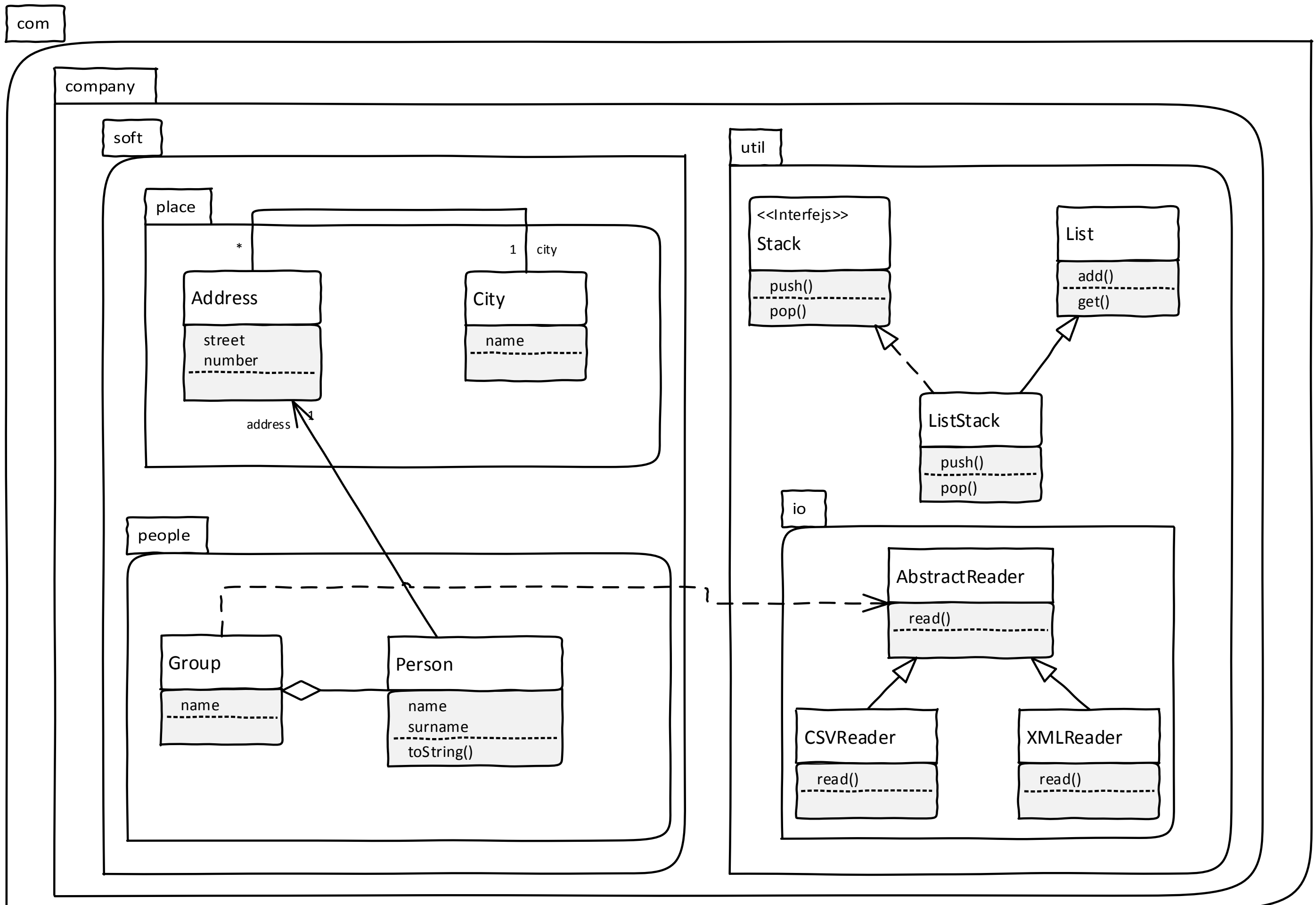
Organizacja kodu

Kod w języku Java zorganizowany jest hierarchicznie w pakiety. Pakiety mogą zawierać deklaracje klas, interfejsów oraz pakiety niższego poziomu.



- Pakiet `com` zawiera pakiet `company`, ten z kolei `soft`, itd.
- W pakiecie `com.company.soft.place` umieszczone są klasy `Address` i `City`
- W pakiecie `com.company.soft.people` znajdują się klasy `Group` i `Person`
- W pakiecie `com.company.util` znajdują się: interfejs `Stack`, klasy `List` i `ListStack` oraz pakiet io.
- Klasy i interfejsy mogą być połączone relacjami – i relacje te mogą przecinać granice pakietów.

Zasięg nazwy klasy (interfejsu)

Zadeklarowana klasa (interfejs) jest widoczna w pakiecie, do którego należy pod nazwę *Identifier*.

Jeżeli klasa należy do pakietu *P*, wówczas w pełni kwalifikowaną nazwą, jednoznacznie identyfikującą klasę, jest *P.Identifier*. Jeżeli klasa należy do pakietu bez określonej nazwy, wówczas pełną nazwą klasy jest *Identifier*.

Przykład

```
package com.company.soft.people;
class Person
{
  String name;
  String surname;
}
```

W pełni kwalifikowaną nazwą jest:
`com.company.soft.people.Person`.

```
class BezpańskiKot
{
  String name;
}
```

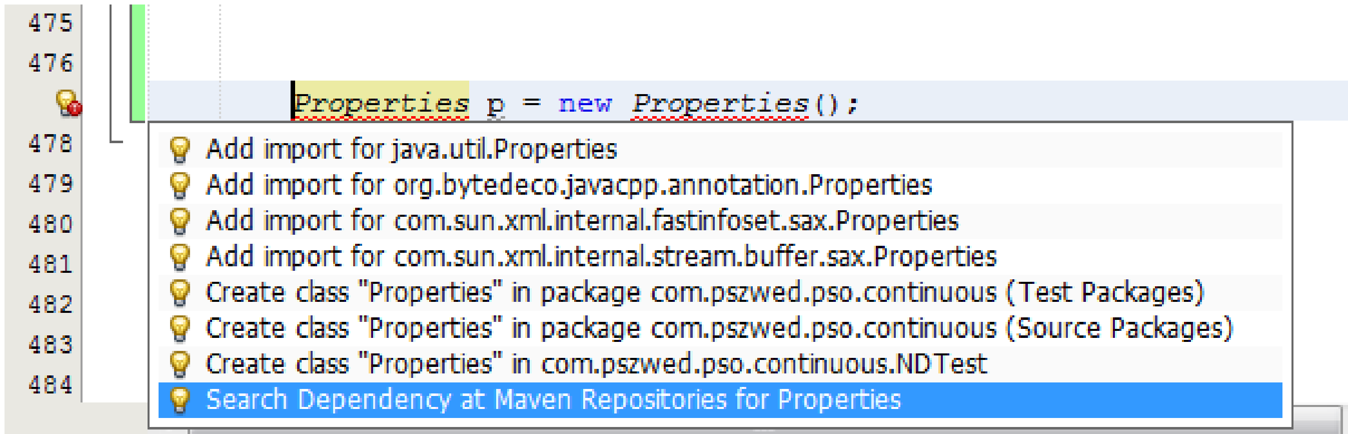
W pełni kwalifikowaną nazwą jest `BezpańskiKot`.

Chcąc odwołać się do klasy lub interfejsu spoza pakietu należy:

- Użyć w pełni kwalifikowanej nazwy:
  `com.company.util.io.CSVReader r = new com.company.util.io.CSVReader();`
- Zaimportować klasę (interfejs):
  `import com.company.util.io.CSVReader;`

```
//...
CSVReader r = new CSVReader();
```

IDE są w stanie automatycznie dodać deklaracje importu do klas, których kod jest widoczny poprzez zmienną CLASSPATH. Czasami jednak kilka definicji jest dostępnych



Wewnątrz pakietu nie mogą występować klasy lub interfejsy o tych samych nazwach.

