Dziedziczenie

Streszczenie

Celem wykładu jest omówienie tematyki dziedziczenia klas.
Czas wykładu – 45 minut.

Rozpatrzmy przykład przedstawiający klasy `Student` oraz `Pracownik`:

```
class Student
{
    String nazwisko;
    int rokUrodzenia;
    double[] oceny;
}

class Pracownik
{
    String nazwisko;
    int rokUrodzenia;
    double pensja;
}
```

Na potrzeby wykładu nie przedstawiamy ich pełnych charakterystyk, pomijając na przykład ważne dane jakim jest semestr studiów w przypadku studentów czy wykładane przedmioty w przypadku pracowników.

Wygenerowanie obiektu reprezentującego pewnego pracownika może wyglądać następująco:

```
class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p = new Pracownik();
        p.nazwisko = "Nazwisko";
        p.rokUrodzenia = 1960;
        p.pensja = 3512.0;
    }
}
```

Zauważmy, że część pól zadeklarowanych w obu klasach powtarza się (`nazwisko` oraz `rokUrodzenia`). Są to wspólne cechy opisujące studentów i pracowników najpierw jako osoby. Wydaje się zatem sensowne wydzielenie pewnej struktury przechowującej tylko dane osobowe. Przy podejściu strukturalnym można wydzielić klasę `Osoba` dokładając odpowiednie pola do obu klas `Student` oraz `Pracownik`, pola charakterystyczne tylko dla tych klas.

```
class Osoba
{
    String nazwisko;
    int rokUrodzenia;
}

class Student
{
    Osoba daneOsobowe;
    double[] oceny;
}

class Pracownik
{
    Osoba daneOsobowe;
    double pensja;
}
```

Wprawdzie płacimy definiując dodatkową klasę (`Osoba`), ale unikamy powtarzania deklarowania identycznych pól w różnych klasach. W omawianym przykładzie powtarzane były tylko dwa pola. Wydaje się, że strata niewielka. Ale gdybyśmy osoby opisywali jeszcze innymi cechami (np. miejsce urodzenia, PESEL, adres zamieszkania, itp), powtórzeń byłoby wiele więcej.

W przypadku wydzielenia klasy `Osoba` wygenerowanie obiektu reprezentującego pewnego pracownika może wyglądać następująco:

```
class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p = new Pracownik();
        p.daneOsobowe = new Osoba();
        p.daneOsobowe.nazwisko = "Nazwisko";
        p.daneOsobowe.rokUrodzenia = 1960;
        p.pensja = 3512.0;
    }
}
```

Java, jako język obiektowy, udostępnia inny mechanizm budowania klas z klas już zdefiniowanych, mechanizm zwany *dziedziczeniem*. Znamy go już z języka C++. W Javie występują jednak pewne znaczące różnice w stosunku do dziedziczenia w C++.

Zacznijmy od syntaktyki. W celu wskazania klasy nadrzędnej stosuje się słowo `extends`, w postaci:

```
class Osoba
{
    String nazwisko;
    int rokUrodzenia;
}
```

```
class Student extends Osoba
{
    double[] oceny;
}

class Pracownik extends Osoba
{
    double pensja;
}
```

Pola klasy `Osoba` (`nazwisko` oraz `rokUrodzenia`) są dostępne w klasach potomnych bez konieczności ich deklarowania ani explicite ani poprzez dodatkowe pole (np. `daneOsobowe`). Pewnego pracownika można zatem wygenerować następująco:

```
class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p = new Pracownik();
        p.nazwisko = "Nazwisko";
        p.rokUrodzenia = 1960;
        p.pensja = 3512.0;
    }
}
```

Dostęp do pól `nazwisko` oraz `rokUrodzenia` jest bezpośredni, tak jakby były one zadeklarowane w klasie `Pracownik`.

Java jest językiem *jednobazowym*. To znaczy, że dowolna klasa może mieć co najwyżej jednego przodka bezpośredniego. Ponadto istnieje klasa `Object`, która jest domyślnie przodkiem każdej innej klasy, nawet jeśli nie jest zapisana w kodzie programu.

Dziedziczenie metod

Przykłady przedstawiane do tej pory prezentowały wyłącznie dziedziczenie pól. Java, tak jak inne języki obiektowe, umożliwiają również dziedziczenie metod. Metody zdefiniowane w klasach nadrzędnych są dostępne również w ich potomkach. Rozbudujmy klasę `Osoba` o metodę `wiek` obliczającą wiek osoby w roku przekazanym jako argument:

```
class Osoba
{
    String nazwisko;
    int rokUrodzenia;
    int wiek (int rokBiezacy)
    {
      return rokBiezacy - rokUrodzenia;
    }
}
```

Możemy ją wykorzystać do obliczenia wieku naszego pracownika w roku na przykład 2012:

```
class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p = new Pracownik();
        p.nazwisko = "Nazwisko";
        p.rokUrodzenia = 1960;
        p.pensja = 3512.0;
        System.out.println(p.wiek(2012));
    }
}
```

Metody dziedziczone z klas nadrzędnych mogą być nadpisywane w klasach podrzędnych. Nadpisywane, to znaczy zmieniane ich definicje, przy zachowaniu ich deklaracji. Ale tym tematem zajmiemy się na następnym wykładzie przy omawianiu polimorfizmu.

