Programowanie urządzeń mobilnych

dr inż. Andrzej Grosser
na podstawie wykładu
dr inż. Juliusza Mikody

Konstrukcja obiektów

blok inicjalizacji

- Blok inicjalizacji to kolejna metoda nadawania początkowej wartości składowym klasy.
- Blok inicjalizacji wykonywany jest przed konstruktorem.
- W bloku inicjalizacji nie można wywołać konstruktora klasy nadrzędnej (poprzez super).

```
class Klasa{
    final int i,j;
    // blok inicjalizacji
    {
          System.out.println("Blok inicjalizacyjny");
          i = 10; // możliwa inicjalizacja zmiennych finalnych
    }
    // konstruktor
    Klasa() {
          System.out.println("Konstruktor");
          i = 10; // BŁĄD - próba ponownej inicjalizacji zmiennej finalnej
          j = 10;
    }
}
Klasa k = new Klasa();
// Blok inicjalizacyjny
// Konstruktor
```

Gdy wywoływany jest konstruktor . . .

1. Wszystkim polom składowym nadawane są wartości domyślne.
2. Tworzony jest obiekt klasy nadrzędnej.
3. Jeśli w pierwszej linii konstruktora znajduje się wywołanie innego konstruktora jest on wykonywany.
4. Wykonywane są bezpośrednie inicjalizacje pól składowych oraz bloki inicjalizacji w takiej kolejności w jakiej występują w deklaracji klasy.
5. Wykonywany jest kod wywołanego konstruktora.

```
class A{
    { System.out.println("Blok ini. A"); }
    A(){ System.out.println("Konstr. A"); };
}

class B extends A{
    B(int i){
        super();
        System.out.println("Konstr. B z
parametrem");
    }
    { System.out.println("Blok ini. B"); }
    B(){
        this(0);
        System.out.println("Konstr. B");
    }
}
B b = new B();
```

```
Blok ini. A
Konstr. A
Blok ini. B
Konstr. B z param.
Konstr. B
```

Gdy wywoływany jest konstruktor

```
class A{
    { System.out.println("Blok ini. A"); }
    A(){ System.out.println("Konstr. A"); }
    A(int i){ System.out.println("Konstr. A z
param."); }
}
class B extends A{
    { System.out.println("Blok ini. B"); }
    B(){ this(1); System.out.println("Konstr. B"); };
    B(int i){ System.out.println("Konstr. B z
param."); };
}
class C {
    { System.out.println("Blok ini. C"); }
    C(){ System.out.println("Konstr. C"); };
}
class D extends B{
    A a = new A(1);
    { System.out.println("Blok ini. D nr 1"); }
    C c = new C();
    { System.out.println("Blok ini. D nr 2"); }
    D(){ System.out.println("Konstr. D"); };
}

D d = new D();
```

```
Blok ini. A
Konstr. A
Blok ini. B
Konstr. B z param.
Konstr. B
Blok ini. A
Konstr. A z param.
Blok ini. D nr 1
Blok ini. C
Konstr. C
Blok ini. D nr 2
Konstr. D
```

Niszczenie obiektów

w teorii

W języku Java nie ma destruktorów - Java automatycznie usuwa zbędne dane, ręczna dealokacja nie jest potrzebna. Często konieczne jest jednak zwolnienie zasobów systemowych, np. zamknięcie plików.

`finalize`

- metoda wywoływana przed zniszczeniem obiektu,
- programista nie jest w stanie określić kiedy dokładnie metoda ta zostanie wykonana - **nie można polegać na wywołaniu tej metody!**

Jeśli konieczne jest zwolnienie zasobów należy utworzyć metodę `dispose` i jawnie ją wywołać w momencie gdy chcemy zwolnić zasoby.

Jeśli klasa zawiera pole składowe, które posiada metodę `dispose`, należy również napisać metodę `dispose`, która będzie uruchamiała metodę `dispose` należącą do tego pola oraz wywołać tą metodę.