Nazwa deklarowanej klasy nie może pokrywać się z nazwą pojedynczo zaimportowanej klasy z innego pakietu:

```
import from.somewhere.A;
class A {…}
```

Klasy

Deklaracja klasy jest równocześnie deklaracją nowego typu referencyjnego, a także opisem w jaki sposób implementowane są obiekty danej klasy.

Podstawowe cechy

- Klasa może być zadeklarowana jako *publiczna* (`public`), wówczas może być dostępna z innych pakietów
- Klasy mogą być zadeklarowane jako *abstrakcyjne* (`abstract`) jeśli deklarują lub dziedziczą niezaimplentowane metody abstrakcyjne.
- Klasy *finalne* (`final`) nie mogą mieć klas potomnych
- Wszystkie klasy poza klasą `Object` są klasami potomnymi jakiejś innej klasy. Klasy mogą implementować jeden lub kilka interfejsów.
- W ciele klasy mogą znajdować się:
    - deklaracje pól i metod składowych obiektów,
    - konstruktory,
    - deklaracje pól i metod statycznych,
    - deklaracje klas wewnętrznych
- Elementom deklarowanym wewnątrz klasy można przypisywać standardowe modyfikatory określające prawa dostępu (`public`, `protected`, `private`)

Składnia deklaracja klas

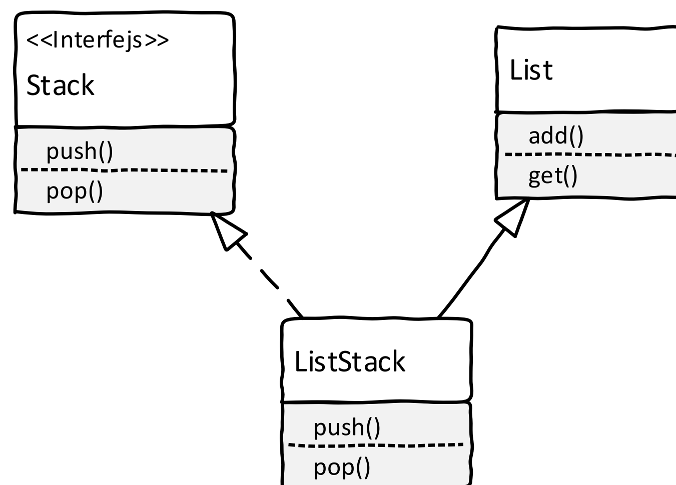
Deklaracja klasy ma następującą postać:

ClassModifiers$_{opt}$ `class` *Identifier* *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

gdzie:

- *ClassModifiers*$_{opt}$ to modyfikatory (`public abstract final`)
- *Identifier* – nazwa klasy
- *Super*$_{opt}$ – specyfikacja klasy bazowej (postaci `extends` *ClassType*). Jej brak oznacza, że dziedziczymy bezpośrednio po klasie `Object`.
- *Interfaces*$_{opt}$ – specyfikacja interfejsów realizowanych przez klasę (postaci `implements` *IntefaceName* [, *IntefaceList*])
- *ClassBody* – deklaracja składowych umieszczona w bloku `{ }`.

Przykład




```
class List
{
  public void add(Object o){. . .}
  public Object get(int n){. . .}
}
```

```
interface Stack
{
  void push(Object o);
  Object pop();
}


class ListStack extends List implements Stack
{
  public void push(Object o){. . .}
  public Object pop(){. . .}
}
```

Interfejs `Stack` deklaruje metody `push(Object o)` i `pop()`, ale ich nie implementuje.

Klasa `ListStack` **dziedziczy** metody klasy `List` oraz implementuje metody interfejsu `Stack`.

Nadklasy i podklasy

Terminem nadklasa (ang. *superclass*) określana jest klasa, po której dana klasa dziedziczy (klasa bazowa). Terminem podklasa (ang. *subclass*) określana jest klasa potomna.

Deklaracja

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

określa relację dziedziczenia:

- klasa `A` jest bezpośrednią nadklasą `B`
- klasa `B` jest bezpośrednią podklasą `A`

Jeżeli deklarujemy klasę bez podania jej nadklasy (bez frazy `extends` *Super*), wówczas nadklasą jest `java.lang.Object`.

Podobna terminologia stosowana jest dla interfejsów.

Przykład

```
interface IF
{
  abstract void f();
}

interface IG extends IF
{
  abstract void g();
}

class A implements IG
{
  public void f(){}
  public void g(){}
}
```

- Interfejs `IF` jest nadinterfejsem (ang. *superinterface*) dla `IG`
- Interfejs `IG` jest podinterfejsem (ang. *subinterface*) dla `IF`
- Interfejs `IG` jest bezpośrednim nadinterfejsem dla klasy `A`.
- Klasa `A` implementuje interfejsy `IG` (oraz pośrednio `IF`)

Modyfikatory

Modyfikatory stosowane przy deklaracji klas to:

`public abstract final`

Klasy publiczne

Modyfikator `public` oznacza, że dostęp do danej klasy może zostać zrealizowany spoza jej pakietu. W jednostce kompilacji może się znaleźć tylko jedna klasa zadeklarowana jako `public`. Kompilator oczekuje, że nazwa klasy będzie się pokrywała z nazwą pliku.

Klasy abstrakcyjne

Klasy abstrakcyjne, to klasy, których definicja nie pozwala na utworzenie konkretnego obiektu za pomocą operatora `new`.

Klasa jest klasą abstrakcyjną, jeżeli:

- jawnie deklaruje metodę abstrakcyjną
- dziedziczy metodę abstrakcyjną po klasie nadrzędnej, ale jej nie implementuje
- jest zadeklarowana jako klasa realizujący pewien interfejs, ale nie deklaruje ani nie dziedziczy zdefiniowanych w nim metod

Deklaracja klasy abstrakcyjnej wymaga użycia modyfikatora `abstract`.

Jeżeli nie chcemy pozwolić na utworzenie obiektu danej klasy, wówczas możemy zadeklarować ją jako abstrakcyjną, nawet jeżeli nie dziedziczy lub deklaruje metod abstrakcyjnych. Zazwyczaj taka klasa będzie deklarowała szereg metod statycznych, które mogą być wywoływane bez tworzenia obiektu klasy.

```
abstract class C
{
  static void foo(){...}
}
```

Alternatywnym (zalecanym) rozwiązaniem, które zabezpiecza przed możliwością stworzenia obiektu danej klasy (instancjacją) jest zdefiniowanie prywatnego bezparametrowego konstruktora:

```
class C
{
  private C(){}
  static void foo(){...}
}
```

W obu przypadkach można wywołać metodę statyczną jako `C.foo()`, natomiast nie można skompilować kodu tworzącego obiekt: `new C()`.

Klasy finalne

Klasy finalne deklaruje się z użyciem modyfikatora `final`. Są to klasy, dla których nie przewiduje się dalszego rozszerzania funkcjonalności przez dziedziczenie.

Zastosowaniem klas finalnych jest między innymi zabezpieczenie przed naruszeniem bezpieczeństwa w wyniku przedefiniowania zaimplementowanych w nich metod.

Klasy finalne nie mogą być równocześnie klasami abstrakcyjnymi, stąd ewentualne zabezpieczenie przed instancjacją klasy powinno być zrealizowane w postaci prywatnego bezparametrowego konstruktora.

Składowe klas

Ciało klasy może zawierać deklaracje składowych klasy (pól i metod) deklaracje pól i metod statycznych oraz konstruktorów.

