new Kot());  
    psy.add(new Pies());  
}
```



Konwersja tablic na kolekcję i odwrotnie – Kolekcje4.java

```
Integer[] tablicaLiczby = {1,2,3,4,5};
```

```
List<Integer> listaLiczby = new ArrayList<Integer>();  
//List<Integer> listaLiczby2 = Arrays.asList(6,7,8);
```

```
listaLiczby.addAll( Arrays.asList(tablicaLiczby) );
```

```
// Całą kolekcję można łatwo wypisać:  
System.out.println(listaLiczby);
```

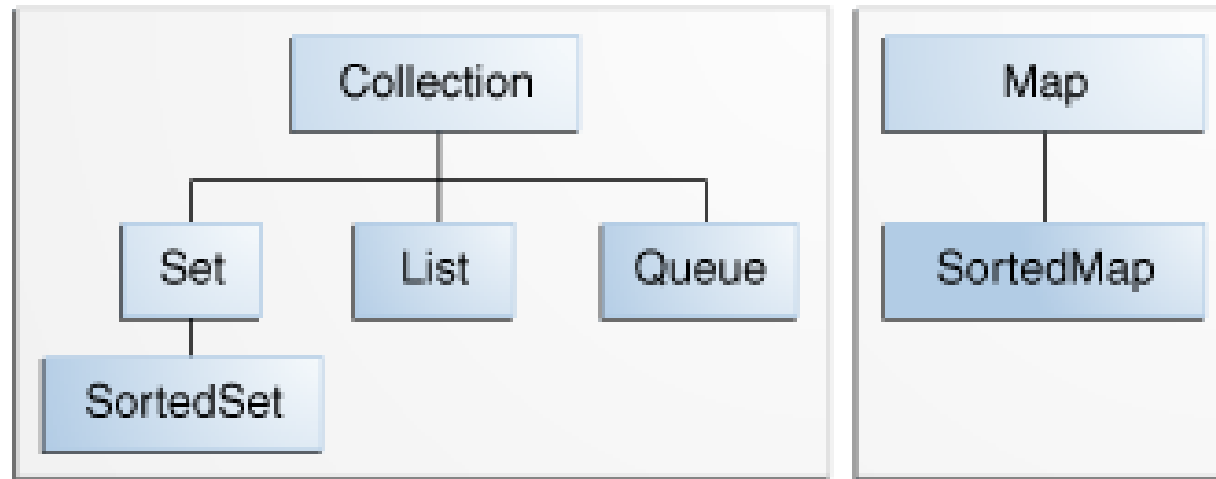
```
Integer[] innaTablica =  
    listaLiczby.toArray(new Integer[listaLiczby.size()]);
```

```
//Dla tablic takie wyświetlanie nie działa:  
System.out.println(innaTablica);
```

