

Czy stado słońi jest stadem zwierząt? O kowariancji i kontrawariancji

Paweł LIPSKI*

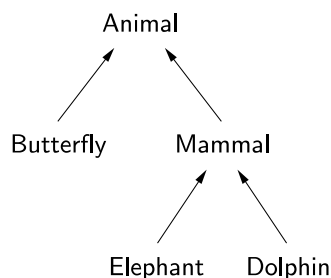
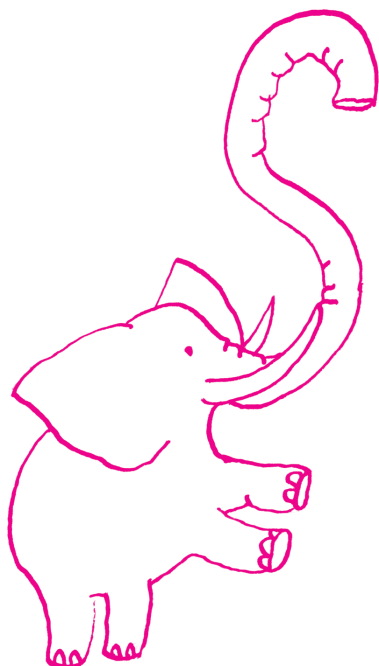


Diagram klas dla zamieszczonego kodu, w którym strzałkami zaznaczono relację dziedziczenia.



Rozwiązanie zadania M 1434.

Niech a , b , c oznaczają liczbę głównych przekątnych odpowiednio o dwóch końcach czerwonych, o dwóch końcach niebieskich i o końcach w różnych kolorach. Ponieważ każdy wierzchołek jest końcem dokładnie jednej głównej przekątnej, więc mamy $2a + c$ czerwonych wierzchołków i $2b + c$ niebieskich. Ponieważ liczby te są równe, otrzymujemy $a = b$.

Informatycy wiedzą, że odpowiedzi na pewne z pozoru filozoficzne pytania wcale nie tak trudno znaleźć. Wystarczy napisać i uruchomić odpowiedni kod. W taki też sposób podejmiemy do problemu postawionego w tytule artykułu.

Na wstępie powiedzmy sobie trochę o *programowaniu obiektowym* – to właśnie z jego pomocą będziemy modelować w kodzie tytułowe „stada”, „słonie” i inne „zwierzęta”. Zapewne spora grupa czytelników spotkała się z programowaniem obiektowym, choćby przy okazji używania kontenerów STL w języku C++, często używanych w rozwiązywaniach zadań olimpijskich.

Najogólniej, *Object-Oriented Programming* (OOP) jest paradygmatem programowania. Jego podstawowymi pojęciami są *klasy* i *obiekty*. Klasę możemy rozumieć jako typ danych. Klasami mogą być np. zwierzę, ssak, słoń, delfin – rozumiane jako *rodzaje* istot żywych, a nie jako *konkretni przedstawiciele* tych rodzajów. Obiektem klasy słoń natomiast będzie np. Jumbo – a więc pewien konkretny przedstawiciel tego gatunku. Jumbo (i generalnie każdy słoń) *jest* w ogólności także ssakiem. I dalej, każdy ssak *jest* w ogólności zwierzęciem.

Ta relacja „*jest*” mówi nam o *dziedziczeniu*. Jeśli każdy obiekt klasy Elephant jest także obiektem klasy Mammal, to mówimy, że klasa Elephant *dziedziczy* z klasy (albo inaczej: *rozszerza* klasę) Mammal.

Przełożmy teraz te dywagacje na kod (patrz też rysunek):

```
class Animal {
    def move { ... }
}
class Butterfly extends Animal {
    def fly { ... }
}
class Mammal extends Animal {
    def feedYoung { ... }
}
class Elephant extends Mammal {
    def trumpet { ... }
}
class Dolphin extends Mammal {
    def swim { ... }
}
```

Powyższy fragment kodu, jak i wszystkie kolejne, są napisane w języku programowania Scala. Język ten powstał jako następcą szeroko używanego języka Java. Dowolnej biblioteki napisanej w Javie można użyć z poziomu kodu źródłowego w Scali. Scala zapewnia bardzo dokładną kontrolę poprawności kodu (zwłaszcza poprawności typów – jest tzw. *type-safe language*) na etapie kompilacji, dzięki czemu można uniknąć wielu błędów powstających już po uruchomieniu programu.