Niszczenie Obiektów

w praktyce

```
import java.awt.Window;
class MojaKlasa{
    Window w; // wiemy, że Window ma metodę dispose
    /*
     * ...
     */
    public void dispose(){
          w.dispose(); // wywołanie metody dispose dla pola składowego
    }
}

MojaKlasa mk = new MojaKlasa();
/*
 * ...
 */
mk.dispose(); // wywołanie metody dispose dla klasy MojaKlasa
```

Rzutowanie

niebezpieczne

```
class A{
    void f_a(){ System.out.print("jestem f_a z klasy A"); }
}
class B extends A{
    void f_b(){ System.out.print("jestem f_b"); }
    void f_a(){ System.out.print("jestem f_a z klasy B"); }
}

A a = new A(); B b = new B();
a.f_a(); // jestem f_a z klasy A
B b0 = a; // błąd
B b1 = (B)a; // Kompilacja OK, ale..
/* Exception in thread "main" java.lang.ClassCastException:
   A cannot be cast to B at Test.main(Test.java:17) */

a = b;
a.f_a(); // jestem f_a z klasy B
a.f_b(); // błąd
b.f_a(); // jestem f_a z klasy B
B b2 = a; // błąd
B b3 = (B)a; // jest OK, bo a wskazuje na obiekt klasy B
b3.f_a(); // jestem f_a z klasy B
b3.f_b(); // jestem f_b
```

Rzutowanie

bezpieczne

`instanceof` - operator pozwalający na sprawdzenie czy obiekt należy do danej klasy

```
A a = new A(); B b = new B();

if (a instanceof B) {  // jeśli a jest klasy B
    System.out.println("1: a jest klasy B - rzutowanie możliwe");
    B b1 = (B)a; // rzutowanie jest OK
} else { System.out.println("1: a nie jest klasy B - rzutowanie niemożliwe"); }
a = b;
if (a instanceof B) {  // jeśli a jest klasy B
    System.out.println("2: a jest klasy B - rzutowanie możliwe");
    B b2 = (B)a; // rzutowanie jest OK
} else { System.out.println("2: a nie jest klasy B - rzutowanie niemożliwe"); }
if (b instanceof A) {  // jeśli b jest klasy A
    System.out.println("3: B jest klasy A - rzutowanie możliwe");
    A a1 = b; // rzutowanie nie potrzebne (ale możliwe)
}else{ System.out.println("3: b nie jest klasy A"); }

// 1: a nie jest klasy B - rzutowanie niemożliwe
// 2: a jest klasy B - rzutowanie możliwe
// 3: B jest klasy A - rzutowanie możliwe
```

Object

uniwersalna nadklasa

Każda klasa w Javie dziedziczy po klasie Object

(nawet jeśli nie jest to jawnie zdefiniowane poprzez extends Object)

- Zmienna typu Object może wskazywać na obiekt dowolnego typu:

```
Object o1 = new Pracownik();
Object o2 = new A();
Object o3 = new String();
```

Przydatne metody klasy Object:

- `Class getClass(); // zwraca klasę obiektu`
- `boolean equals(Object); // sprawdza czy obiekty są takie same`
- `String toString(); // zwraca obiekt reprezentację obiektu w postaci Stringa`
- `Object clone(); // tworzy i zwraca klon obiektu`

getClass()

powiedz mi kim jesteś

`Class getClass(); // zwraca klasę obiektu`

`Class` to klasa reprezentująca klasy i interfejsy. `Class` jest klasą finalną - niemożliwe dziedziczenie po tej klasie.

Metody klasy `Class`:

- `String getName(); // zwraca nazwę klasy`
- `Class getSuperclass(); // zwraca klasę bazową`

```
class A{}
class B extends A {}

A obj1 = new A();
Class klasa_obj1 = obj1.getClass();
System.out.println( klasa_obj1.getName() ); // A
Class klasa_nadrzedna_obj1 = obj1.getSuperclass();
System.out.println( klasa_nadrzedna_obj1.getName() ); // java.lang.Object

Object obj2 = new B();
Class klasa_obj2 = obj2.getClass();
System.out.println( klasa_obj2.getName() ); // B
System.out.println( klasa_obj2.getSuperclass().getName() ); // A
```

equals()

czy jesteśmy tacy sami?

```
boolean equals(Object);
// sprawdza czy obiekty są takie same
```

- jeśli nie przeładujemy tej metody, sprawdzi czy obydwa obiekty znajdują się w tym samym miejscu,
- jeśli chcemy porównywać stan obiektów danej klasy konieczne jest przeładowanie metody equals,
- implementując metodę equals we własnej klasie należy sprawdzić:
  1. czy this jest równe obiektowi przekazanemu obiektowi - jeśli tak, zwrócić true,
  2. czy przekazany obiekt jest równy null - jeśli tak, zwrócić false,
  3. czy this i przekazany obiekt są tego samego typu - jeśli nie, zwrócić false,
  - sprawdzić czy wszystkie pola składowe klasy są równe - jeśli tak zwrócić true, jeśli nie false.

equals()

