

PODSTAWY PROGRAMOWANIA W JAVA

dr inż. Michał Tomaszewski

katedra Metod Programowania
Polsko-Japońska Akademia Technik Komputerowych

PARADYGMAT

CZYM JEST PARADYGMAT?

... przyjęty sposób widzenia rzeczywistości w danej dziedzinie ...

- Wikipedia

CZYM JEST PARADYGMAT?

... przyjęty sposób widzenia rzeczywistości w danej dziedzinie ...

- Wikipedia

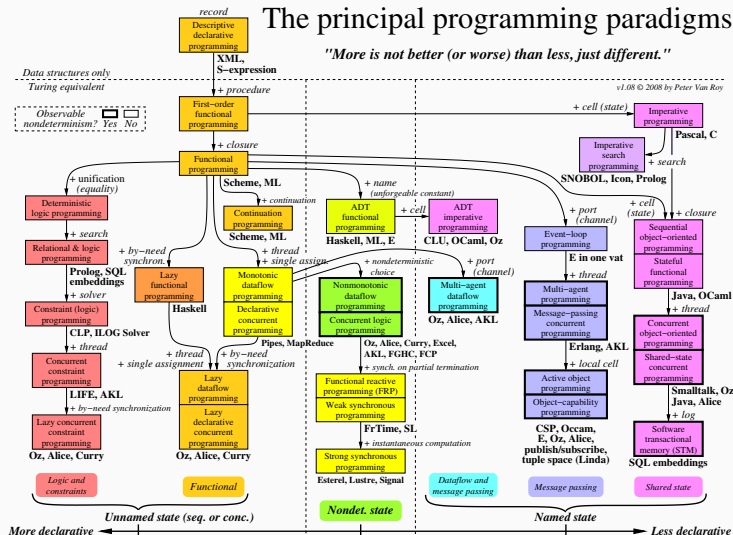
Paradygmat programowania definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego.

- Wikipedia

The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.08 © 2008 by Peter Van Roy



Explanations

See "Concepts, Techniques, and Models of Computer Programming".

The chart classifies programming paradigms according to their kernel languages (the small core language in which all the paradigm's abstractions can be defined). Kernel languages are ordered according to the creative extension principle: a new concept is added when it cannot be encoded with only local transformations. Two languages that implement the same paradigm can nevertheless have very different "flavors" for the programmer, because they make different choices about what programming techniques and styles to facilitate.

When a language is mentioned under a paradigm, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment.

State is the ability to remember information, or more precisely, to store a sequence of values in time. Its expressive power is strongly influenced by the paradigm that contains it. We distinguish four levels of expressiveness, which differ in whether the state is unnamed or named, deterministic or nondeterministic, and sequential or concurrent. The least expressive is functional programming (threaded state, e.g., DCGs and monads: unnamed, deterministic, and sequential). Adding concurrency gives declarative concurrent programming (e.g., synchrocells: unnamed, deterministic, and concurrent). Adding nondeterministic choice gives concurrent logic programming (which uses stream mergers: unnamed, nondeterministic, and concurrent). Adding ports or cells, respectively, gives message passing or shared state (both are named, nondeterministic, and concurrent). Nondeterminism is important for real-world interaction (e.g., client/server). Named state is important for modularity.

Axes orthogonal to this chart are typing, aspects, and domain-specificity. Typing is not completely orthogonal: it has some effect on expressiveness. Aspects should be completely orthogonal, since they are part of a program's specification. A domain-specific language should be definable in any paradigm (except when the domain needs a particular concept).

Metaprogramming is another way to increase the expressiveness of a language. The term covers many different approaches, from higher-order programming, syntactic extensibility (e.g., macros), to higher-order programming combined with syntactic support (e.g., meta-object protocols and generics), to full-fledged tinkering with the kernel language (introspection and reflection). Syntactic extensibility and kernel language tinkering in particular are orthogonal to this chart. Some languages, such as Scheme, are flexible enough to implement many paradigms in almost native fashion. This flexibility is not shown in the chart.

Paradygmat programowania zorientowanego obiektowo zasadza się na tworzeniu obiektów, które zawierają dane i metody.

Paradygmat programowania zorientowanego obiektowo zasadza się na tworzeniu obiektów, które zawierają dane i metody.

Programując obiektowo, należy wyróżnić trzy kluczowe cechy:

Paradygmat programowania zorientowanego obiektowo zasadza się na tworzeniu obiektów, które zawierają dane i metody.