```
for (int b : innaTablica) System.out.println(b);
```



Podstawowe interfejsy



- Set – nie może być duplikatów
- List – elementy w określonej kolejności
- Queue – uporządkowane zgodnie z dyscypliną kolejki
- Map – grupa par obiektów klucz-wartość

Metody „wspólne” – interfejs Collection



Modifier and Type	Method and Description
boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
boolean	addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	clear() Removes all of the elements from this collection (optional operation).
boolean	contains(Object o) Returns <code>true</code> if this collection contains the specified element.
boolean	containsAll(Collection<?> c) Returns <code>true</code> if this collection contains all of the elements in the specified collection.
boolean	equals(Object o) Compares the specified object with this collection for equality.



Metody „wspólne” – interfejs Collection



int	hashCode() Returns the hash code value for this collection.
boolean	isEmpty() Returns true if this collection contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this collection.
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).



Metody „wspólne” – interfejs Collection

<code>int</code>	<code>size()</code> Returns the number of elements in this collection.
<code>Object[]</code>	<code>toArray()</code> Returns an array containing all of the elements in this collection.
<code><T> T[]</code>	<code>toArray(T[] a)</code> Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Więcej na:

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

Warto zwrócić uwagę, że część metod jest traktowanych jako operację „opcjonalne” – nie wszystkie są implementowane w interfejsach/klasach pochodnych



List – wybrane metody

dostęp pozycyjny do elementów

- Object get(int indeks)
- Object set(int indeks)
- Object add(int indeks)
- Object remove(int indeks)

wyszukiwanie

- int indexOf(Object obiekt)
- int lastIndexOf(Object obiekt)

rozszerzona iteracja

- ListIterator listIterator()

widok przedziałowy

- List subList(int poczatek, int koniec)

LinkedListDemo.java



Przenoszenie projektów

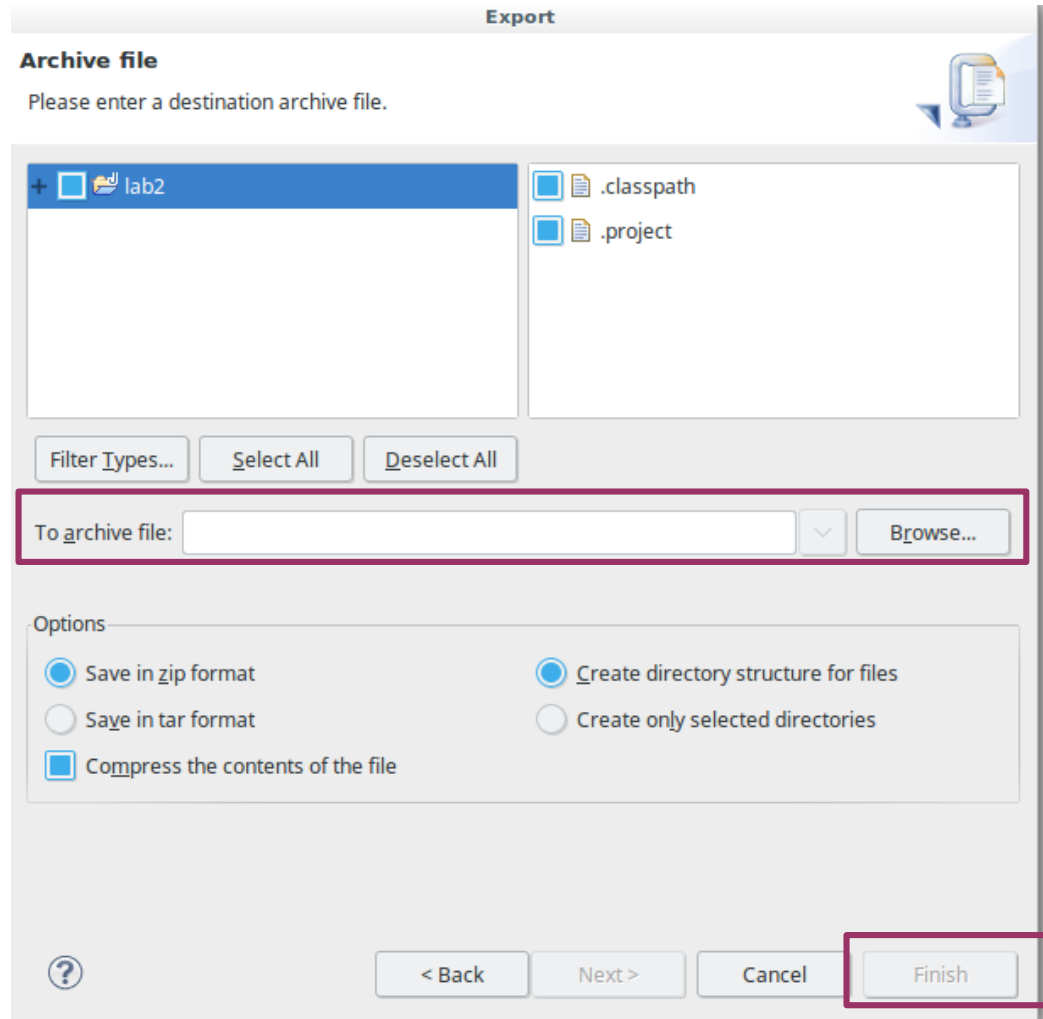
Import & Export



Eksportowanie projektu do pliku zip:

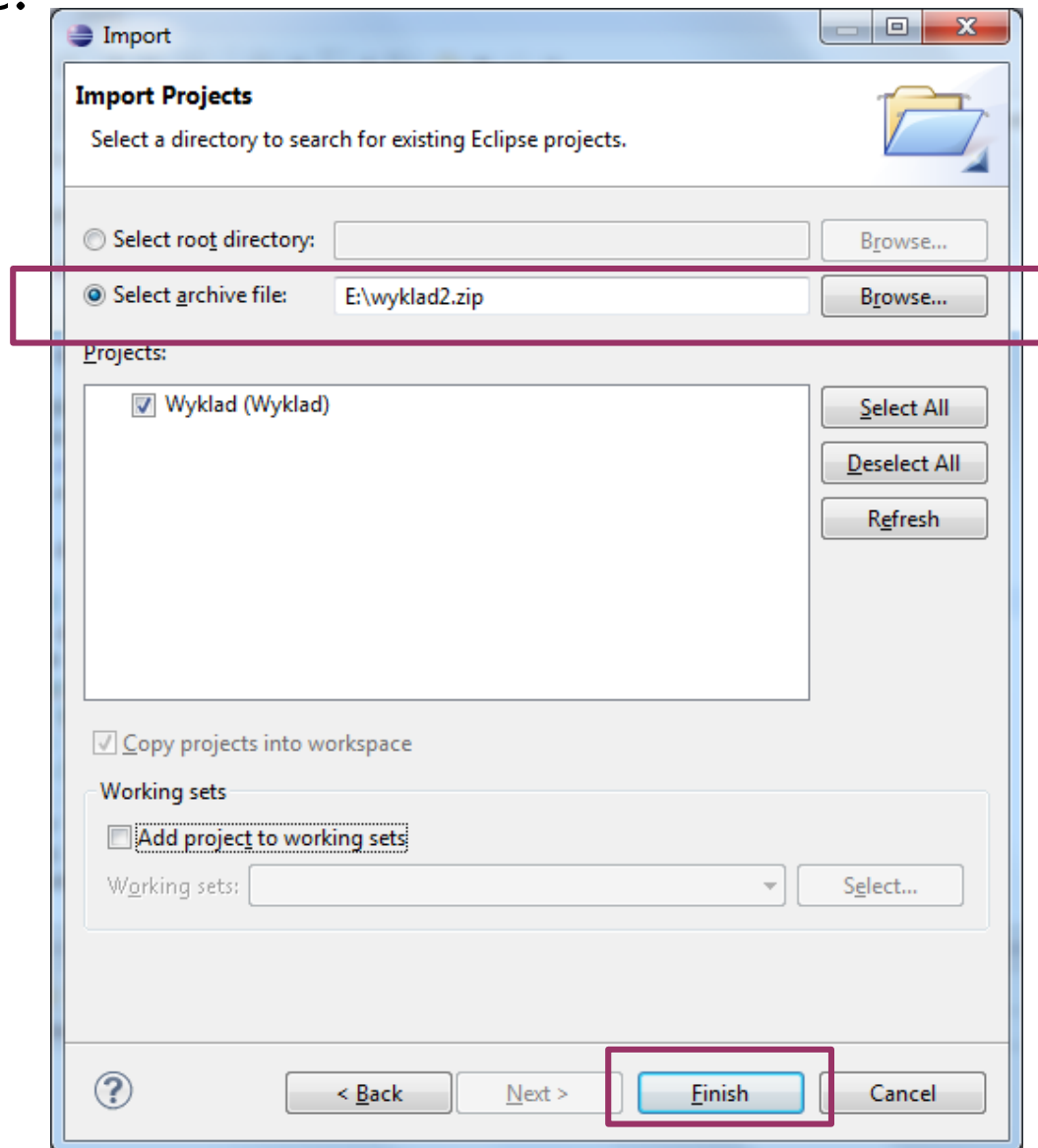
File → Export → General – Archive File →

To archive file: (podać ścieżkę przy użyciu przycisku Browse). →
Finish



Importowanie projektu do Eclipse:

File -> Import -> General – Existing Projects into Workspace
-> archive file:



Ciekawostki

Formatowanie tekstu