...w praktyce

```
class MojaKlasa
{
    int i;
    String s;
    A a;
    public boolean equals(Object obj)
    {
        if (this == obj) return true; // 1. czy wskazują w to samo miejsce
        if (obj == null) return false; // 2. czy obj nie wskazuje na nic
        if (getClass() != obj.getClass() ) return false;
        // 3. czy są innych typów
        // jeśli są tych samych typów, rzutujemy obj do typu MojaKlasa
        MojaKlasa obj_mk = (MojaKlasa)obj;
        // 4. sprawdzamy po kolei wszystkie pola składowe,
        //    tam gdzie to możliwe za pomocą metod equals
        if ( i == obj_mk.i // jeśli składowe i są takie same
            && s.equals(obj_mk.s) // oraz składowe (Stringi) s są takie same
            && a.equals(obj_mk.a) ) // oraz składowe (klasy A) s są takie same
                return true; //zwracamy prawdę
        return false; //w przeciwnym wypadku fałsz
    }
}
```

toString()

opisz mi kim jesteś

```
String toString(); // zwraca obiekt reprezentację obiektu w
postaci Stringa
```

- jeśli nie przeładujemy metody zwróci:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

- zaleca się aby każda klasa definiowała swoją metodę toString,

- metoda toString jest automatycznie wywoływana podczas wypisywania obiektów.

toString()

opisz mi kim jesteś

- bez przeładowanej metody toString():

```
class Osoba{
    String imie, nazwisko;
    Osoba (String i, String n) {
        imie = i; nazwisko = n;
    }
}
Osoba jk = new Osoba("Jas", "Kowalski");
String s = jk.toString();
System.out.println(s); // Osoba@5ed70d7a
System.out.println(jk); // Osoba@5ed70d7a
```

toString()

opisz mi kim jesteś

- z przeładowaną metodą toString():

```
class Osoba{
    String imie, nazwisko;
    Osoba (String i, String n) {
          imie = i; nazwisko = n;
    }
    public String toString()
    {
          return getClass().getName() + ": " + imię + ' ' + nazwisko;
    }
}

Osoba jk = new Osoba("Jas", "Kowalski");
String s = jk.toString();
System.out.println(s); // Osoba: Jas Kowalski
System.out.println(jk); // Osoba: Jas Kowalski
```

clone()

sklonuj się!

```
Object clone(); // tworzy i zwraca klon obiektu
```

- clone to chroniona metoda klasy Object,
- przydziela pamięć dla nowego obiektu i kopiuje w to miejsce zawartość pamięci obiektu klonowanego,
- zazwyczaj, jeśli chcemy umożliwić klonowanie obiektów własnych klas należy:
  1. zaimplementować interfejs Cloneable,
  2. przedefiniować metodę clone z modyfikatorem dostępu public,
  3. wewnątrz metody clone wywołać super.clone() oraz metody clone dla zmienialnych podobiektów,
  - przechwycić wyjątek CloneNotSupportedException.

clone()

jak się sklonować - w praktyce

```
class A implements Cloneable // implementacja interfejsu Cloneable
{
    int i = 10;
    String s ="A";
    B b = new B();
    public Object clone(){
        try {
            A a = (A) super.clone();
            // wywołanie clone z klasy nadrzędnej
            a.b = (B) b.clone(); // sklonowanie zmienialnej składowej
            return a; // zwrócenie obiektu
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

static

jeden dla wszystkich

Statyczne pole składowe

- istnieje tylko jedna taka zmienna - wspólna dla wszystkich obiektów klasy,
- zmienna statyczna należy do klasy, a nie do obiektu.

```
class Pracownik{
      static int nastepneID = 1;
      int id;
      Pracownik() { id = nastepneID++; }
}
```

Metody statyczne klasy to metody, które nie operują na obiektach tej klasy. Ich wywołanie jest niezależne od istnienia obiektu klasy.

```
class Kalkulator{
      static int dodaj(int a, int b){ return a+b; }
}
int wynik = Kalkulator.dodaj(1,2);
```

abstract

niby jest, a jednak nie ma...

Klasa abstrakcyjna

- nie można utworzyć obiektu tej klasy,
- może posiadać metody zadeklarowane, ale nie zdefiniowane (metody te muszą zostać zdefiniowane we wszystkich nieabstrakcyjnych klasach dziedziczących po klasie abstrakcyjnej)

Metoda abstrakcyjna to metoda nie posiadająca definicji (zdefiniowana, ale nie zaimplementowana w danej klasie)

```
abstract class Osoba{
      abstract String kimJestes();
}

class Pracownik extends Osoba{
      String kimJestes(){ return "Jestem pracownikiem"; }
}
```

Klasa posiadająca chociaż jedną metodę abstrakcyjną musi być abstrakcyjna.

Klasa abstrakcyjna nie musi posiadać metody abstrakcyjnej.

`final` po raz pierwszy

stałe, niezmienne, ale inne dla każdego

Finalne pola składowe

- wartość pola finalnego musi być zdefiniowana w kodzie każdego konstruktora lub w jednym z bloków inicjujących (bezpośrednia inicjacja pola),
- nie można modyfikować wartości pól finalnych poza konstruktorem.