Programując obiektowo, należy wyróżnić trzy kluczowe cechy:

- zachowanie obiektu – co możemy zrobić z obiektem, czyli jakie metody możemy wywołać na jego rzecz;

Paradygmat programowania zorientowanego obiektowo zasadza się na tworzeniu obiektów, które zawierają dane i metody.

Programując obiektowo, należy wyróżnić trzy kluczowe cechy:

- zachowanie obiektu – co możemy zrobić z obiektem, czyli jakie metody możemy wywołać na jego rzecz;
- stan obiektu – jak obiekt zachowuje się gdy wywołujemy metody?

Paradygmat programowania zorientowanego obiektowo zasadza się na tworzeniu obiektów, które zawierają dane i metody.

Programując obiektowo, należy wyróżnić trzy kluczowe cechy:

- zachowanie obiektu – co możemy zrobić z obiektem, czyli jakie metody możemy wywołać na jego rzecz;
- stan obiektu – jak obiekt zachowuje się gdy wywołujemy metody?
- identyfikacje obiektu – czy obiekt różni się od innych obiektów, mających ten sam stan i zachowanie.

OOP - JAVA

Klasa jest opisem rodziny obiektów.

Klasa jest opisem rodziny obiektów.

Klasa jest opisem rodziny obiektów.


Definicja klasy, opisującej obiekty reprezentujące ciasteczka z rodzynkami ma postać:

```
class Cookie {  
    int weight;  
    String ingredient;  
  
    Cookie(int weight, String ingredient){  
        this.weight = weight;  
        this.ingredient = ingredient;  
    }  
  
    Cookie getHalf(){  
        // ...  
    }  
}
```

Klasa jest opisem rodziny obiektów.

Definicja klasy, opisującej obiekty reprezentujące ciasteczka z rodzynkami ma postać:

```
class Cookie {  
    int weight;  
    String ingredient;  
  
    Cookie(int weight, String ingredient){  
        this.weight = weight;  
        this.ingredient = ingredient;  
    }  
  
    Cookie getHalf(){  
        // ...  
    }  
}
```



Klasa jest opisem rodziny obiektów.

Definicja klasy, opisującej obiekty reprezentujące ciasteczka z rodzynkami ma postać:

```
class Cookie {  
    int weight;  
    String ingredient;  
  
    Cookie(int weight, String ingredient){  
        this.weight = weight;  
        this.ingredient = ingredient;  
    }  
  
    Cookie getHalf(){  
        // ...  
    }  
}
```

pola

konstruktor

Klasa jest opisem rodziny obiektów.

Definicja klasy, opisującej obiekty reprezentujące ciasteczka z rodzynkami ma postać:

```
class Cookie {  
    int weight;  
    String ingredient;  
  
    Cookie(int weight, String ingredient){  
        this.weight = weight;  
        this.ingredient = ingredient;  
    }  
  
    Cookie getHalf(){  
        // ...  
    }  
}
```

pola

konstruktor

metody

Do utworzenia obiektu służy konstruktor (fabrykator).

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora z nazwą klasy, na przykład:

```
new Cookie( 10, "raisins")
```

powoduje utworzenie **obektu** klasy.

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

1. utworzenia obiektu o elementach opisanych przez pola;

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

1. utworzenia obiektu o elementach opisanych przez pola;
2. utworzenie zmiennej **this** zainicjowanej odniesieniem do obiektu;

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

1. utworzenia obiektu o elementach opisanych przez pola;
2. utworzenie zmiennej **this** zainicjowanej odniesieniem do obiektu;
3. wstępnego zainicjowania elementów obiektu danymi opisanymi przez inicjatory zawarte w deklaracjach pól;

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

1. utworzenia obiektu o elementach opisanych przez pola;
2. utworzenie zmiennej **this** zainicjowanej odniesieniem do obiektu;
3. wstępnego zainicjowania elementów obiektu danymi opisanymi przez inicjatory zawarte w deklaracjach pól;
4. wywołanie konstruktora w celu ostatecznego zainicjowania elementów obiektu;

Do utworzenia obiektu służy konstruktor (fabrykator).