Oprócz słów kluczowych `class` i `extends`, deklarujących klasy i ich relacje dziedziczenia, używamy też słowa kluczowego `def`. Służy ono ogólnie do definiowania funkcji, a w tym przypadku dokładniej *metod klas*. Metody można rozumieć jako „osobiste” funkcje obiektów klasy. Jeśli mamy obiekt `animal` klasy `Animal`, to możemy na nim wywołać metodę `animal.move()`. Dzięki dziedziczeniu metoda `move` może zostać wywołana również na obiekcie klasy `Mammal` lub `Elephant` (choć co dokładnie zrobi obiekt po wywołaniu tej metody, zależy od tego, jak zostanie ona zaimplementowana w odpowiedniej klasie). Natomiast na obiekcie klasy `Butterfly` nie można wywołać metody `feedYoung` ani `trumpet`. W powyższych przykładach wnętrza metod pozostawiliśmy jednak niezdefiniowane, bo nie będą miały one dla nas większego znaczenia.

*student, Katedra Informatyki, Akademia Górniczo-Hutnicza, Kraków



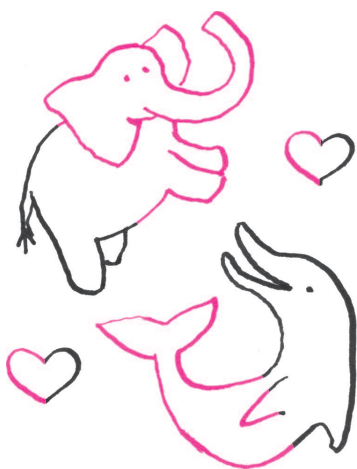
Rozwiązanie zadania M 1433.

Mamy $f(x) = (x - b/2)^2 + c - b^2/4$. Możemy bez utraty ogólności przyjąć więc, że $b = 0$ (zmiana wartości parametrów na $c = c + b^2/4$ oraz $b = 0$ przesuwa wykres w poziomie, co nie zmienia zbioru A). Mamy kilka przypadków.

1. $c > 1$; wtedy $A = \emptyset$.

2. $1 \geq c \geq -1$; wtedy A jest odcinkiem $[-\sqrt{1-c}, \sqrt{1-c}]$ o długości $2\sqrt{1-c} \leq 2\sqrt{2}$.

3. $c < -1$; wtedy $A = -I \cup I$, gdzie $I = [\sqrt{-1-c}, \sqrt{1-c}]$. Łączna długość tych dwóch odcinków wynosi $2(\sqrt{1-c} + \sqrt{-1-c}) = \frac{4}{\sqrt{1-c} + \sqrt{-1-c}} \leq \frac{4}{\sqrt{2}} = 2\sqrt{2}$.



Nowy obiekt tworzymy za pomocą słowa kluczowego `new`. Obiekty są w Javie i Scali przekazywane do funkcji przez referencję (sposób podobny do przekazania przez wskaźnik w C++), a więc przy wywołaniu nie jest tworzona żadna lokalna kopia. Ponadto, jeśli mamy obiekt klasy `Dolphin`, `Mammal`, `Butterfly` czy ogólnie klasy dziedziczącej po `Animal`, to możemy go przekazać do funkcji, która oczekuje obiektu klasy `Animal`. Dzięki dziedziczeniu obiekty klasy bardziej specyficznej nadal są obiektami klasy bardziej ogólnej:

```
def playWith(m: Mammal) { ... }
var elephant = new Elephant
var butterfly = new Butterfly
playWith(elephant)           - w porządku, słoń jest ssakiem
playWith(butterfly)          - błąd! motyl nie jest ssakiem
```

Przejdźmy teraz do *generyków*. Jest to mechanizm bardzo podobny do wzorców (*templates*) znanych z STL. Najlepiej wyjaśni przykład:

```
class Herd[T] {
  def getLatestMember: T = { ... }
  def addMember(x: T) { ... }
}
```

Mamy stado, które może trzymać obiekty dowolnego typu (niekoniecznie nawet dziedziczące po klasie `Animal`). Metoda `getLatestMember` zwraca ostatnio dodanego (za pomocą metody `addMember`) członka stada. Możemy stworzyć stado słoń (wywołując `new Herd[Elephant]`) i takie stado może przechowywać słonie, ale nie można do niego dołożyć np. delfina. Możemy stworzyć stado zwierząt (wykonując `new Herd[Animal]`) i takie stado może trzymać dowolne zwierzę (a więc zarówno słonie, jak i delfiny), ale nie można do niego dodać np. łańcucha znaków (obiekту klasy `String`).