```
class Osoba{
      final String pesel;
      Osoba(String p) { pesel = p; } // wewnątrz konstruktora OK
      void zmienPesel { pesel = "ABC"; } // poza konstruktorem
BŁAD!
}
```

Klasa, której wszystkie pola składowe są finalne nazywana jest klasą niezmienną. Po zakończeniu działania konstruktora stan klasy jest niezmienny.

Przykładem klasy niezmiennej jest klasa String

`final` po raz drugi

nie próbuj zmieniać tego kim jestem i jak się zachowuję!

Klasa finalna to klasa, której nie można rozszerzać (niemożliwe jest dziedziczenie po tej klasie).

```
class Dyrektor {/*...*/}
final class Prezes extends Dyrektor {/*...*/}  // dziedziczenie po klasie niefinalnej OK
class SuperPrezes extends Prezes {/*...*/} // dziedziczenie po klasie finalnej Prezes BŁĄD
```

Metoda finalna to metoda, która nie może zostać przeładowana (zmodyfikowana) w klasie pochodnej.

```
class Osoba{
	String imie, nazwisko;
	final String przedstawSie() { return imie+nazwisko; }
	String przywitajSie() { return "Witam"; }
}
class Pracownik extends Osoba{
	String przedstawSie() { return "Jestem "+imie+nazwisko; }
// metoda finalna BŁĄD!
	String przywitajSie() { return "Dzien dobry"; }
// metoda nie jest finalna OK
}
```

Klasa zawierająca metody finalne nie musi być klasą finalną.

Zdefiniowanie klasy jako finalna powoduje, że wszystkie metody stają się finalne (wzrost efektywności !).
Dotyczy to tylko metod - pola składowe klasy finalnej mogą, ale nie muszą być finalne.

`static final`

wspólne dla wszystkich i niezmienne aż do końca programu

Połączenie modyfikatorów `static` i `final` powoduje utworzenie **stałego pola statycznego**.

Wartość tego pola jest stała, wspólna dla wszystkich obiektów klasy i musi być określona bezpośrednio w definicji klasy.

```
class A{ int i; }
class Kalkulator{
      public static final double PI = 3.14;
      public static final double E =  2.71;
      public static final A a = new A();
}
Kalkulator.a = new A(); // BŁAD próba modyfikacji stałej referencji
Kalkulator.a.i = 10; // OK modyfikacja pola obiektu wskazywanego przez stałą referencję
```

Nie powinno się definiować pól składowych publicznych, można jednak bez obaw definiować publiczne stałe pola statyczne.

Polimorfizm

```
class A{
      void przedstawSie(){ System.out.println("Jestem A"); };
}
class B extends A{
      void przedstawSie(){ System.out.println("Jestem B"); };
}
class C extends A{
      void przedstawSie(){ System.out.println("Jestem C"); };
}
A a = new A();
A ac = new C();
A ab = new B();
a.przedstawSie(); // Jestem A
ab.przedstawSie(); // Jestem B
ac.przedstawSie(); // Jestem C
```

Sytuacje wyjątkowe

Wyjątek (ang. exception) to niestandardowa sytuacja, która przerywa normalne wykonanie programu.

Obsługa sytuacji wyjątkowych:

- po wystąpieniu sytuacji „nienormalnej” może zostać zgłoszony wyjątek (odpowiedniego typu); wyjątek musi być obiektem klasy dziedziczącej po klasie Throwable (nie koniecznie bezpośrednio po tej klasie),
- referencja do obiektu wyjątku jest „wyrzucana” poza bieżący kontekst wykonywania programu,
- wyjątek może zostać przechwycony i obsłużony - po obsłużeniu wyjątku działanie programu (metody) może być kontynuowane,
- nie przechwycony wyjątek jest wyrzucany o jeden poziom wyżej.

Przechwytywanie wyjątków

co zrobić gdy coś pójdzie nie tak

Do przechwytywania wyjątków wykorzystywany jest blok `try-catch`