```
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");
```



Dziękuję za Uwagę!

**Do zobaczenia za tydzień.
Wreszcie pojawią się wyjątki
- i może łatwiej będzie wam zrozumieć
wyrzucane przez Eclipse'a błędy
+
Powtórzymy tworzenie GUI**



Więcej o modyfikatorach

- https://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm
- <http://javaconceptoftheday.com/access-and-non-access-modifiers-in-java/>



Ciekawostki

5 implementacji interfejsów



Anonimowa klasa wewnętrzna

```
JButton b = new JButton ("Zmien nazwe okna");  
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```

(1)

```
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```



Anonimowa klasa wewnętrzna

```
JButton b = new JButton ("Zmien nazwe okna");
b.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        setTitle("Anonimowa klasa wewnetrzna");
    }
});
```

(1)

```
b.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        setTitle("Anonimowa klasa wewnetrzna");
    }
});
```

To samo, tylko
bardziej rozwlekle:

(2)

```
ActionListener nasluchiwacz1 = new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        setTitle("Anonimowa klasa wewnetrzna");
    }
};
b.addActionListener(nasluchiwacz1);
```



Anonimowa klasa wewnętrzna

(1) `b.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent arg0) {
 setTitle("Anonimowa klasa wewnetrzna");
 }
});`

To samo, tylko
bardziej rozwlekle:

(2) `ActionListener nasluchiwacz1 = new ActionListener() {
 public void actionPerformed(ActionEvent arg0) {
 setTitle("Anonimowa klasa wewnetrzna");
 }
};
b.addActionListener(nasluchiwacz1);`



(nie-anonimowa) klasa wewnętrzna

(1)

```
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```

To samo, tylko
bardziej rozwlekłe:

(2)

```
ActionListener nasluchiwacz1 = new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};
```

```
b.addActionListener(nasluchiwacz1);
```

To samo, tylko nazwana
klasa wewnętrzna:

(3)

```
class MojNasluchiwacz implements ActionListener{  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};  
MojNasluchiwacz nasluchiwacz1 = new MojNasluchiwacz();  
b.addActionListener(nasluchiwacz1);
```



(nie-anonimowa) klasa wewnętrzna

(1)

```
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```

To samo, tylko
bardziej rozwlekłe:

(2)

```
ActionListener nasluchiwacz1 = new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};
```

```
b.addActionListener(nasluchiwacz1);
```

To samo, tylko nazwana
klasa wewnętrzna:

(3)

```
class MojNasluchiwacz implements ActionListener{  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};  
MojNasluchiwacz nasluchiwacz1 = new MojNasluchiwacz();  
b.addActionListener(nasluchiwacz1);
```

ActionListener jest
INTERFEJSEM, a interfejsy
implementujemy w klasie
poprzez słowo
"implements"



Klasa zewnętrzna

```
b.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
});
```

To samo, tylko
bardziej rozwlekłe:

```
ActionListener nasluchiwacz1 = new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        setTitle("Anonimowa klasa wewnetrzna");  
    }  
};
```

```
b.addActionListener(nasluchiwacz1);
```

Możemy wyrzucić na
zewnątrz:

```
(4) class MoaNasluchiwacz implements ActionListener{  
    public void actionPerformed(ActionEvent arg0) {  
(5)        System.out.println("Zewnętrzna");  
    }  
};
```

```
MojNasluchiwacz nasluchiwacz1 = new MoaNasluchiwacz();
```

```
b.addActionListener(nasluchiwacz1);
```

Nowa klasa może powstać w
tym samym pliku (4) obok
naszej klasy głównej, albo w
nowym (5), oddzielnym
pliku