Teraz widać już powoli, do czego zmierzamy. Jeśli mamy klasę `Herd` (albo dowolny inny generyk, np. `List`, `Vector`, `Set` itp.) obiektów klasy `Elephant`, to czy jest on w ogólności obiektem klasy `Herd[Mammal]`? Czy przekazanie `Herd[Elephant]` do funkcji, która oczekuje `Herd[Mammal]`, będzie poprawne i bezpieczne?

Żałujemy, że tak faktycznie jest i że stado słoń *jest* stadem ssaków, a w ogólności klasa `Herd[Y]` jest `Herd[X]` dla typu `Y` dziedziczącego po `X`. Zobaczmy, co się stanie w poniższym kodzie (`wakeLatestMember` jest zwykłą funkcją, nie metodą):

```
def wakeLatestMember(herd: Herd[Mammal]) {
  herd.getLatestMember().feedYoung()
}
var elephantHerd = new Herd[Elephant]
elephantHerd.addMember(new Elephant)
wakeLatestMember(elephantHerd)
```

Kiedy uruchomimy funkcję `wakeLatestMember` na ostatnio dodanym do `elephantHerd` ssaku (czyli słońcu), zostanie wywołana metoda `feedYoung` – jest to zupełnie w porządku, bo słonie są ssakami.

Jest jednak druga strona medalu. Skoro `elephantHerd` jest stadem ssaków, powinniśmy móc do niego dodać *zupełnie* dowolnego ssaka.

```
def addDolphin(herd: Herd[Mammal]) {
  herd.addMember(new Dolphin)
}
```

W funkcji `addDolphin` parametr `herd` jest przekazywany przez referencję, więc dodawany delfin pozostanie w stadzie po zakończeniu działania funkcji.

Gdybyśmy teraz uruchomili `addDolphin(elephantHerd)`, to na pierwszy rzut oka nie byłoby problemu. Jednak wywołajmy teraz

```
elephantHerd.getLatestMember().trumpet()
```

Jest to jak najbardziej dozwolone – w końcu `elephantHerd` jest `Herd[Elephant]`, stadem słoń. Tymczasem okazuje się, że `getLatestMember` zwróci... delfina! Próba zawołania `trumpet` na obiekcie klasy `Dolphin` skończyłaby się błędem. (W istocie kompilator Scali uzna taki kod za niepoprawny.)

Spróbujmy więc z drugiej strony – założmy, że stado zwierząt (tak, *dowolnych* zwierząt!) jest stadem słoń. Każdy obiekt klasy `Herd[Animal]` musiałby w ogólności być obiektem `Herd[Elephant]`. Generalnie, przyjmijmy, że `Herd[Y]` jest `Herd[X]` wtedy i tylko wtedy, gdy `X` jest (dziedziczy po) `Y`. Wydaje się to z pozoru absurdalne, ale może akurat okaże się, że takie założenie nie prowadzi do błędów?

```
var animalHerd = new Herd[Animal]
addDolphin(animalHerd)
```

Funkcja `addDolphin` wymaga parametru typu `Herd[Mammal]`, a zgodnie z naszym założeniem `Herd[Animal]` jest `Herd[Mammal]`. Teraz nie ma żadnego problemu – `addDolphin` dodaje do stada delfina.

Po powrocie z `addDolphin` możemy bezpiecznie wywołać `animalHerd.getLatestMember().move()`

Jednak taki sposób dziedziczenia ma poważny mankament. Spójrzmy jeszcze raz na funkcję `wakeLatestMember`. Zgodnie z nową logiką możemy do niej przekazać obiekt klasy `Herd[Animal]` – np. stado składające się z jednego motyla:

```
var butterflySwarm = new Herd[Animal]
butterflySwarm.addMember(new Butterfly)
wakeLatestMember(butterflySwarm)
```

Skutek jest prosty: wewnątrz `wakeLatestMember` będziemy chcieli wywołać `feedYoung` na motyłu! Znowu spowodowałoby to błąd w czasie wykonania programu, więc kompilator Scali zaprotestuje.

Gdybyśmy się zgodzili na dziedziczenie w którąkolwiek stronę między `Herd[Animal]` a `Herd[Elephant]`, mogłyby się zdarzyć (nieraz trudne do przewidzenia!) błędy czasu uruchomienia. W takiej sytuacji mówimy o klasie `Herd[T]`, że jest ona **inwariantna** w typie T .

Nie musi jednak być tak zawsze. (Reguły, które teraz podamy, są uproszczone – od razu mówię, że nie uwzględniamy one wszystkich przypadków.)