```
try{
        /* blok instrukcji, w którym spodziewamy się
           wystąpienia sytuacji wyjątkowej */
} catch ( Typ_wyjątku1 ex ) {
        /* blok instrukcji wykonywany
           w wypadku wystąpienia wyjątku typu 1 */
} catch (Typ_wyjątku2 ex){
        /* blok instrukcji wykonywany
           w wypadku wystąpienia wyjątku typu 2 */
} finally {
        /* blok instrukcji wykonywany bez względu na
           to czy wystąpił wyjątek  */
}
```

Kolejność przechwytywania

kto pierwszy ten lepszy

```
class Kalkulator{
    static double podziel(double a, double b)
        throws ArithmeticException {
        return a / b;
    }
}

try {
    Kalkulator.podziel(1,0);
} catch (Exception ex){
    /* obsługa wyjątków różnych typów    */
    System.out.println(ex.toString());
}
```

Kolejność przechwytywania

kto pierwszy ten lepszy

```
class Kalkulator{
    static double podziel(double a, double b)
        throws ArithmeticException {
        return a / b;
    }
}

try {
    Kalkulator.podziel(1,0);
} catch (ArithmeticException ex){
    /* obsługa wyjątków arytmetycznych   */
    System.out.println(ex.toString());
} catch (Exception ex){
    /* obsługa wyjątków różnych typów    */
    System.out.println(ex.toString());
}
```

Wyjątki w metodach

złap wszystko to co wyrzucasz
lub powiedz, że czegoś nie złapiesz

```
private boolean czySprzedacAlkohol(int wiek) throws Exception{
     if (wiek<0) throw new Exception("Złe dane");
     return wiek>18;
}
void kupAlkohol(int wiek){
     if ( czySprzedacAlkohol(wiek) ) System.out.println("Sprzedaje
alkohol");
     else System.out.println("Nie sprzedaje alkoholu");
}
```

Błąd - nie przechwycony wyjątek; funkcja nie informuje, że może zgłosić sytuację wyjątkową

Przechwytywanie wyjątków

złap wszystko to co wyrzucasz lub powiedz, że czegoś nie złapiesz

Rozwiązanie 1: poinformować, że może pojawić się sytuacja wyjątkowa

```
void kupAlkohol(int wiek) throws Exception{
      if ( czySprzedac(wiek) )
         System.out.println("Sprzedaje alkohol");
      else System.out.println("Nie sprzedaje alkoholu");
}
```

Rozwiązanie 2: obsłużyć sytuację wyjątkową

```
void kupAlkohol(int wiek) {
      try{
            if ( czySprzedac(wiek) )
               System.out.println("Sprzedaje alkohol");
            else System.out.println("Nie sprzedaje alkoholu");
      } catch (Exception ex){
            System.out.println(ex.toString());
      }
}
```

Pobieranie danych

wyjątkowe dane od użytkownika

Aby odczytać dane od użytkownika należy:

- zaimportować bibliotekę `java.io.*`,
- przechwycić wyjątki typu `IOException`,
- pobrać wartość i rzutować ją do typu char.

```
char c = (char) System.in.read(); // pobranie jednego znaku
import java.io.*;
try {
    char c;
    while( ( c = (char)System.in.read() ) != '\n' ) s+=c;
} catch (IOException ex ) {
    System.out.println("Wyjątek wej/wyj: "+ ex.toString());
}
```

Pobieranie danych

```
Scanner scanner = new Scanner ( System.in );
System.out.print("Podaj swoje imię:");
String imie = scanner.nextLine() ;
System.out.print("Podaj swój wiek:");
int wiek = scanner.nextInt() ;
```

Buforowany
odczyt i zapis danych

- Utworzenie obiektów do odczytu/zapisu plików:

```
FileReader plikWejsciowy = new FileReader(/* nazwa pliku */);
FileWriter plikWyjściowy = new FileWriter(/* nazwa pliku */);
```

- Utworzenie strumieni powiązanych z plikami:

```
BufferedReader in = new BufferedReader(plikWejsciowy);
BufferedWriter out = new BufferedWriter(plikWyjsciowy);
```

- Odczyt i zapis danych:

```
char c = (char)in.read(); // odczyt znaku
String s = in.readLine(); // odczyt linii
out.write( /* char/int/String */ ); // zapis danych na strumień
```

- Zamknięcie buforów:

```
in.close(); out.close();
```

Zapis/odczyt plików

- Otwarcie pliku do odczytu za pomocą klasy Scanner:

```
Scanner wej = new Scanner(new File("plik.txt"));
String linia = wej.nextLine();
```

- Otwarcie pliku do zapisu za pomocą klasy PrintWriter:

```
PrintWriter wyj = new PrintWriter("plik.txt");
wyj.println(120);
```

Konwersja danych
odczyt i zapis danych

- Konwersja danych tekstowych do liczb:

```
String s = "13.2";
try {
   int i = Integer.parseInt(s);
   System.out.println(i);
}catch(NumberFormatException ex){
   System.out.println(ex.toString());
}
try {
   double d = Double.parseDouble(s);
   System.out.println(d);
}catch(NumberFormatException ex){
   System.out.println(ex.toString());
}
```

Interfejsy

- Interfejs to opis co klasa implementująca dany interfejs powinna robić, ale bez określania jak powinna to robić.
- Interfejs może być porównany do klasy abstrakcyjnej, ale w rzeczywistości nie jest klasą, tylko zbiorem wymagań dla klas, które chcą dostosować się do tego interfejsu.
- Klasa implementująca interfejs musi definiować wszystkie metody interfejsu (chyba, że jest klasą abstrakcyjną).
- Interfejsy nie posiadają zmiennych składowych, ale mogą zawierać stałe. Każde pole składowe jest automatycznie traktowane jako public static final.
- Wszystkie metody interfejsu są automatycznie publiczne.

```
public interface MojInterfejs{ /* stałe i deklaracje metod */ }
```

Przykład Własnego interfejsu

PrzedstawiajacySie - klasa, której obiekty potrafią się przedstawić

```
public interface PrzedstawiajacySie {
    String przedstawSie();
}
class A implements PrzedstawiajacySie {
    public String przedstawSie() { return "Jestem klasa A \n"; } }
class B {
    public String przedstawSie() { return "Jestem klasa B \n"; } }
class Test {
    public static void kimJestes( PrzedstawiajacySie obj ){
        System.out.print(obj.przedstawSie());
}
PrzedstawiajacySie pS = new PrzedstawiajacySie();
// BŁĄD! nie można utworzyć obiektu PrzedstawiajacySie
PrzedstawiajacySie pS = new A(); // ale można utworzyć referencje do tego typu
                // i przypisać obiekt klasy implementującej ten interfejs
ArrayList<PrzedstawiajacySie> al = new ArrayList<PrzedstawiajacySie>();
p.add( new A() );
p.add( new B() ); // BŁĄD! C nie implementuje interfejsu PrzedstawiajacySie
for (int i=0; i<p.size(); ++i){
    Test.kimJestes( al.get(i) );
}
```

Przykład Standardowego interfejsu

Comparable - klasa, której obiekty mogą być porównywane

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

class Osoba implements Comparable{
    String imie = "";
    String nazwisko = "";
    int wiek;
    public int compareTo(Object obj) {
        Osoba inna = (Osoba) obj;
        if ( wiek != inna.wiek ) return (wiek>inna.wiek)?1:-1;
        if ( nazwisko.equals(inna.nazwisko) )
            return nazwisko.compareTo(inna.nazwisko);
        if ( imie.equals(inna.imie) ) return imie.compareTo(inna.imie);
        return 0;
    }
}
```

- Interfejs Comparable wykorzystywany jest między innymi do sortowania.

```
Osoba [] tab_os;
/* ... */
Arrays.sort(tab_os);
// możliwe tylko jeśli klasa Osoba implementuje interfejs Comparable
```

Dziedziczenie i implementacja interfejsów

- Klasa może dziedziczyć tylko po jednej klasie, ale może implementować wiele interfejsów.
- Interfejsy mogą dziedziczyć inne interfejsy.

```
interface IA {
    void fun_IA();
}
interface IB extends IA {
    void fun_IB();
}
interface IC {
    void fun_IC();
}
class Klasa implements IB, IC {
    public void fun_IB() { /* ... */ }
    public void fun_IA() { /* ... */ }
    public void fun_IC() { /* ... */ }
}
```

Stałe w Interfejsach

- Interfejsy, poza metodami, mogą posiadać również pola składowe - publiczne, stałe i finalne.
- Jeśli klasa implementuje interfejs pola te są bezpośrednio dostępne wewnątrz tej klasy.

```
interface TypZasobu{
    int inny = 0;
    int ksiazka = 1;
    int gazeta = 2;
}
abstract class Zasob implements TypZasobu{
    int typ;
    public int jakiTyp() { return typ; }
}
class Ksiazka extends Zasob{
    { typ = ksiazka; }
}
Zasob zasob = new Ksiazka();
switch ( zasob.jakiTyp() ){
    case Zasob.ksiazka : System.out.println("Zasob jest ksiazka"); break;
    case Zasob.gazeta  : System.out.println("Zasob jest gazeta"); break;
    default            : System.out.println("Zasob jest czyms innym");
}
```