Konstruktory

Zmodyfikujmy klasę `Osoba` wprowadzając dwuargumentowy konstruktor:

```
class Osoba
{
    String nazwisko;
    int rokUrodzenia;
    Osoba (String n, int r)
    {
        nazwisko = n;
        rokUrodzenia = r;
    }
}
```

Taka modyfikacja powoduje, że kompilator zgłasza błędy:

```
Dziedziczenie.java:12: cannot find symbol
symbol  : constructor Osoba()
location: class Osoba
class Student extends Osoba

Dziedziczenie.java:21: cannot find symbol
symbol  : constructor Osoba()
location: class Osoba
class Pracownik extends Osoba
```

Co się stało?

Obiekty klas pochodnych składają się z dwóch części: pól odziedziczonych z klasy nadrzędnej oraz pól zadeklarowanych w danej klasie. Zatem, aby poprawnie skonstruować obiekt, należy najpierw poprawnie skonstruować pola odziedziczone, a następnie pola dodeklarowane. Ponieważ klasa nadrzędna (`Osoba`) posiada konstruktor dwuargumentowy, konstruktor domyślny nie jest generowany, a zatem musimy wykorzystać mechanizm wywołania konstruktora nadklasy z konstruktora klasy. W C++ tym mechanizmem

była lista inicjalizacyjna konstruktorów. W Javie rolę tę spełnia konstrukcja `super()`. Wszystkie klasy pochodne klasy `Osoba`, tj. `Student` i `Pracownik` muszą posiadać własne konstruktory, w których wykorzystuje się `super()`:

```
class Student extends Osoba
{
    double[] oceny;
    Student (String n, int r, int iloscOcen)
    {
        super(n,r);
        oceny = new double[iloscOcen];
    }
}

class Pracownik extends Osoba
{
    double pensja;
    Pracownik (String n, int r, double p)
    {
        super(n,r);
        pensja = p;
    }
}
```

Oczywiście taka modyfikacja klas wymaga modyfikacji ich użycia:

```
class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p = new Pracownik("Nazwisko",1960,3512.0);
    }
}
```

Przy użyciu konstrukcji `super()` należy pamiętać, że musi to być pierwsze wywołanie w ciele konstruktora, z którego wywołuje się `super()`. Jeśli konstruktor wykonuje inne instrukcje, muszą być one wykonane po wywołaniu `super()`.

Co wynika z dziedziczenia?

Budując klasy z innych klas stosując dziedziczenie tworzy się drzewo typów. W rezultacie obiekty klas pochodnych mają wiele typów—typ klasy, dla której obiekt został wygenerowany przy pomocy operatora `new`, oraz typy wszystkich klas nadrzędnych danej klasy. W naszym przykładzie, każdy obiekt reprezentujący pracownika (obiekt klasy `Pracownik`), reprezentuje też osobę (obiekt klasy `Osoba`), reprezentuje też obiekt (obiekt klasy `Object`). Co z tym się wiąże poprawne są następujące instrukcje przypisania:

```
Pracownik p1 = new Pracownik("Nazwisko",1960,3512.0);
Osoba p2 = new Pracownik("Nazwisko",1960,3512.0);
Object p3 = new Pracownik("Nazwisko",1960,3512.0);
```

Rozróżniamy tu klasy, dla których dana referencja (obiekt) została zadeklarowana od klas, dla których obiekt został wygenerowany, np: `p3` została zadeklarowana dla klasy `Object`, a wskazuje na wygenerowany obiekt klasy `Pracownik`.

Rozróżnianie klas, dla których się deklaruje od klas, dla których się generuje obiekty, choć w wielu aplikacjach wygodne, ma pewne ograniczenie. Odwołać się można tylko do tych pól i metod obiektu, które są dostępne w klasie, dla której referencję się deklaruje, a nie generuje obiekt. To znaczy, wykonania:

```
System.out.println(p1.wiek(2012));
System.out.println(p2.wiek(2012));
```

są poprawne, natomiast

```
System.out.println(p3.wiek(2012));
```

nie. Metoda `wiek()` nie jest dostępna w klasie `Object`.

Kompletny przykład

```
class Osoba
{
    String nazwisko;
    int rokUrodzenia;
    Osoba (String n, int r)
    {
        nazwisko = n;
        rokUrodzenia = r;
    }
    int wiek (int rokBiezacy)
    {
        return rokBiezacy - rokUrodzenia;
    }
}

class Student extends Osoba
{
    double[] oceny;
    Student (String n, int r, int iloscOcen)
    {
        super(n,r);
        oceny = new double[iloscOcen];
    }
}

class Pracownik extends Osoba
{
    double pensja;
    Pracownik (String n, int r, double p)
    {
        super(n,r);
        pensja = p;
    }
}

class Dziedziczenie
{
    public static void main(String[] args)
    {
        Pracownik p1 = new Pracownik("Nazwisko",1960,3512.0);
        Osoba p2 = new Pracownik("Nazwisko",1960,3512.0);
        Object p3 = new Pracownik("Nazwisko",1960,3512.0);
        System.out.println(p1.wiek(2012));
        System.out.println(p2.wiek(2012));
    }
}
```

Podsumowanie

- Dziedziczenie uzyskuje się stosując `extends`.
- Dziedziczy się pola i metody.
- Java jest językiem jednobazowym.
- Klasa `Object` jest przodkiem każdej innej klasy.
- Jeśli klasa bazowa posiada własny konstruktor, w konstruktorach klas potomnych stosuje się `super()`. `super()` musi być pierwszym wywołaniem w konstruktorze.
- Po operatorze dostępu (`.`) można podać pola i metody dostępne w klasie, dla której obiekt się deklaruje, a nie generuje.
- Wygenerowany obiekt ma wiele typów—typ klasy, dla której obiekt został wygenerowany przy pomocy operatora `new`, oraz typy wszystkich klas nadrzędnych danej klasy.