Składowymi klasy są:

- pola i metody zadeklarowane wewnątrz ciała klasy
- składowe odziedziczone po bezpośredniej nadklasie
- składowe odziedziczone po bezpośrednim nadinterfejsie

Konstruktory oraz pola i metody statyczne nie są traktowane jak składowe klasy i nie podlegają dziedziczeniu.

Możliwe jest odziedziczenie wyłącznie składowych zadeklarowanych jako `public` lub `protected`.

Składowe typu `private` nie są dziedziczone.

Nie są także dziedziczone składowe bez określonego dostępu (czyli z dostępem typu *package*) pomiędzy klasami należącymi do różnych pakietów.

Pola składowe

Ogólna postać deklaracji pól (atrybutów) klas jest następująca:

$Modifier_{opt}$ *Type Identifier* $Initializer_{opt}$ [, *Identifier* $Initializer_{opt}$];

Gdzie:

$Modifier_{op}$ – określa między innymi sposób dostępu (`public`, `protected`, `private`)

Type – jest nazwą typu (wbudowanego, referencyjnego, tablicowego)

Identifier – jest nazwą pola

$Initializer_{opt}$ – pozwala na nadanie polu wartości początkowej

- W jednej deklaracji można zdefiniować większą liczbę pól.
- Nazwy pól i metod mogą się pokrywać; klasa nie może deklarować dwóch pól o tej samej nazwie.

- Zakresem widzialności nazwy pola jest ciało klasy.

Przykład

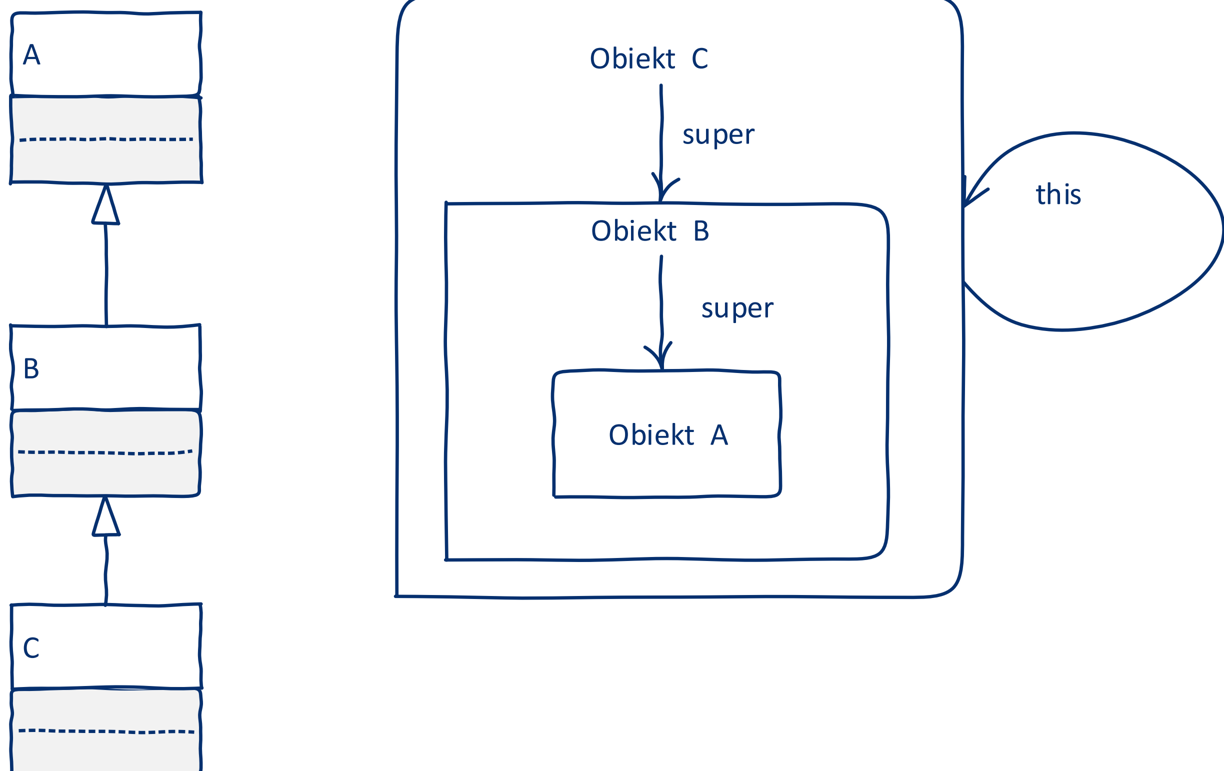
```
class A
{
  private String name=”A”;
  int x,y=1;
  double z=3.5;
  int[] tab=new int[]{1,2,3};
  String s= "Ala ma kota";
}

class B extends A
{
  int z =2;
  int s;
}

class Test
{
  public static void main(String[] args) {
    B b = new B();
    System.out.println(b.x+" "+b.y+" "+b.z);
    System.out.println(b.s);
    System.out.println(((A)b).s);
  }
}
```

Wynik:

```
0 1 2
0
Ala ma kota
```

W powyższym przykładzie:

- Klasa `A` deklaruje pole `name`. Nie jest ono dostępne z zewnątrz.
- Klasa `A` deklaruje pola `x, y`. Pole `x` jest domyślnie inicjowane wartością 0, pole y jest jawnie inicjowane wartością 1.
- Pole `tab` klasy `A` jest inicjowane referencją do obiektu będącego tablicą. Przy tworzeniu instancji obiektu klasy `A` zawsze tworzona jest nowa tablica. Kod:
  ```
  System.out.println(
  new A().tab == new A().tab);
  ```
  wypisze `false`.
- Deklaracje pól `z` i `s` w klasie `B` **przesłaniają** odziedziczone pola. Z zewnątrz dostęp do pól nadklasy jest możliwy poprzez rzutowanie: `((A)b).s`. Wewnątrz metod klasy B można posłużyć się składnią `super.s`.
- W metodach klasy widoczne są dwie referencje:
  - `this` – wskazuje obiekt klasy
  - `super` – wskazuje podobiekt klasy nadrzędnej



Modyfikatory

Modyfikatorami pól mogą być:

- standardowe modyfikatory dostępu (`public`, `protected`, `private`)
- modyfikatory określające dodatkowe własności pola: `final`, `static`, `transient` i `volatile`.

Modyfikatory dostępu

W języku Java można zdefiniować cztery typy dostępu do pól lub metod klasy:

- `public` – dostęp do składowych publicznych możliwy jest z dowolnej klasy
- `protected` – dostęp do składowych chronionych możliwy jest z metod klas pochodnych (podklas) danej klasy oraz z dowolnej klasy należącej do tego samego pakietu.
- `private` – dostęp jest możliwy wyłącznie z metod danej klasy. Pola i metody prywatne nie są dziedziczone.
- `package` – jest to domyślny rodzaj dostępu, który występuje, jeżeli deklaracji pola lub metody nie poprzedzimy żadnym z powyższych. Oznacza, że do deklarowanej składowej będą miały wszystkie klasy należące do tego samego pakietu.

  Wyjątkiem od tej reguły są deklaracje metod wewnątrz interfejsów. Nawet, jeśli nie jest to wyspecyfikowane, są one **zawsze publiczne**.

  Składowe z dostępem domyślnym (`package`) są więc dziedziczone przez podklasy należące do tego samego pakietu, co nadklasa; nie są natomiast dziedziczone przez podklasy należące do innego pakietu.

Możliwości dostępu do składowej w zależności od jej modyfikatora podsumowuje poniższa tabela.

Modyfikator Dostępu	Klasa	Podklasy	Klasa pakietu	Inne klasy
`Public`	✓	✓	✓	✓
`Protected`	✓	✓	✓	
`Private`	✓			
`Package`	✓		✓	

Pola statyczne

Modyfikator `static` definiuje pole, które nie jest składową obiektu, ale składową klasy. Niezależnie od liczby obiektów danej klasy, istnieje dokładnie jedno wcielenie zmiennej statycznej.