Sprzęty domowe

jeszcze jeden przykład wykorzystania interfejsów

```
interface Wlaczalny {
      boolean wlacz();
      boolean wylacz();

      boolean OPERACJA_UDANA = true;
      boolean OPERACJA_NIEUDANA = false;
}

interface Stan {
      int WYLACZONE = 0;
      int WLACZONE = 1;
      int STAN_CZUWANIA = 2;
}
```

Lampka

```
class Lampka implements Wlaczalny, Stan {
	int stan;
	public boolean wlacz() {
		stan = WLACZONE;
		return OPERACJA_UDANA;
	}
	public boolean wylacz() {
		stan = WYLACZONE;
		return OPERACJA_UDANA;
	}
}
```

Komputer

```
class Komputer implements Wlaczalny, Stan {
	int stan;
	boolean cosRobi = false;

	public boolean wlacz() {
		stan = WLACZONE;
		return OPERACJA_UDANA;
	}

	public boolean wylacz() {
		if (stan == WYLACZONE || !cosRobi)
		{
			stan = WYLACZONE;
			return OPERACJA_UDANA;
		}
		return OPERACJA_NIEUDANA;
	}
}
```

TELEWIZOR

```
class Telewizor implements Wlaczalny, Stan {
	int stan;
	public boolean wlacz() {
		stan = WLACZONE;
		return OPERACJA_UDANA;
	}
	public boolean wylacz() {
		stan = WYLACZONE;
		return OPERACJA_UDANA;
	}
	public boolean przejdzWStanCzuwania() {
		if (stan == WLACZONE){
			stan = STAN_CZUWANIA;
			return OPERACJA_UDANA;
		}
		return OPERACJA_NIEUDANA;
	}
}
```

LODÓWKA

```
class Lodowka implements Wlaczalny
{
	private void wlaczWtyczkeDoKontaktu() { /* ... */ }
	private void wyciagnijWtyczkeZKontaktu() {   /* ... */ }

	public boolean wlacz()
	{
		wlaczWtyczkeDoKontaktu();
		return OPERACJA_UDANA;
	}

	public boolean wylacz() {
		wyciagnijWtyczkeZKontaktu();
		return OPERACJA_UDANA;
	}
}
```

Gdy jedziemy na wakacje...

```
ArrayList<Wlaczalny> sprzetyDomowe = new ArrayList<Wlaczalny>();
sprzetyDomowe.add(new Lampka());
sprzetyDomowe.add(new Komputer());
sprzetyDomowe.add(new Telewizor());
sprzetyDomowe.add(new Lampka());
sprzetyDomowe.add(new Lodowka());
sprzetyDomowe.add(new Telewizor());


for (int i=0; i< sprzetyDomowe.size(); ++i )
{
  if (sprzetyDomowe.get(i).wylacz()==Wlaczalny.OPERACJA_NIEUDANA)
    System.out.println("Błąd wyłączania sprzetu nr " + ( i+1 ) );
}
```

NIETYPOWE klasy

wewnętrzne, lokalne i anonimowe

Język Java dostarcza specjalne mechanizmy pozwalające definiować następujące typy klas:

- **klasy wewnętrzne** - definicja klasy wewnętrznej znajduje się wewnątrz definicji innej klasy,
- **lokalne klasy wewnętrzne** - definicja lokalnej klasy wewnętrznej znajduje się wewnątrz metody innej klasy,
- **anonimowe klasy wewnętrzne** - anonimowa klasa wewnętrzna definiowana jest podczas tworzenia obiektu tej klasy; tworzony jest tylko jeden jej obiekt; klasa anonimowa nie ma nazwy.

Klasy wewnętrzne

- Obiekt klasy wewnętrznej w tym przypadku ma wgląd na obiekt klasy zewnętrznej, więc musi być tworzony za jego pośrednictwem.

```
class X {
    int a;
    public class Y {
        int f() {
            return a;
        }
    }
}
//...
X x = new X();
X.Y y = x.new Y();
```

Statyczne klasy wewnętrzne

- W tym przypadku klasa zagnieżdżona nie przechowuje odniesienia do obiektu klasy zewnętrznej, dlatego jest tworzona niezależnie

- Zagnieżdżenie służy jedynie do podkreślenia, że klasa zagnieżdżona jest klasą pomocniczą.

```
class X {
    int a;
    public static class Y {
        int f() {
            return a;
        }
    }
}
//...
X.Y y = new X.Y();
```

