

Polimorfizm

Czego się dowiesz

Wiesz już, że w Javie istnieje możliwość tworzenia klas, które rozszerzają inne klasy, a tym samym tworzona jest pewna hierarchia typów. Na razie wiemy, że możemy zapisać np.:

```
Dog dog = new Dog("Burek");
```

Jednak jak się okazuje, ponieważ typ Dog (czyli pies) jest również zwierzęciem (klasa Animal) to w Javie poprawny będzie także zapis:

```
Animal dog = new Dog("Burek");
```

Możliwość przypisania obiektu typu bardziej szczegółowego do referencji ogólniejszego (nadrzędnego) typu nazywać będziemy **polimorfizmem**, który w programowaniu obiektowym odgrywa znaczącą rolę.

Stwórzmy trzy klasy: Animal (zwierzę) oraz Dog (pies) i Cat (kot), które będą rozszerzać tę pierwszą. W każdej zadeklarujemy pole name, a także metodę giveSound(), która sprawi, że zwierzak wyda dźwięk i się przedstawi.

Animal.java

```
class Animal {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public Animal(String name) {  
        setName(name);  
    }  
  
    public void giveSound() {  
        System.out.println("Jestem zwierzęciem i nazywam się " + getName());  
    }  
}
```

Dog.java

```
class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void giveSound() {
        System.out.println("Jestem psem i nazywam się " + getName());
    }
}
```

Cat.java

```
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void giveSound() {
        System.out.println("Jestem kotem i nazywam się " + getName());
    }
}
```

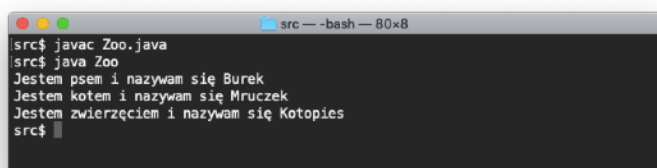
Jest to coś co już znamy - zwykłe nadpisanie metody `giveSound()`, jednak do tej pory nie używaliśmy takiego zapisu jak w poniższej klasie `Zoo`:

Zoo.java

```
class Zoo {
    public static void main(String[] args) {
        Animal dog = new Dog("Burek");
        Animal cat = new Cat("Mruczek");
        Animal doge = new Animal("Kotopies");

        dog.giveSound();
        cat.giveSound();
        doge.giveSound();
    }
}
```

Bardzo ciekawy jest w tej sytuacji wydruk programu:

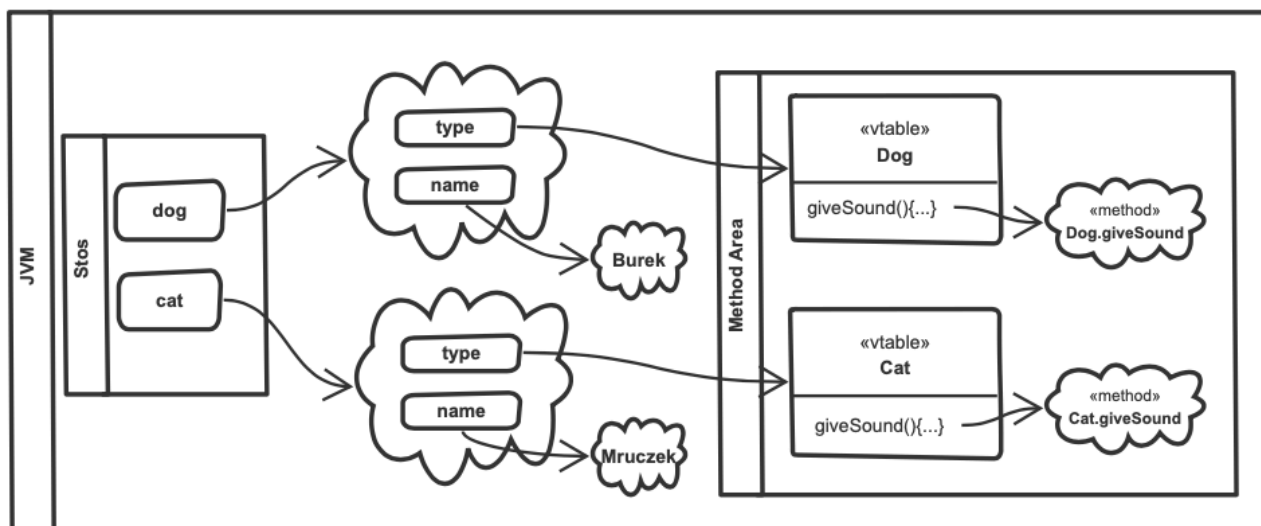


```
src$ javac Zoo.java
src$ java Zoo
Jestem psem i nazywam się Burek
Jestem kotem i nazywam się Mruczek
Jestem zwierzęciem i nazywam się Kotopies
src$
```