Pola statyczne nazywane są zmiennymi klas, w odróżnieniu od zwykłych pól, które są zmiennymi obiektów.

- Do pól statycznych możemy bezpośrednio odwoływać się z metod statycznych i niestatycznych obiektu.
- Z zewnątrz można odwoływać się do pól statycznych za pośrednictwem referencji oraz za pośrednictwem wyrażeń postaci: *NazwaKlasy.nazwaPola*.

Przykład

```
class Counter
{
  static int value;
  void dump()
  {
    System.out.println(value);
  }
  static void inc(){value++;}
}

class Test {
  public static void main(String[] args) {
    Counter.value++;
    System.out.println(Counter.value);
    new Counter().value++;
    Counter.inc();
  }
}
```

Pola finalne

Pola zadeklarowane jako `final` muszą być zainicjowane w momencie deklaracji. Polami finalnymi mogą być zarówno zmienne klas, jak i obiektów.

Pole finalne nie może stać po lewej stronie operatora przypisania, stąd raz zainicjowane pole finalne nigdy nie zmienia wartości.

Pole finalne może być referencją do obiektu (lub tablicy). Zawartość wskazywanego obiektu może być modyfikowana, ale pole zawsze wskazuje ten sam obiekt.

Kombinacja static final = stała

Pola finalne są często wykorzystywane do definiowania stałych, np.:

```
public class Math{
    public static final double PI = 3.14;
}
```

Skąd taki wybór:

- Pole statyczne istnieje niezależnie od obiektów klasy i możliwy jest dostęp poprzez podanie nazwy klasy: `Math.PI`
- Po inicjalizacji nie można zmienić wartości pola finalnego.

Pola typu transient

Pola typu `transient` są polami, które są ignorowane przez standardowe mechanizmy zapisu stanu obiektów.

Typowym przykładem mogą być atrybuty obiektu, które mogą zostać obliczone na podstawie wartości innych atrybutów. Często wprowadza się je dla przyspieszenia działania programu, ale nie ma potrzeby ich zapisywać.

Pola typu volatile

W środowisku wielowątkowym dostęp do zmiennych dzielonych przez kilka wątków może zostać zoptymalizowany. Zamiast odczytywać wartość zmiennej za każdym razem wątek może przechowywać we własnej pamięci kopię dzielonej zmiennej i synchronizować oryginalną zawartość zmiennej oraz kopii w wybranych miejscach wykonania programu (podczas uzyskiwania dostępu i zwalniania monitora).

Zgodnie z powyższą regułą, dostęp do dzielonej zmiennej powinien być realizowany przez metodę synchroniczną (`synchronized`) lub w synchronicznym bloku instrukcji.

Alternatywnym rozwiązaniem jest zadeklarowanie zmiennej jako `volatile`. Wówczas przy każdym dostępie do zmiennej będzie używana rzeczywista wartość, a nie wartość kopii.

Inicjalizacja pól

Deklaracja pola może być połączona z inicjalizacją, która jest traktowana analogicznie jak przypisanie. W przypadku braku inicjalizacji pola otrzymają standardowe zerowe wartości (`null` dla referencji).

Po prawej stronie znaku przypisania może wystąpić dowolne wyrażenie (w tym stała, referencja do obiektu stworzonego w wywołaniu operatora `new` lub wywołanie funkcji

Inicjalizacja pól odbywa się w kolejności określonej przez ich deklarację, nie jest możliwe odwołanie się do wartości pola zdefiniowanego później.

Inicjalizacja pól niestatycznych

Dla pól niestatycznych (zmiennych obiektu) inicjalizacja zachodzi przy tworzeniu obiektu. Jeżeli klasa deklaruje konstruktor, to jest on wołany po inicjalizacji.

Dodatkowo, inicjalizacja może być umieszczona wewnątrz bloku instrukcji umieszczonego bezpośrednio w ciele klasy.

Przykład:

```
class InstanceVars
{
  int x = 7;

  InstanceVars(){
    System.out.print (x+" ");
    x = 4;
  }

  {
    System.out.print(x+" ");
    x = 45;
  }
  public static void main(String[] args) {
    System.out.println(new InstanceVars().x);
  }
}
```

Wypisze: `7 45 4`

Inicjalizacja pól statycznych

Inicjalizacja pól statycznych odbywa się „na żądanie”:

- kiedy jest tworzony obiekt danej klasy
- kiedy uzyskujemy dostęp do statycznego atrybutu
- kiedy wywołujemy statyczną metodę

Przykład:

```
class Attribute
{
  Attribute()
  {
    System.out.println("Attribute constructor
    called");
  }
}

class StaticMembers
{
 static Attribute a = new Attribute();
 static int x= 3;
 static void dump(){
   System.out.println("static dump() called");
 }
}

public class Test {
  public static void main(String[] args) {
    //new StaticMembers();
    //System.out.println(StaticMembers.x);
    //StaticMembers.dump();
  }
}
```

Inicjalizacja pól statycznych zachodzi przy wykonaniu każdej z instrukcji. Za każdym razem wpierw wypisywany jest tekst „`Attribute constructor called`".

Analogicznie, jak dla zmiennych klasy, język Java pozwala na umieszczenie inicjalizacji pól statycznych w blokach instrukcji zadeklarowanych bezpośrednio w ciele klasy, ale poprzedzonych modyfikatorem `static`.

Przykład 1

```
class StaticInitializer
{
  static final double PI; // brak inicjalizacji
  static{
    double sum=0;
    int count = 100000000;
    java.util.Random rand
    = new java.util.Random();
    for(int i=0;i<count;i++){
      double px = rand.nextDouble();
      double py = rand.nextDouble();
      if( px*px + py*py <= 1)sum+=1;
    }
    // jednokrotne przypisanie
    PI = 4*sum/count;
  }
}
```

Powyższy przykład oblicza liczbę π metodą Monte-Carlo. Inicjalizacja zmiennej statycznej PI może trwać kilkadziesiąt sekund. Otrzymana wartość nie jest zbyt dokładna i oczywiście za każdym razem inna.

Przykład 2

```
public class StaticInitializer {

    static Map<String,String> m = new HashMap<>();
    static{
        m.put("kot","cat");
        m.put("pies","dog");
        m.put("koń","horse");
        m.put("krowa","cow");
        m.put("słoń","elephant");
    }

    public static void main(String[] args) {
        System.out.println(m.get("kot"));
    }
}
```

W statycznym bloku mapa (słownik) m została zainicjowana wartościami. Nie ma potrzeby pisania (i wywoływania) dedykowanej funkcji.

Metody

Metoda deklaruje kod wykonywalny, który może zostać uruchomiony po przesłaniu ustalonej liczby parametrów.

Składnia

MethodModifiers$_{opt}$ *ResultType Identifier (FormalParameterList*$_{opt}$*)* *Throws*$_{opt}$ *MethodBody*

Gdzie:

- *MethodModifiers*$_{opt}$ – modyfikatory
- *ResultType* –zwracany typ lub `void`
- *Identifier*– nazwa metody
- *FormalParameterList*$_{opt}$ – lista parametrów oddzielonych przecinkami
- *Throws*$_{opt}$ – lista wyrzucanych wyjątków postaci `throws` *ClassList*
- *MethodBody* – ciało metody. Ma postać instrukcji blokowej `{}` lub średnika `;`

Formalne parametry metod

Formalne parametry podaje się przez określenie listy parametrów oddzielonych przecinkami. Każdy z parametrów ma postać: `ParamType ParamIdentifier`. Lista parametrów może być pusta.