Gdybyśmy tak zmienili klasę `Herd[T]`, że nie byłoby w niej żadnej metody przyjmującej parametr typu T (lub generyka zawierającego T , np. `List[T]`), to okazałoby się, że `Herd[Elephant]` faktycznie jest `Herd[Animal]`! Przypomnijmy sobie – gdy stado słońi było stadem zwierząt, problem pojawiał się tylko przy metodzie *przyjmującej* obiekt typu T , a nie tej *zwracającej* obiekt typu T .

```
class CovariantHerd[+T] {
  def getLatestMember(): T = { ... }
  def getMaleMembers(): List[T] = { ... }
}
```

Mówimy, że klasa `CovariantHerd` jest **kowariantna** w typie T . Język Scala (przeciwieństwo niż Java i C++) pozwala wprost zadeklarować, że `CovariantHerd[Y]` dziedziczy po `CovariantHerd[X]` dla Y dziedziczącego po X . Służy do tego znak plus przed nazwą typu w deklaracji klasy. Kompilator musi jeszcze, oczywiście, zweryfikować, czy aby na pewno taka kowariancja nie doprowadzi do potencjalnych błędów.

Z drugiej strony, gdyby żadna metoda nie zwracała obiektu typu „zawierającego” typ T , to wtedy okazałoby się, że `Herd[Animal]` jest `Herd[Elephant]`.

```
class ContravariantHerd[-T] {
  def addMember(x: T) { ... }
  def removeMember(x: T) { ... }
}
```

Klasa `ContravariantHerd` jest **kontrawariantna** w typie T – sygnalizujemy to minusem przed T .

Często zdarza się, że klasa jest kowariantna w jednym typie, a kontrawariantna w drugim. Spójrzmy:

```
class Person { ... }
class Professor extends Person { ... }

class Publication { ... }
class Article extends Publication { ... }

def extractAuthors(func: Article => Person) { ... }
```

Powyższa składnia oznacza, że `extractAuthors` jako parametr przyjmuje funkcję. Jeśli teraz mamy funkcję, która dla danej publikacji zwraca obiekt profesora, który ją napisał:

```
def whose(p: Publication): Professor = { ... }
```

to możemy ją użyć jako parametr do funkcji pobierającej autorów: `extractAuthors(whose)`. Wszystko się zgadza – jeśli funkcja `whose` zostanie wywołana wewnątrz `extractAuthors`, to w parametrze podany zostanie jej pewien artykuł (który zawsze jest publikacją), a zwróci pewnego profesora (który zawsze jest osobą). Funkcja zwracająca profesora dla publikacji jest więc funkcją zwracającą osobę dla artykułu. Ogólnie, funkcje (w pewnym uproszczeniu w Scali to „obiekty klasy `Function`”) są kontrawariantne w typie parametru i kowariantne w typie zwracanym.

Mówiliśmy, że wcześniej podane reguły są uproszczone. Spójrzmy na przypadek, w którym przestają działać:

```
class Node[T, U, V] {
  def handle(item: T, next: Node[V, U, T]):
    Node[Node[V, U, T], U, V] = { ... }
}
```

Ciężko na pierwszy rzut oka określić, jaka jest właściwie wariancja typów T , U i V . Po sprawdzeniu, z jakimi kombinacjami $+$ i $-$ przy T , U , V kod się kompiluje, okazuje się, że klasa `Node` w typie T jest kontrawariantna, w typie U – inwariantna, a w typie V – kowariantna.

Kompilator zgodzi się na deklarację `Node[-T, U, +V]`, ale przy deklaracji `Node[-T, -U, +V]` stwierdzi, że minus przy U może doprowadzić do błędów w czasie działania programu. Notabene, `Node[-T, U, V]` albo `Node[T, U, V]` też są poprawne – nie spowodują błędów, jedynie niepotrzebnie ograniczają relacje dziedziczenia.

Gdy już oznaczymy typy w deklaracji klasy jako $-T$, U oraz $+V$, do funkcji przyjmującej parametr typu `Node[Article, String, Person]` można przekazać parametr typu np. `Node[Publication, String, Professor]`.

Te wnioski nie zgadzają się z tym, co mówiliśmy wcześniej – np. w V klasa jest kowariantna, a mimo to V pojawia się w typie parametru (`next`). Dokładne kryteria są w istocie dosyć złożone (a momentami nawet zaskakujące). Nie sposób ich tutaj jednak wyjaśnić – potrzebny byłby zapewne kolejny artykuł.

Trzymając się naszych uproszczonych reguł, wróćmy do tytułowego pytania. Odpowiedź można ująć tak: stado słońi *jest* stadem zwierząt, ale tylko wtedy, gdy nie przyjmuje żadnych obcych przybyszów. Ciekawy wniosek, czyż nie?