Pomimo iż zmienne `dog` i `cat` są zadeklarowane jako typ `Animal`, to w rzeczywistości przypisane są do nich obiekty typów bardziej sprecyzowanych `Dog` i `Cat` i to metody `giveSound()` zdefiniowane w tych klasach są wywoływane.

Zapamiętaj więc, że **metody instancji zawsze wywoływane są na typie obiektu, a nie referencji.**

Profesjonalnie mówimy tutaj o czymś takim jak wirtualne wywołanie metod. W Javie większość metod jest wirtualna i ich wywołanie najlepiej sobie wyobrazić odnosząc się do poniższego diagramu.



Każdy obiekt w Javie (chmurki na powyższym obrazku) posiada referencję na obiekt przechowujący informacje o swoim typie (klasie) oraz poszczególnych wartościach przypisanych do pól. Obiekty reprezentujące typ przechowywane są w przestrzeni pamięci o nazwie *Method Area*. Zapisane są w nich informacje np. o nazwie klasy, metodach w niej zdefiniowanych, czy klasie nadrzędnej. Wywołując metodę na jakiejś referencji, np. `dog.giveSound()`, przechodzimy do obiektu, na który referencja ta wskazuje. Obiekt posiada referencję na tablicę metod, które możemy na nim wywołać (vtable). Jeżeli odnajdzie w nim informację o metodzie `giveSound()` to jest ona wczytywana do pamięci. Jeżeli w klasie `Dog` nie byłoby metody `giveSound()` to w vtable byłaby referencja na metodę `giveSound()` z klasy `Animal`. Takie zachowanie, polegające na wyszukiwaniu metody, która ma zostać wywołana w trakcie działania programu nazywamy

dynamicznym lub późnym wiązaniem (dynamic dispatch). Zachowanie to może się różnić w zależności od tego z jakiej implementacji wirtualnej maszyny korzystasz (np. od oracle, albo ibm).

Polimorfizm będzie szczególnie przydatny, gdy chcesz w kolekcji (np. tablicy) przechowywać obiekty różnych typów, które mają jedną wspólną klasę bazową. Pozwoli to uprościć kwestię wywołania metod na wszystkich obiektach np. korzystając z pętli. Co więcej w taki sam sposób możesz definiować ogólniejsze metody, które będą mogły przyjmować argumenty różnych typów.

Klasę Zoo możemy w takiej sytuacji przerobić na poniższą, zachowując tę samą funkcjonalność:

```
Zoo.java
class Zoo {
    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Dog("Burek");
        animals[1] = new Cat("Mruczek");
        animals[2] = new Animal("Kotopies");

        for (Animal animal : animals) {
            animal.giveSound();
        }
    }
}
```

Teraz, żeby jeszcze lepiej wyobrazić sobie zalety tego rozwiązania, spójrz na to z takiej perspektywy, że klas dziedziczących po *Animal* jest 10, a obiektów w tablicy nie 3, ale 100. Iteracja po nich i wywołanie metody `giveSound()` na każdym obiekcie nadal zajmuje tylko 3 linijki kodu.