- Zakresem widzialności identyfikatorów formalnych parametrów jest całe ciało metody.

> W języku Java nie jest możliwe przesłanianie, dlatego wewnątrz metod nie można deklarować zmiennych lokalnych o takich samych identyfikatorach, jak formalne parametry.

- Wewnątrz metod obiektu automatycznie uzyskiwany jest dostęp do wszystkich pól obiektu.

> W tym przypadku formalne parametry metod mogą przesłonić deklaracje pól.

Odwołanie się do zawartości pola jest dalej możliwe za pomocą referencji `this` lub przez podanie nazwy klasy i nazwy pola statycznego: `ClassName.fieldName`.

Przykład:

```
class Methods
{
  String x="x";
  static String y="y";
 void foo(int x, int y)
 {
    {
     //double x = 4.3; błąd
    }
    System.out.print(x + " ");
    System.out.print (y + " ");
    System.out.print (this.x + " ");
    System.out.println(Methods.y);
 }
  public static void main(String[] args) {
    new Methods().foo(1,2);
  }
}
```

Rezultat: `1 2 x y`

Sygnatury metod

Sygnaturą metody nazywana jest jej nazwa wraz z określeniem liczby i typów kolejnych formalnych parametrów.

1. Kompilator napotykając wywołanie metody określa liczbę i typy argumentów przekazanych do metody. Na tej podstawie określana jest sygnatura metody, która ma zostać wywołana.
2. Jeżeli wywoływana metoda jest metodą obiektu, wówczas rzeczywista funkcja, która ma zostać wywołana określana jest dynamicznie podczas wykonania programu na podstawie typu obiektu wskazywanego przez referencję.

Klasa nie może deklarować dwóch metod o tych samych sygnaturach, ponieważ kompilator nie mógłby określić, do której funkcji przekazać sterowanie.

Modyfikatory metod

Podobnie jak dla pól, podczas deklaracji metod stosuje się modyfikatory:

- określające sposób dostępu :
  `public`, `protected`, `private`;
- określające własności metod
  `abstract`, `static`, `final`, `synchronized`, `native`.

Metody abstrakcyjne

Deklaracja metody abstrakcyjnej wprowadza nową składową metodę obiektów dla danej klasy, specyfikuje jej sygnaturę, zwracany typ (oraz opcjonalnie listę generowanych wyjątków), ale nie zawiera implementacji. Ciałem metody jest w tym przypadku średnik (;).

Metoda abstrakcyjna może pojawić się wyłącznie w deklaracji klasy abstrakcyjnej lub interfejsu.

Jeżeli metoda abstrakcyjna jest zadeklarowana w abstrakcyjnej klasie A, wówczas każda jej nieabstrakcyjna podklasa musi zapewnić implementację metody.

Przykład

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

abstract class B extends A {
  int i = 5;
}

class C extends B {
  void dump(){System.out.println("C: i="+i);}
}
```

Metodami abstrakcyjnymi nie mogą być:

- metody prywatne (`private`)
- metody statyczne (`static`)
- metody finalne (`final`)

Metody statyczne

Metody statyczne nazywane są często metodami klasy (w odróżnieniu od metod obiektów).

Metody statyczne mogą być wykonywane bez konieczności tworzenia obiektu danej klasy. Wewnątrz metod statycznych nie można uzyskać dostępu do pól i metod niestatycznych, a także nie można posługiwać się referencjami `this` (referencja do bieżącego obiektu) oraz `super` (referencja do obiektu bezpośredniej nadklasy).

Aby wywołać metodę statyczną możemy posłużyć się wyrażeniem postaci:

- `ClassName.staticMethod()`
- lub za pośrednictwem referencji do obiektu, np.: `new ClassName().staticMethod()`

Metody statyczne są często stosowane jako konstrukcja umożliwiająca wprowadzenie do kodu typowych funkcji globalnych, np.: funkcje matematyczne klasy Math: `Math.sin()`, `Math.cos()`, itd.

Dostęp do metod (i pól statycznych) można także uzyskać stosując **statyczny import**:

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
// import static java.lang.Math.*;
public class Main {
    public static void main(String[] args) {
        System.out.printf"sin(%f)=%f\n",PI/3,sin(PI/3));
    }
}
```

Za pomocą metod statycznych można też realizować wzorzec projektowy `Singleton` (i inne podobne). Zadaniem wzorca jest ograniczenie liczby instancji klasy do jednego obiektu.

Przykład

```
final class Singleton{
  private Singleton(){}
  private static Singleton theInstance
  = new Singleton();
  void dump(){System.out.println("Singleton");}
  public static Singleton get()
  {
   return theInstance;
  }
}

class Test
{
public static void main(String[] args) {
    Singleton.get().dump();
  }
}
```

- Pierwsze wywołanie metody statycznej `Singleton.get()` spowoduje incjalizację pola statycznego `Singleton.theInstance`.
- Prywatny konstruktor klasy `Singleton` uniemożliwia jawne stworzenie obiektu, stąd obiekt może zostać stworzony tylko raz.
- Klasa `Singleton` nie może mieć podklas, jako klasa finalna.
- Jedyny dostęp do obiektu klasy `Singleton` możliwy jest więc wyłącznie przez wywołanie metody `Singleton.get()`.

Metody finalne

Metody zadeklarowane jako finalne nie mogą być przedefiniowane w podklasach danej klasy. Wprowadzenie metod finalnych do klasy jest najczęściej decyzją projektową: chcemy pozwolić użytkownikowi kodu na wywoływanie metod, ale nie chcemy, aby przez dziedziczenie mógł np.: naruszyć zasady bezpieczeństwa.

Wszystkie metody zadeklarowane jako `private` są automatycznie metodami finalnymi.

W przypadku metod finalnych kompilator może wygenerować nieco bardziej efektywny kod zastępując ich wywołania kodem *inline*, czyli instrukcjami metod umieszczanymi bezpośrednio w miejscu wywołania.

Metody native

Metody zadeklarowane jako `native` implementowane są w języku zależnym od platformy wykonania, typowo są implementowane w C, C++ lub asemblerze. W przypadku metod `native` nie jest podawana ich implementacja (ciałem metody jest średnik).

Metody synchroniczne

W przypadku, kiedy w programie działa dokładnie jeden wątek, kod metod jest wykonywany w sposób nieprzerwany. W aplikacji wielowątkowej wątek wykonujący kod metody może zostać zawieszony. W miejscu gdzie nastąpiło przerwanie wykonania metody, dane wewnętrzne klasy mogą być niespójne.

Przykład

```
class SampleBuffer
{
  static final int MAX_LENGTH=100;
  private int[] buffer = new int[MAX_LENGTH];
  private int count = 0;
  boolean put(int data){
    if(count>=MAX_LENGTH)return false;
    buffer[count]=data;
    // tu dane są w stanie niespójnym
    count++;
    return true;
  }
}
```

- Gdyby przerwanie wątku wykonującego metodę `put()` nastąpiło w miejscu oznaczonym komentarzem, wówczas następne wywołanie metody `put()` przez inny wątek doprowadziłoby do utraty danych.
- Zadeklarowanie metody jako `synchronized` zapewnia wzajemne wykluczanie podczas realizacji dostępu do tej metody i innych metod synchronicznych obiektu (lub klasy).
- Metodami synchronicznymi mogą być zarówno metody obiektu, jak i metody statyczne klasy.

Dalsze informacje zostaną podane przy omawianiu wątków i monitorów.

Specyfikacja wyrzucanych wyjątków

Deklaracja metod (i konstruktorów) może specyfikować listę wyjątków, które mogą zostać wygenerowane przy wykonywaniu metody.

Specyfikacja wyjątków ma postać listy klas oddzielonych przecinkami pojawiającej się po słowie kluczowym `throws`.

