Kurs programowania

Wykład 2

Wojciech Macyna

17 marca 2016

Dziedziczenie

Klasy bazowe i potomne

Dziedziczenie jest łatwym sposobem rozwijania oprogramowania. Mając klasę bazową możemy ją uszczegółowić (dodać nowe pola i metody) nie przepisując starych komponentów, tworząc klasę potomną.

W JAVIE słowo kluczowe `extends` służy do definiowania po jakiej klasie dziedziczymy.
W C++ operator : (dwukropek).

Pojedyncze vs. wielokrotne dziedziczenie

Wiele języków obiektowych umożliwia dziedziczenie jednocześnie po kilku klasach (C++). Java ze względu na prostotę umożliwia dziedziczenie tylko po jednej klasie.

Dziedziczenie

Klasa potomna dziedziczy składowe prywatne klasy bazowej, ale nie może się do nich bezpośrednio odwołać. Wygodne jest więc użycie modyfikatora dostępu `protected` – klasy pochodne mają dostęp a inne tak jak `private`.

Publiczne metody z klasy bazowej mogą być nadpisywane w nowej klasie potomnej lub odziedziczone bezpośrednio.

Nie dziedziczy się konstruktorów i destruktora (oraz operatora = w C++).
Konstruktor klasy bazowej jest wywoływany jako pierwszy przed wywołaniem zawartości konstruktora klasy potomnej. Ale można samemu wywołać odpowiedni konstruktor klasy bazowej.
Destruktor klasy potomnej na sam koniec wywołuje destruktor klasy bazowej.

Można tylko zawężać dostęp (składowe prywatne nigdy nie będą dziedziczone jako chronione, a chronione jako publiczne).

Klasa `java.lang.Object`

Java zawiera specjalną klasę która jest korzeniem w hierarchii klas. Jest ona także domyślną klasą bazową jeśli nie podamy przy definiowaniu klasy innej.

Metody klasy `Object`

`public final Class getClass()` – zwraca nazwę klasy obiektu dla którego została wywołana.

`public int hashCode()` – zwraca wartość funkcji hashującej związanej z obiektem, przydatna np. do przechowywania obiektów w tablicach hashujących.

`public boolean equals(Object x)` – sprawdza czy dwa obiekty są tym samym obiektem.

`public String toString()`

`protected void finalize() throws Throwable`

... i jeszcze kilka związanych z wątkami.

Metody `final` nie mogą być zmieniane przez klasy potomne.

Przykład

```
1  class TestObject {
2    public int a;
3    TestObject( int i ) { a=i;}
4    public int getValue() { return a; }
5  }
6  public class ObjectTest {
7    public static void main(String[] arg) {
8      TestObject a=new TestObject(1), b=new TestObject(2);
9      System.out.println("Klasa obiektu a: "+a.getClass());
10     System.out.println("Hash obiektu a: "+a.hashCode());
11     System.out.println("Hash obiektu b: "+b.hashCode());
12     System.out.println("Porownanie a z a: "+a.equals(a));
13     System.out.println("Porownanie a z b: "+a.equals(b));
14   }
15 }
```

Własności dziedziczenia

Słowo kluczowe `super` w języku JAVA

Słowo super pełni rolę referencji do klasy bazowej. Jeśli chcemy wywołać metodę z klasy bazowej, którą nadpisaliśmy w klasie potomnej.

W C++ wypisujemy po deklaracji konstruktora wstawiamy : (dwukropek) i wywołujemy konstruktor klasy bazowej

Metoda `super` w języku JAVA

Metoda `super` wywołuje konstruktor klasy bazowej. Używa się jej najczęściej w konstruktorze klasy pochodnej.

W C++ metodę z klasy bazowej wywołujemy jak metodę statyczną (nazwa klasy i operator ::).

Przykład w języku JAVA

```
1  class Baza {
2    int a;
3    Baza() { a=0; System.out.println("Domyslny konstruktor klasy Baza"); }
4    Baza(int i) { a=i; System.out.println("Konstruktor klasy Baza" ); }
5    public void m1() { System.out.println("Baza.m1()"); }
6    public void m2() { System.out.println("Baza.m2()"); }
7  }
8  class Potomna extends Baza {
9    Potomna() { super(0); System.out.println("Domyslny konstruktor klasy Poto
10   Potomna(int i) { a=i; System.out.println("Konstruktor klasy Potomna" ); }
11   public void m3() { System.out.println("Potomna.m3()"); super.m2(); }
12   public void m2() { System.out.println("Potomna.m2()"); }
13 }
14 public class SuperTest {
15   public static void main(String[] args) {
16     Baza a = new Potomna();
17     Potomna b = new Potomna(1);
18     Object c = new Baza();
19 //     a.m3(); // blad kompilatora
20     b.m1();
21     b.m2();
22     b.m3();
23   }
24 }
```