Opracowanie fabrykatora składa się z:

1. utworzenia obiektu o elementach opisanych przez pola;
2. utworzenie zmiennej **this** zainicjowanej odniesieniem do obiektu;
3. wstępnego zainicjowania elementów obiektu danymi opisanymi przez inicjatory zawarte w deklaracjach pól;
4. wywołanie konstruktora w celu ostatecznego zainicjowania elementów obiektu;
5. dostarczenia w miejscu fabrykacji odniesienia do obiektu;

W deklaracji pól (składników) klasy może wystąpić specyfikator dostępu:

W deklaracji pól (składników) klasy może wystąpić specyfikator dostępu:

- `private` – jest dostępny tylko z ciała klasy;

W deklaracji pól (składników) klasy może wystąpić specyfikator dostępu:

- `private` – jest dostępny tylko z ciała klasy;
- `public` – jest dostępny wszędzie;

W deklaracji pól (składników) klasy może wystąpić specyfikator dostępu:

- `private` – jest dostępny tylko z ciała klasy;
- `public` – jest dostępny wszędzie;
- `protected` – jest dostępny tylko z ciała klasy oraz z ciała jego klasy pochodnej.

W deklaracji pól (składników) klasy może wystąpić specyfikator dostępu:

- `private` – jest dostępny tylko z ciała klasy;
- `public` – jest dostępny wszędzie;
- `protected` – jest dostępny tylko z ciała klasy oraz z ciała jego klasy pochodnej.

Zaleca się, aby **pola** były deklarowane jako prywatne albo chronione, a metody jako publiczne.

Polem **klasowym** jest pole zadeklarowane ze specyfikatorem `static`.

Polem **klasowym** jest pole zadeklarowane ze specyfikatorem **static**.

Pole klasowe opisuje zmienną, która jest **wspólna** dla wszystkich obiektów klasy i istnieje nawet wówczas, gdy nie utworzono **ani jednego** obiektu klasy.

Polem **klasowym** jest pole zadeklarowane ze specyfikatorem `static`.

Pole klasowe opisuje zmienną, która jest **wspólna** dla wszystkich obiektów klasy i istnieje nawet wówczas, gdy nie utworzono **ani jednego** obiektu klasy.

Polem **obiekowym** jest pole zadeklarowane bez specyfikatora `static`.

Polem **klasowym** jest pole zadeklarowane ze specyfikatorem `static`.

Pole klasowe opisuje zmienną, która jest **wspólna** dla wszystkich obiektów klasy i istnieje nawet wówczas, gdy nie utworzono **ani jednego** obiektu klasy.

Polem **obiekowym** jest pole zadeklarowane bez specyfikatora `static`. Każdemu polu obiekowemu odpowiada dokładnie jeden element obiektu klasy.

W Java znajdujemy trzy rodzaje metod:

W Java znajdujemy trzy rodzaje metod:

- klasową – zadeklarowaną ze specyfikatorem `static`;

W Java znajdujemy trzy rodzaje metod:

- klasową – zadeklarowaną ze specyfikatorem **static**;
- konstruktorową - jest metodą, której nazwa jest identyczna z nazwą klasy i w której definicji (nagłówku) nie występuje typ rezultatu;

W Java znajdujemy trzy rodzaje metod:

- klasową – zadeklarowaną ze specyfikatorem **static**;
- konstruktorową - jest metodą, której nazwa jest identyczna z nazwą klasy i w której definicji (nagłówku) nie występuje typ rezultatu;
- obiektową – nie konstruktorową i **bez** specyfikatora **static**.

Metody **klasowe** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Metody **klasowe** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Metody klasowe mogą się odwoływać **tylko** do **klasowych** pól lub metody.

Metody **klasowe** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Metody klasowe mogą się odwoływać **tylko** do **klasowych** pól lub metody.

```
class SpecialNumbers {  
    static final double c = 299792458.0; //[m/s]  
  
    static boolean isHappyNumber(int val){  
        //...  
    }  
  
    static void testFunc(){  
        System.out.println(c);  
    }  
}
```

Metody **klasowe** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Metody klasowe mogą się odwoływać **tylko** do **klasowych** pól lub metody.

```
class SpecialNumbers {  
    static final double c = 299792458.0; //[m/s]  
  
    static boolean isHappyNumber(int val){  
        // ...  
    }  
  
    static void testFunc(){  
        System.out.println(c);  
        System.out.println(isHappyNumber(153));  
    }  
}
```

Metody **klasowe** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Metody klasowe mogą się odwoływać **tylko** do **klasowych** pól lub metody.

```
class SpecialNumbers {  
    static final double c = 299792458.0; //[m/s]  
  
    static boolean isHappyNumber(int val){  
        //...  
    }  
  
    static void testFunc(){  
        System.out.println(SpecialNumbers.c);  
        System.out.println(  
            SpecialNumbers.isHappyNumber(153)  
        );  
    }  
}
```

DZIĘKUJĘ