Przykłady:

```
void foo()
      throws Exception1, Exception2
public final void wait()
      throws InterruptedException
public int read()
      throws IOException
```

Klasy pojawiające się na liście wyrzucanych wyjątków muszą być podklasami klasy `Throwable`.

Specyfikacja wyjątków ma na celu przeprowadzenie statycznej analizy ścieżek przetwarzania błędów. Jeśli deklaracja metody określa listę wyrzucanych wyjątków, to muszą być one przechwytywane przy wywołaniu metody lub odsyłane dalej.

Przykład

```
class Exception1 extends Exception { }
class Exception2 extends Exception { }

class ThrowsTest {
  void f() throws Exception1, Exception2  {  }
  void g() throws Exception2   {
    try{
      f();
    }catch (Exception1 e){}
  }
}
```

Dalsze informacje zostaną podane przy omawianiu wyjątków.

Przeciążanie, przedefiniowanie, przesłanianie metod

Przeciążanie metod

Terminem przeciążanie (ang. *overloading*) określane jest równoczesne występowanie metod o tych samych nazwach, ale o innych sygnaturach. Mogą być one odziedziczone lub zadeklarowane w ciele klasy.

Przykład

```
class A
{
  int f(){System.out.prinln("f()");return 0;}
  void f(int a){ System.out.prinln("f("+a+")");}
}
```

Metody przeciążone są zawsze traktowane jak metody niepowiązane. Nie jest wymagana zgodność zwracanych typów lub wyrzucanych wyjątków.

Przedefiniowanie metod

Jeżeli klasa deklaruje metodę obiektu (niestatyczną) o takiej samej sygnaturze jak metoda występująca w nadklasie lub nadinterfejsie, wówczas mówimy, że klasa *przedefiniowuje* jedną lub więcej metod występujących w nadklasach lub nadinterfejsach.

Jeżeli metoda jest deklarowana jako nieabstrakcyjna, a w nadklasach lub nadinterfejsach była ona metodą abstrakcyjną, wówczas mówimy, że klasa *implementuje* metodę o danej sygnaturze.

- Dostęp do przedefiniowanej wersji metody występującej w klasie bazowej możliwy jest za pośrednictwem słowa kluczowego `super`. Jest on możliwy wyłącznie z metod klasy potomnej.
- Przedefiniowane metody muszą zwracać wartości tego samego typu.

Przykład

```
abstract class A {
  void dump(){System.out.println("A");}
  abstract void foo();
}

abstract class B extends A {
  void dump(){
    super.dump();
    System.out.println("B");
  }
}

class C extends B {
  void dump(){
    super.dump();
    System.out.println("C");
  }
  void foo(){System.out.println("foo");}
}
class Test
{
  public static void main(String[] args) {
    C c = new C();
    c.foo();
    c.dump();
    ((A)c).dump();
  }
}
```

Klasa `C` przedefiniowuje metodę `dump()` i przedefiniowuje oraz implementuje metodę `foo()`.

Rzutowanie nie umożliwia wywołania metody zdefiniowanej w klasie bazowej. Wyrażenie `((A)c).dump()` wywoła metodę zdefiniowaną dla obiektu, który jest wskazywany przez referencję.

Proces przekształcania wywołania metody obiektu w rzeczywiste wywołanie kodu jest dwuetapowy.

- Pierwszy etap jest przeprowadzany na etapie kompilacji: określana jest sygnatura metody poprzez znalezienie deklaracji metody o odpowiedniej nazwie oraz liście parametrów. Brak takiej metody sygnalizowany jest jako błąd.
- Drugi etap zachodzi podczas wykonania programu. Na podstawie typu obiektu wskazywanego przez referencję, określany jest rzeczywiste położenie kodu funkcji implementującego metodę i następnie po przekazaniu parametrów następuje jego wywołanie. Ten mechanizm nazywany jest późnym linkowaniem lub dynamicznym linkowaniem (ang. *late binding*, *dynamic binding*, lub *run-time binding*)

Oznacza to, że wszystkie przedefiniowane metody zachowują się polimorficznie: posługując się referencją bazowego typu zawsze wywołamy metodę zaimplementowaną w klasie, której obiekt jest wskazywany przez referencję.

W odróżnieniu od języka C++, gdzie konieczna jest deklaracja metod jako `virtual` (wirtualne), język Java nie wymaga żadnych dodatkowych deklaracji dla stworzenia metod polimorficznych.

Mechanizm późnego linkowania nie dotyczy metod finalnych oraz prywatnych, które domyślnie są metodami finalnymi.

Przesłanianie metod

Jeżeli klasa deklaruje **statyczną** metodę, wówczas ta metoda *przesłania* wszystkie deklaracje statycznych metod, które mogłyby zostać odziedziczone po nadklasach.

Statyczna metoda nie może przesłaniać metod obiektu (niestatycznych).

Przesłonięta metoda może zostać wywołana przez:

- użycie pełnej nazwy postaci `ClassName.methodName()`
- użycie słowa kluczowego `super`
- rzutowanie

Przykład

```
class A {
  static void foo(){
    System.out.println("A.foo");}
}

class B extends A {
  static void foo(){
    System.out.println("B.foo");}
}

class C extends B {
  static void foo(){
    System.out.println("C.foo");
  }
  void g(){
    A.foo();          // A.foo
    super.foo();      // B.foo
    foo();            // C.foo
    ((A)this).foo();  // A.foo
  }
}
```

W przypadku przedefiniowywania i przesłaniania:

- metody muszą zwracać ten sam typ wartości
- wyrzucane wyjątki zadeklarowane na liście `throws` muszą być identyczne
- przedefiniowane lub przesłaniające metody muszą zapewnić co najmniej taki sam poziom dostępu (lub szerszy). Metoda publiczna może być przedefiniowana (przesłonięta) przez metodę publiczną, `protected` przez `public` lub `protected`, metoda z dostępem domyślnym typu *package*, nie może zostać przedefiniowana (przesłonięta) przez metodę prywatną.

Konstruktory

Konstruktor jest specjalną metodą klasy odpowiedzialną za prawidłową inicjalizację tworzonego obiektu.

Deklaracja konstruktora ma następującą postać:

ConstructorModifiers$_{opt}$ ***TypeName* (*FormalParameterList*$_{opt}$)**
Throws$_{opt}$ ***ConstructorBody***

Gdzie:

- *ConstructorModifiers*$_{opt}$ – modyfikatory określające dostęp (`public`, `protected`, `private`)
- *TypeName* – nazwa konstruktora; musi odpowiadać nazwie klasy deklarującej konstruktor
- *FormalParameterList*$_{opt}$ – lista formalnych parametrów
- *Throws*$_{opt}$ – specyfikacja wyrzucanych wyjątków
- *ConstructorBody* – ciało konstruktora; lista instrukcji umieszczonych w bloku `{ }`

Właściwości

- Formalnie, konstruktor nie jest składową klasy.

> Konstruktor nie podlega dziedziczeniu, ani **nie może zostać jawnie wywołany w wyrażeniu posługującym się referencją, tak jak metoda składowa.**

- Konstruktor wołany jest wyłącznie podczas tworzenia obiektu za pomocą operatora `new`.
- Pierwszą instrukcją ciała konstruktora może być wywołanie innego konstruktora danej klasy.

> Można posłużyć się tu wyłącznie wyrażeniem postaci `this(`*ArgumentsList*$_{opt}$`)`. Wywołany zostanie konstruktor o sygnaturze dopasowanej do występującej w wywołaniu listy argumentów.

- Analogicznie, pierwszą instrukcją ciała konstruktora może być wywołanie konstruktora nadklasy.

> W tym przypadku należy użyć wyrażenia postaci `super(`*ArgumentsList*$_{opt}$`)`.