Na koniec jeszcze ostateczna forma klasy Zoo przedstawiająca zaletę korzystania z metody, która jako parametr przyjmuje typ ogólniejszy, a nie szczegółowy:

```
Zoo.java
class Zoo {
    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Dog("Burek");
        animals[1] = new Cat("Mruczek");
        animals[2] = new Animal("Kotopies");
    }
}
```

```

        changeAnimalName(animals[0], "Pieseł");

        for (Animal animal : animals) {
            animal.giveSound();
        }
    }

    private static void changeAnimalName(Animal animal, String newName) {
        animal.setName(newName);
    }
}

```

Dzięki temu, że metoda `changeAnimalName()` ma zdefiniowany parametr typu `Animal`, a nie typ szczegółowy `Dog`, czy `Cat`, to możemy do niej przekazać każdego zwierza. Nawet jeśli utworzymy nowe klasy dziedziczące po `Animal`, to ta metoda nadal będzie działała i będziemy mogli do niej przekazywać obiekty nowych typów, a jednocześnie nie będziemy musieli tworzyć przeciążonych wersji metody dla każdego z nich. Genialne!

Rzutowanie typów obiektowych

W jednej z początkowych lekcji pokazałem ci, że pomiędzy typami prostymi możesz wykorzystywać konwersje rozszerzające lub zawężające, czyli np. zmieniać *int* na *double*, albo *double* na *int*. Podobnie jest w przypadku typów obiektowych. Jeżeli pomiędzy klasami istnieje zależność dziedziczenia, to możemy zastosować **rzutowanie**, precyzując tym samym typ obiektu. Żeby zobaczyć do czego może ci to być potrzebne dodajmy w klasie `Dog` metodę `bark()` (szczekaj), a w klasie `Cat` metodę `meow()` (miaucz).

Dog.java

```

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void giveSound() {
        System.out.println("Jestem psem i nazywam się " + getName());
    }

    public void bark() {
        System.out.println("Hau hau!");
    }
}

```

Metoda bark() wyświetla tekst "Hau hau!".

Cat.java

```
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void giveSound() {
        System.out.println("Jestem kotem i nazywam się " + getName());
    }

    public void meow() {
        System.out.println("Miaaaaaau");
    }
}
```

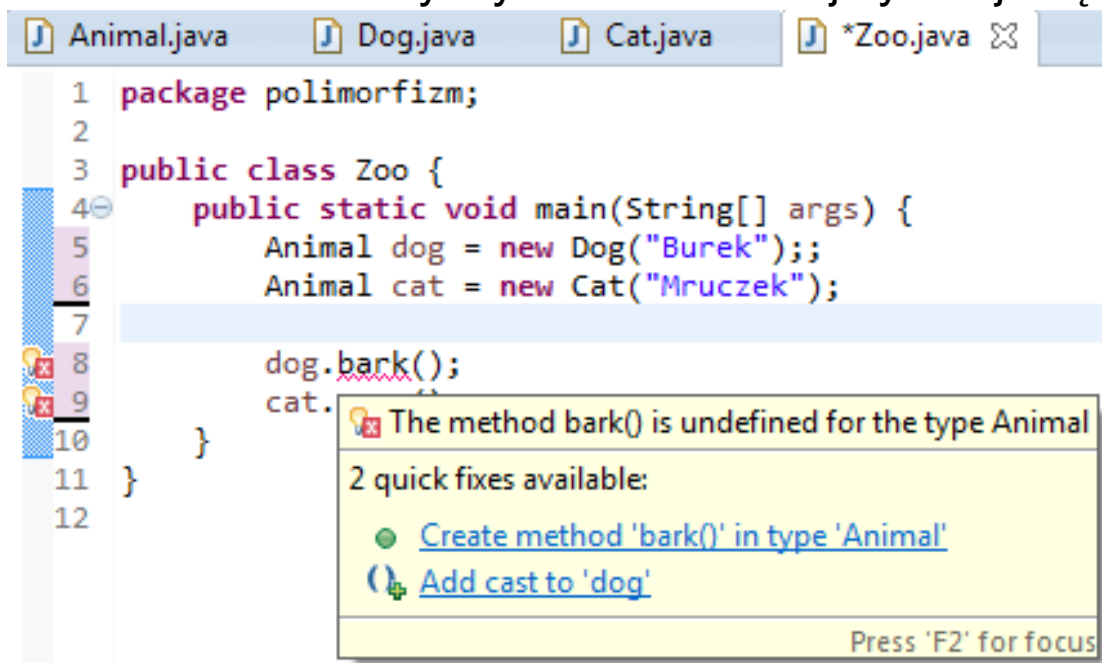
Metoda meow() wyświetla tekst "Miaaaaaau".