Anonimowe klasy wewnętrzne

nie ważne jak się nazywam, ważne co robię

- Składnia anonimowych klas wewnętrznych:

```
new TypNadrzedny ( /* arumenty konstruktora typu nadrzędnego */ ) {
  /* metody i pola składowe klasy anonimowej */
};
```

- Wykorzystanie klas lokalnych i lokalnych anonimowych:

```
public static void main(String[] args) {
  /* klasa lokalna */
  class Osoba {
      String imie; String nazwisko;
      Osoba (String n, String i) { nazwisko = n; imie = i; }
      public String toString() { return imie + " " + nazwisko; }
  };
  //utworzenie obiektu lokalnej klasy anonimowej dziedziczącej po lokalnej
  klasie Osoba
  Osoba pracownik = new Osoba ( "Jas", "Fasolka" ) {
    //wywołanie konstruktora klasy Osoba
    //definicja pól i metod rozszerzających funkcjonalność klasy Osoba
    String stanowisko = "Sprzedawca warzyw";
    public String toString() { return stanowisko + ": " + super.toString(); }
  };
  System.out.print(pracownik); // Sprzedawca warzyw: Fasolka Jas
}
```

Typy wyliczeniowe

- Typy wyliczeniowe są specjalnym typem, który umożliwia przechowywanie w zmiennej predefiniowanego zbioru wartości.

- Są użyteczne w sytuacji, gdy potrzebny jest dla zmiennej ściśle określony zestaw stałych.

- W Javie definicję typu wyliczeniowego poprzedza się słowem kluczowym enum, po którym następuje nazwa tego typu i w wyliczenie kolejnych wartości.

Typy wyliczeniowe

- Na przykład:

```
public enum DzienTygodnia {
   PONIEDZIAŁEK, WTOREK, ŚRODA,
   CZWARTEK, PIĄTEK, SOBOTA,
   NIEDZIELA;
}
```

Typy wyliczeniowe

- Definicje typów wyliczeniowych mogą zawierać metody i pola składowe:

```
public enum Planeta {
    MERKURY (3.303e+23, 2.4397e6),
    WENUS   (4.869e+24, 6.0518e6),
    ZIEMIA   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6);
```

```
private final double masa;
private final double promien;
```

Typy wyliczeniowe

```
 Planeta(double m, double r)
   {
    masa = m;
    promien = r;
   }
     private double masa() { return masa; }
     private double promien() { return promien; }
}
```

Typy wyliczeniowe

- Wartości typu wyliczeniowego tworzy się przypisując stałą do zmiennej, na przykłade:

```
DzienTygodnia dzien = DzienTygodnia.PONIEDZIAŁEK;
Planeta planeta;
planet a = Planeta.ZIEMIA ;
```

Typy wyliczeniowe

- Kompilator dodaje kilka metod do typu wyliczeniowego (np. values()).
- Wynika to z tego, że typy wyliczeniowe dziedziczą niejawnie po java.lang.Enum.
- Wyliczenie nie może dziedziczyć po klasie.
- Nie można definiować metod i pól składowych przed wyliczeniem stałych.

ArrayList

wygodniejsze tablice

- ArrayList to tablica, której wielkości nie musimy określać podczas tworzenia. Możliwe jest dodawanie i usuwanie elementów z tej tablicy.

```
ArrayList<typ_elementow> nazwa_zmiennej = new
ArrayList<typ_elementow>();

import java.util.ArrayList;

ArrayList<Integer> al = new ArrayList<Integer>();
for (int i =0; i< 10; ++i) al.add(i); // dodanie elementu
```

ArrayList

```
for (int i =0; i< al.size(); ++i) System.out.print(al.get(i) + "
");
// 0 1 2 3 4 5 6 7 8 9
// get(i) - odczyt elementu o indeksie i
al.remove(3); // usuniecie elementu o indeksie 3
for (int i =0; i< al.size(); ++i) System.out.print(al.get(i) + "
");
// 0 1 2 4 5 6 7 8 9
al.add(4, 2); // dodanie elementu 2 na pozycji o indeksie 4
for (int i =0; i< al.size(); ++i) System.out.print(al.get(i) + "
");
// 0 1 2 4 2 5 6 7 8 9
al.remove(new Integer(2)); /
/ usunięcie pierwszego elementu równego przekazanemu parametrowi
for (int i =0; i< al.size(); ++i) System.out.print(al.get(i) + "
");
// 0 1 4 2 5 6 7 8 9
```