- Podobnie jak inne metody, konstruktory mogą być przeciążane. Kompilator dobiera konstruktor o odpowiedniej sygnaturze na podstawie składni wyrażenia tworzącego obiekt.

Przykład

```
class Point
{
  double x,y;
  Point(double x, double y){
    this.x=x; this.y=y;
  }
  Point(){
    this(0,0);
  }
  void dump(){System.out.print("x="+x+" y="+y);}
}

class Point3D extends Point
{
  double z;
  Point3D(double x, double y, double z){
    super(x,y);
    this.z=z;
  }
  Point3D(){
    this(0,0,0);
  }
  void dump(){super.dump();
        System.out.print(" z="+z);}
}

class Test{
  public static void main(String[] args) {
    Point p1 = new Point(1.0,2.0); p1.dump();
    System.out.println();
    Point3D p2 = new Point3D(1.0,2.0,3.0); p2.dump();
    System.out.println();
    Point p3 = new Point3D(); p3.dump();
  }
}
```

Wynik

```
x=1.0 y=2.0
x=1.0 y=2.0 z=3.0
x=0.0 y=0.0 z=0.0
```

Standardowy konstruktor

W języku Java możliwa jest inicjalizacja pól przy ich deklaracji. Ta własność języka jest szeroko wykorzystywana, dlatego w wielu przypadkach nie jest konieczna implementacja konstruktora.

Zazwyczaj stosuje się konstruktory z parametrami, których zadaniem jest nadanie wymaganych wartości początkowych lub wymuszenie na użytkowniku ustalenia asocjacji pomiędzy obiektami.

Jeżeli klasa nie deklaruje konstruktora, wówczas automatycznie generowany jest bezparametrowy konstruktor, którego pierwszą instrukcją jest wywołanie konstruktora nadklasy. Standardowy konstruktor ma takie same modyfikatory dostępu, jak klasa.

Deklaracja:

```
public class Point {
    int x, y;
}
```

jest więc równoważna deklaracji:

```
public class Point {
    int x, y;
    public Point() { super(); }
}
```

Kolejność operacji przy inicjalizacji obiektu

Przy inicjalizacji obiektu podklasy zakłada się, że składowe obiektu nadklasy oraz atrybuty (pola składowe i pola statyczne) są zainicjowane przed wykonaniem ciała konstruktora.

Standardowa kolejność inicjalizacji obiektu zapewniająca realizację tego założenia jest więc następująca:

- Wołany jest konstruktor nadklasy. Ten krok jest powtarzany rekurencyjnie, aż do osiągnięcia szczytu hierarchii klas.
- Następuje inicjalizacja składowych pól w kolejności ich deklaracji
- Wykonywane jest ciało konstruktora (z pominięciem pierwszej instrukcji, która wykonywana jest w pierwszym kroku)

Rozważmy następujący przykład kodu

```
class A
{
  int attrib;
  A(int a){
    attrib=a;
    System.out.print("A("+attrib+") " );
  }
  A(){
    System.out.print("A("+attrib+") " );
  }
}

class B extends A
{
  B(){
    super(2);
    System.out.print("B ");
  }
}

class C extends A
{
  B b = new B();
  C(){
    System.out.print("C ");
  }
}
```

Przy tworzeniu obiektu klasy `C` za pomocą wyrażenia `new C()` zostanie wypisane „**`A(0)`** `A(2) B` **`C`**”.

- Jeżeli klasa dziedziczy po innej klasie, wówczas w pierwszej instrukcji konstruktora wołany jest zawsze konstruktor bezpośredniej nadklasy (klasy bazowej).

- W momencie tworzenia obiektu klasy `C` wywoływany jest konstruktor `C()`. W jego pierwszej instrukcji w sposób niejawny wołany jest bezargumetowy konstruktor klasy bazowej `A`. To wywołanie jest automatycznie dodawane przez kompilator. Możemy też umieścić je w sposób jawny wołając `super()`.
  To wywołanie spowoduje wypisanie „`A(0)`”.
- Następnie następuje inicjalizacja atrybutów. Tworzony jest obiekt klasy `B`. W jego pierwszej instrukcji wołany jest konstruktor `A(int)`. Przekazanym argumentem jest `2`. Inicjalizacja atrybutu spowoduje wypisanie „`A(2) B`”.
- Na końcu wykonywane jest ciało konstruktora klasy C. Na ekranie zostanie wypisane „`C`”.

Wywołanie metod polimorficznych w konstruktorach

Wywoływanie metod polimorficznych w konstruktorach w języku Java przebiega inaczej niż w C++. Przedefiniowanie metod następuje **przed** wywołaniem konstruktora klasy bazowej, a więc także i przed inicjalizacją atrybutów.

Niebezpieczeństwem jest to, że metoda przedefiniowana w podklasie może odwoływać się do wartości atrybutów podklasy, które nie zostały zainicjowane.

Przykład

```
class A
{
  A(){
    System.out.print("A ");
    f(); // wywoła A.f() lub klasy potomnej!!!
  void f(){
    System.out.print("A.f() ");
  }
}

class B extends A
{
  B(){
    System.out.print("B ");
  }
  void f(){
    System.out.print("B.f() s=" + s + " ");
  }
  String s = "Initialized";
}
class Test
{
  public static void main(String[] args) {
    new A();
    System.out.println();
    new B();
  }
}
```

Rezultatem wykonania jest

```
A A.f()
A B.f() s=null B
```

- Przy konstrukcji obiektu klasy `A` w wyrażeniu `new A()` wywoływana jest metoda `A.f()`.

- Przy konstrukcji obiektu klasy `B` konstruktor klasy bazowej A wywoła przedefiniowaną metodę `B.f()` , która odwoła się do atrybutu `String s` przed jego inicjalizacją.
- Wywołanie metod polimorficznych w konstruktorach klasy bazowej może więc prowadzić do trudnych do zidentyfikowania błędów, które będą narzucały konieczność przeprojektowania aplikacji – np.: odwołanie do pliku przed jego otwarciem, próba wyświetlenia tekstów przed ich inicjalizacją, itd.

Konstruktor kopiujący i operator przypisania

Programiści języka C++ przyzwyczajeni są do tego, że język zapewnia wbudowane mechanizmy kopiowania zawartości obiektów. Kompilator C++ jest w stanie automatycznie wygenerować konstruktor kopiujący, który zainicjuje obiekt kopiując zawartość kolejnych pól innego obiektu. W podobny sposób jest realizowana standardowa implementacja operatora przypisania.

W języku Java konstruktor kopiujący (lub jego odpowiednik) musi zostać zaimplementowany i wywołany jawnie. Standardowy operator przypisania kopiuje jedynie wartości referencji albo typów wbudowanych, natomiast nie kopiuje całych obiektów.

Przykład

```
class Person
{
  String forename;
  String surname;
  int age;
  Person(String f, String s, int age) {
    forename = f;
    surname = s;
    this.age=age;
  }
// "konstruktor kopiujący"
  Person(Person other){
    forename=other.forename;
    surname=other.surname;
    age = other.age;
  }
  // porównanie zawartości obiektów
  boolean equals(Person other){
    return forename.equals(other.forename) &&
    surname.equals(other.surname) &&
    age==other.age;
  }
}
class Test{
  public static void main(String[] args) {
    Person p1 =
       new Person("Jan","Kowalski",25);
    Person p2 = p1;
    System.out.println(p1==p2);
    p2=new Person(p1);
    System.out.println(p1==p2);
    System.out.println(p1.equals(p2));
  }
}
```

Rezultat: `true, false, true.`

Klonowanie

Twórcy języka Java przewidzieli specjalną metodę (oraz interfejs), której zadaniem jest sporządzenie kopii obiektu.