Teraz możemy przetestować ich działanie w klasie Zoo:

```
class Zoo {
    public static void main(String[] args) {
        Animal dog = new Dog("Burek");
        Animal cat = new Cat("Mruczek");

        dog.bark();
        cat.meow();
    }
}
```

Środowisko niestety wyświetla w takiej sytuacji błędy:



Zasada jest tutaj prosta. **Metody instancji zawsze wywoływane są na typie obiektu, ale żeby je wywołać, to metoda o wskazanej sygnaturze musi być dostępna**

w typie referencji. A bardziej ludzkimi słowami powiemy - żeby móc wywołać metodę bark(), to metoda ta musi być zdefiniowana w klasie Animal, albo referencja, na której wywołujesz tę metodę musi być typu Dog, czyli np.:

```
Dog dog = new Dog("Burek");  
dog.bark();
```

Nie oznacza to jednak, że musimy od razu rezygnować z polimorfizmu. Typ referencji możemy zmienić poprzez rzutowanie, podobnie jak było przy typach prostych. Robimy to zapisując nowy typ referencji w okrągłych nawiasach, przed referencją, której typ chcemy zmienić. Zrzuowaną referencję możemy przypisać do zmiennej, lub wywołać na niej metodę "w locie".

```
class Zoo {  
    public static void main(String[] args) {  
        Animal dog = new Dog("Burek");  
        Animal cat = new Cat("Mruczek");  
  
        // rzutowanie do zmiennej  
        Dog specificDog = (Dog) dog;  
        specificDog.bark();  
  
        // rzutowanie "w locie"  
        ((Cat) cat).meow();  
    }  
}
```

Rzutuując referencję dog na typ Dog i przypisując go do zmiennej specificDog uzyskujemy dostęp do metody bark(). Podobnie dzieje się w przypadku zmiennej cat, tyle, że jak widzisz jeżeli ma to być pojedyncze wywołanie metody, to nie musimy nawet wykorzystywać dodatkowej zmiennej. Możliwe jest także rzutowanie "w górę", czyli z typu konkretnego na typ ogólniejszy o czym już wiesz, ponieważ od początku tej lekcji robiliśmy to przypisując np. obiekt typu Dog do referencji typu Animal. Pamiętaj, że tracisz wtedy jednak dostęp do metod specyficznych dla obiektu klasy.

Rzutowanie może być jednak niebezpieczne i łatwo w nim o pomyłkę. Przykładowo poniższy kod:

```
Animal cat = new Cat("Mruczek");  
// rzutowanie "w locie"  
((Dog) cat).bark();
```

choć wydaje się bez sensu, to jest poprawny w sensie semantycznym i się skompiluje. Przy próbie uruchomienia programu otrzymamy jednak wyjątek *ClassCastException*, który jest rzucany ponieważ próbujemy zmienić typ referencji z Cat na Dog, a pomiędzy tymi dwoma klasami nie ma żadnej bezpośredniej relacji (kot nie jest psem i odwrotnie). W praktyce rzutowanie najczęściej oznacza więc, że z architekturą naszego kodu jest coś nie do końca tak, a jeżeli nie widzimy innego rozwiązania, to zawsze warto najpierw sprawdzić, czy rzutowanie na określony typ jest w ogóle możliwe. Możemy do tego wykorzystać operator `instanceof`.

```
Animal cat = new Cat("Mruczek");  
// rzutowanie "w locie"  
if (cat instanceof Dog) {  
    // rzutujemy referencję cat na Dog, tylko wtedy, kiedy jest do niej przypisany  
    // obiekt Dog. W tym przypadku if się nie wykona i nie dojdzie do rzutowania.  
    ((Dog) cat).bark();  
}
```