Przykład w języku C++

```
1  #include<iostream>
2  using namespace std;
3  class Baza { public:
4    int a;
5    Baza() { a=0; cout << "Domyslny konstruktor klasy Baza" << endl; }
6    Baza(int i) { a=i; cout << "Konstruktor klasy Baza" << endl; }
7    ~Baza() { cout << "Destruktor klasy Baza" << endl; }
8    virtual void m1() { cout << "Baza.m1()" << endl; }
9    virtual void m2() { cout << "Baza.m2()" << endl; } };
10 class Potomna : public Baza { public:
11   Potomna() : Baza(0) { cout << "Domyslny konstruktor klasy Potomna" << endl
12   Potomna(int i) { a=i; cout << "Konstruktor klasy Potomna" << endl; }
13   ~Potomna() { cout << "Destruktor klasy Potomna" << endl; }
14   void m3() { cout << "Potomna.m3()" << endl; Baza::m2(); }
15   void m2() { cout << "Potomna.m2()" << endl; } };
16 int main(int argc, char* argv[]) {
17     Baza* a = new Potomna();
18     Potomna* b = new Potomna(1);
19     void* c = new Baza();
20 //    a->m3(); // blad kompilatora
21     b->m1();
22     b->m2();
23     b->m3();
24     delete a; delete b; delete (Baza*)c;
25 }
```

Użycie referencji klasy bazowej do klasy potomnej (JAVA)

Pod referencję klasy bazowej można podstawić obiekt klasy potomnej, ale można wtedy odwołać się tylko do pól i metod zdefiniowanych w klasie bazowej.

Przy podstawianiu referencji klasy bazowej wskazującej obiekt klasy potomnej pod referencję klasy potomnej musimy rzutować na typ klasy potomnej.

Słowo kluczowe `instanceof`

Konstrukcja `<referencja> instanceof <nazwa klasy>` sprawdza czy referencja wskazuje na obiekt należący do podanej klasy lub potomnej. Dzięki temu możemy prawidłowo posłużyć się rzutowaniem i wywołać odpowiednie metody mimo, ze referencja nie jest odpowiedniego typu.

Przykład

```
1  class Baza {
2    int a;
3    Baza() { a=0; System.out.println("Domyslny konstruktor klasy Baza"); }
4    Baza(int i) { a=i; System.out.println("Konstruktor klasy Baza" ); }
5    public void m1() { System.out.println("Baza.m1()"); }
6    public void m2() { System.out.println("Baza.m2()"); } }
7  class Potomna extends Baza {
8    Potomna() { super(0); System.out.println("Domyslny konstruktor klasy Poto
9    Potomna(int i) { a=i; System.out.println("Konstruktor klasy Potomna" ); }
10   public void m3() { System.out.println("Potomna.m3()"); super.m2(); }
11   public void m2() { System.out.println("Potomna.m2()"); } }
12 public class SuperTest {
13   public static void main(String[] args) {
14     Object a = new Potomna(); Object b = new Baza();
15     // a.m3(); // blad kompilatora
16     if( a instanceof Potomna ) {
17       System.out.println("a wskazuje na obiekt klasy Potomna");
18       ((Potomna) a).m3();
19     }
20     if( b instanceof Potomna )
21       System.out.println("c wskazuje na obiekt klasy Potomna");
22   }
23 }
```

Metody abstrakcyjne (wirtualne) w JAVIE – słowo `abstract`

Dzięki słowu `abstract` możemy tworzyć klasy lub metody które nie są implementowane, stanowią tylko wzorzec do implementacji. Na ich podstawie tworzymy klasy potomne lub implementujemy określone metody w klasach potomnych.

Nie można tworzyć obiektów klas abstrakcyjnych (można posługiwać się referencjami tych klas) ani wywoływać metod abstrakcyjnych. Ich implementacją zajmują się klasy pochodne.

Metody abstrakcyjne (wirtualne) w C++ – słowo `virtual`

Dzięki słowu `virtual` możemy tworzyć metody które nie są rozpoznawane na podstawie typu referencji czy wskaźnika, ale na podstawie typu obiektu (wiązanie dynamiczne).

Dzięki słowu `virtual` możemy również tworzyć metody które nie są implementowane (abstrakcyjne), stanowią tylko wzorzec do implementacji.

```
virtual metoda_abstrakcyjna()=0
```

Na ich podstawie implementujemy określone metody w klasach potomnych. Klasa zawierająca metody abstrakcyjne nazywana jest klasą abstrakcyjną.

Interfejsy w JAVIE – słowa kluczowe `interface` i `implements`

Interfejs definiujemy podobnie jak klasę (słowo `interface` zamiast `class`) ale nie ma ona pól tylko metody jakie powinna implementować klasa realizująca dany interfejs.
Interfejs jest więc zupełnie abstrakcyjną klasą bazową.

Klasa może realizować podane interfejsy jeśli podamy je przy definicji klasy po słowie kluczowym `implements`.

Klasa może dziedziczyć tylko po jednej klasie bazowej ale może implementować wiele interfejsów.

Interfejs może zawierać pola ale tylko zdefiniowane jako `static` lub `final`.

Interfejsy mogą rozszerzać inne interfejsy (nawet kilka). Korzystamy ze słowa `extends`.