Metoda ta jest zadeklarowana w klasie `Object` jako:

```
protected Object clone() throws
CloneNotSupportedException{...}
```

Klasa, w której metoda jest przedefiniowana powinna być podklasą interfejsu `Cloneable`.

Standardowa implementacja tej metody w klasie `Object` jest następująca:

- Sprawdza się, czy klasa implementuje interfejs `Cloneable`. Jeżeli nie, generowany jest wyjątek `CloneNotSupportedException`.
- Tworzony jest nowy obiekt klasy. Jego wszystkim polom przypisuje się wartości pól oryginalnego obiektu. Kopiowane są więc wartości pól typów wbudowanych oraz referencji. Nie są klonowane obiekty wskazywane przez referencję.
  W tym implementacja metody `clone` podobna jest do działania operatora przypisania i konstruktora kopiującego C++.

Przykład

```
class Person implements Cloneable
{
  String forename;
  String surname;
  int age;
// …inne metody
  public Object clone()
  throws CloneNotSupportedException{
      return super.clone();
  }
}
```

Kod testujący

```
class Test{
  public static void main(String[] args) {
  try {
        Person p1 = new Person("Jan","Kowalski");
        Person p2 = (Person) p1.clone();
        System.out.println(p1.forename==p2.forename);
        System.out.println(p2.surname==p1.surname);
        System.out.println(p2.age==p1.age);
      }
      catch (CloneNotSupportedException ex) {
      }
  }
}
```

Rezultat: `true, true, true`

Powyższy przykład pokazuje, że wbudowany mechanizm klonowania sporządza „płytką” kopię obiektu. Pola typu referencyjnego oryginalnego obiektu i kopii wskazują te same obiekty (wyrażenia postaci `p1.forename==p2.forename` mają wartość `true`).

- W przypadku klasy `Person` takie działanie jest wystarczające, ponieważ obiekty klasy `String` , na które wskazują atrybuty `forename` i `surname` są niemodyfikowalne (brak jest metody, która mogłaby zmodyfikować zawartość obiektu klasy `String`).
- Zmiana zawartości jednego z obiektów klasy `Person`, np.: przypisanie `p2.forename = "Andrzej"` nie zmieni zawartości obiektu klasy `String` wskazywanego przez `p2.forename`, ale po prostu zmieni wartość referencji, tak aby wskazywała inny obiekt. Tym samym, modyfikacja obiektu wskazywanego przez `p2` nie przeniesie się na zmianę zawartości obiektu wskazywanego przez `p1`.
- W przypadku, kiedy polami klasy są referencje do modyfikowalnych obiektów, implementacja metody `clone` powinna wpierw wywołać `super.clone()`, a następnie sporządzić kopie wybranych obiektów (modyfikowalnych).

Klonowanie tablic

Obiekty reprezentujące tablice implementują interfejs `Cloneable`. Wywołanie metody `clone()` obiektu typu tablicowego stworzy nowy obiekt będący kopią oryginalnego.

- Jeśli elementami tablicy są typy wbudowane, wówczas kopia obiektu będzie zawierała identyczne wartości.
- Jeżeli tablica zawiera referencje, wówczas referencje umieszczone w kopii będą wskazywały te same obiekty, co referencje umieszczone w oryginalnej tablicy.

Przykład

```
class Test{
  public static void main(String[] args) {
    int []a= new int[]{1,2,3};
    int []b = (int [])a.clone();
    a[0]=0;
    for(int i=0;i<a.length;i++)
      System.out.print(a[i]+" ");
    for(int i=0;i<b.length;i++)
      System.out.print(b[i]+" ");
  }
}
```

Rezultat: `0 2 3` **`1 2 3`**

Usuwanie obiektów

W każdym obiektowym języku programowania powołane do życia obiekty mogą uzyskiwać dostęp do zasobów systemowych lub zmieniać parametry systemu. W momencie ich usuwania powinny zwalniać te zasoby lub przywracać wcześniejszy stan otoczenia.

W języku C++ to zadanie jest powierzone destruktorowi, który na przykład powinien:

- zwolnić dynamicznie przydzieloną pamięć
- zamknąć otwarty plik
- usunąć wątek
- usunąć pliki tymczasowe
- zamknąć połączenie sieciowe
- przywrócić zmodyfikowane ustawienia systemowe

W języku C++ najczęstszym zadaniem destruktora jest zwalnianie dynamicznie przydzielonej pamięci.

Proces usuwania obiektów i zwalniania pamięci w języku Java odbywa się automatycznie za pośrednictwem mechanizmu *garbage collector*. Jednakże mechanizm ten nie jest w stanie zwolnić innych zasobów systemowych lub przywrócić stan otoczenia programu.

Z myślą o podobnych zastosowaniach w klasie `Object` zadeklarowano metodę `finalize()`:

```
protected void finalize() throws Throwable {}
```

Metoda ta jest wywoływana, co najwyżej jeden raz, bezpośrednio przed zwolnieniem pamięci obiektu. Jeżeli przy usunięciu obiektu danej klasy powinny zostać przeprowadzone dodatkowe zadania zwalniające zasoby systemowe lub przywracające stan otoczenia, odpowiedzialny za nie kod może zostać umieszczony w przedefiniowanej metodzie `finalize()`.

Pomiędzy metodą `finalize()` i destruktorem istnieją jednak zasadnicze różnice. Język C++ definiuje ściśle czas życia rozmaitych obiektów i dzięki temu programista ma pełną kontrolę nad tym, kiedy zostanie wywołany destruktor.

W języku Java brak jest gwarancji, kiedy i czy w ogóle dojdzie do zwolnienia pamięci obiektu przez maszynę wirtualną i tym samym wywołania metody `finalize()`.

Teoretycznie, mechanizm *garbage collect* wykonywany jest przez działający w tle wątek o niskim priorytecie. W większości implementacji, jednak, jest on uruchamiany w momencie, kiedy zaczyna brakować pamięci.

Przykład

```
class FinalizationTest
{
  public void finalize()
  {
    System.out.println("finalized");
  }
  public static void main(String[] args) {
    new FinalizationTest();
    // System.gc();
    // System.runFinalization();
  }
}
```

- Uruchomienie powyższego przykładu pokazało, że metoda `finalize()` nie została nigdy wywołana, ponieważ program zakończył działanie, zanim uaktywnił się wątek odpowiedzialny za zwalnianie pamięci. Pamięć została zwolniona w całości, przy kończeniu pracy maszyny wirtualnej.
- Uruchomienie mechanizmu *garbage collect* można wymusić wołając jawnie `System.gc()`.

- Możliwe jest też przyspieszenie finalizacji nieużywanych obiektów – `System.runFinalization()` – jednakże metoda `finalize()` zostanie wywołana wyłącznie dla tych obiektów, które podczas wykonywania mechanizmu *garbage collect* zostały zaznaczone jako nieużywane.

Oznacza to, że jeżeli konieczne są dodatkowe operacje związane z destrukcją obiektu, jak np.: zamknięcie pliku, zatrzymanie wątku, zwolnienie pamięci alokowanej w metodach `native` – operacje takie powinny być oprogramowane jawnie, a nie umieszczane w funkcji `finalize()`.

Po co w takim razie używać metody `finalize()`?

- Dokumentacja języka Java sugeruje, że należy umieszczać tam właśnie kod odpowiedzialny za zwalnianie zasobów, jak np.: zamknięcie pliku. W bibliotecznych implementacjach klas obsługujących pliki w metodach `finalize()` pojawia się wywołanie metody `close()` odpowiedzialnej za zamknięcie pliku.
- Bruce Eckel, autor Thinking in Java, proponuje, aby wykorzystać metodę `finalize()` do testowania, czy usuwany obiekt jest w poprawnym stanie końcowym. Tego typu informacje mogą być pomocą przy uruchamianiu oprogramowania.