Obiektowe Języki Programowania

Generyczność

Łukasz Rybka · Gdańsk 2016/17

Ideaologia

> *Generics add stability to your code by making more of your bugs detectable at compile time.*
>
> — Learning the Java Language

Słowem wstępu…

Słowem wstępu…

Wprowadzone dopiero w JDK 5.0 (2004!)

Słowem wstępu…

Wprowadzone dopiero w JDK 5.0 (2004!)

Pozwalają na wprowadzenie warstwy abstrakcji ponad typami

Słowem wstępu…

Wprowadzone dopiero w JDK 5.0 (2004!)

Pozwalają na wprowadzenie warstwy abstrakcji ponad typami

Najczęściej spotykane przy użyciu kontenerów (ale nie tylko!)

Tworzenie listy Integer'ów przed JSE 5.0

```
public class Main {
    public static void main(String[] args) {
        List myIntList = new LinkedList();
        myIntList.add(new Integer(0));

        // ...

        Integer x = (Integer) myIntList.iterator().next();❶
    }
}
```

❶ Może skutkować wyjątkiem czasu wykonania ClassCastException (unchecked)!

Tworzenie listy Integer'ów od JSE 5.0

```
public class Main {
    public static void main(String[] args) {
        List<Integer> myIntList = new LinkedList<Integer>();
        myIntList.add(new Integer(0));

        // ...

        Integer x = myIntList.iterator().next();(1)
    }
}
```

(1) - weryfikowane w czasie kompilacji

Generyczne interfejsy

```
package java.util;

public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Interpretacja generycznego interfejsu List<E>

Interpretacja generycznego interfejsu List<E>

W czasie kompilacji generuje on jedną klasę, w jednym pliku (a nie ListInteger, ListString etc.)

Interpretacja generycznego interfejsu List<E>

W czasie kompilacji generuje on jedną klasę, w jednym pliku (a nie ListInteger, ListString etc.)

E jest tzw. formalnym parametrem typu

Interpretacja generycznego interfejsu List<E>

W czasie kompilacji generuje on jedną klasę, w jednym pliku (a nie ListInteger, ListString etc.)

E jest tzw. formalnym parametrem typu

W czasie wywołania formalny parametr typu zostaje zastąpiony przez właściwy argument i dopiero wtedy ciało metody jest ewaluowane

Koncepcja nazewnicza

Koncepcja nazewnicza

Im bardziej zwięzła nazwa typu - tym lepiej (możliwie jedna litera)

Koncepcja nazewnicza

Im bardziej zwięzła nazwa typu - tym lepiej (możliwie jedna litera)

Unikamy małych liter (odróżnienie od właściwych klas czy interfejsów)

Koncepcja nazewnicza

Im bardziej zwięzła nazwa typu - tym lepiej (możliwie jedna litera)

Unikamy małych liter (odróżnienie od właściwych klas czy interfejsów)

Kontenery stosują E od "Element"

Dziedziczenie i rzutowanie w kontekście generyczności

```
public class Main {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<String>();
        List<Object> objects = strings;❶

        objects.add(new Object());

        String s = strings.get(0);
    }
}
```

❶ - ????

Dziedziczenie i rzutowanie w kontekście generyczności

```
public class Main {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<String>();
        List<Object> objects = strings;❶

        objects.add(new Object());

        String s = strings.get(0);
    }
}
```

❶ - zostanie zablokowane przez kompilator!

Ideaologia

> *In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo> is a subtype of G<Bar>.*
>
> — Learning the Java Language

Wypisanie wszystkich elementów kolekcji (#1)

```
public class Main {
    void printCollection(Collection c) {
        Iterator i = c.iterator();

        for (k = 0; k < c.size(); k++) {
            System.out.println(i.next());
        }
    }
}
```

Wypisanie wszystkich elementów kolekcji (#2)

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Wypisanie wszystkich elementów kolekcji

Wypisanie wszystkich elementów kolekcji

Wersja pierwsza działa z dowolnym typem kolekcji

Wypisanie wszystkich elementów kolekcji

Wersja pierwsza działa z dowolnym typem kolekcji

Wersja druga działa tylko z kolekcjami typu Collection<Object> (!)

Wypisanie wszystkich elementów kolekcji

Wersja pierwsza działa z dowolnym typem kolekcji

Wersja druga działa tylko z kolekcjami typu Collection<Object> (!)

Co w takim razie jest "supertypem" wszystkich kolekcji?

Wypisanie wszystkich elementów kolekcji

Wersja pierwsza działa z dowolnym typem kolekcji

Wersja druga działa tylko z kolekcjami typu Collection<Object> (!)

Co w takim razie jest "supertypem" wszystkich kolekcji?

Collection<?> - "kolekcja nieznanego typu" (wildcard)

Wypisanie wszystkich elementów kolekcji (#3)

```
void printCollection(Collection<?> c) {
    for (Object e : c) {❶
        System.out.println(e);
    }
}
```

❶ - bezpieczne ponieważ wszystkie klasy dziedziczą po klasie Object

Wypisanie wszystkich elementów kolekcji (#3)

```
public class Main {
    public static void main(String[] args) {
        Collection<?> c = new ArrayList<String>();

        c.add(new Object());(1)
    }
}
```

(1) - **błąd kompilacji!** - dlaczego?

Przykładowa hierarchia dziedziczenia

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
        ...
    }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
        ...
    }
}
```

Klasa Canvas

```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }

    public void drawAll(List<Shape> shapes) {❶
        for (Shape s: shapes) {
            s.draw(this);
        }
    }
}
```

❶ - może zostać wywołana jedynie z listą typu Shape

Klasa Canvas

```
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }

    public void drawAll(List<? extends Shape> shapes) {
        for (Shape s: shapes) {❶
            s.draw(this);
        }
    }
}
```

❶ - bezpieczne ponieważ wiemy, że przekazany typ będzie potomkiem Shape

Bounded wildcard

Bounded wildcard

Wyrażenie "<? extends X>" oznacza, że nie znamy dokładnego typu, lecz wiemy, że typ ten jest potomkiem X

Bounded wildcard

Wyrażenie "<? extends X>" oznacza, że nie znamy dokładnego typu, lecz wiemy, że typ ten jest potomkiem X

Typ X jest nazywany upper bound of the wildcard

Bounded wildcard - pułapka

```
public class Main {
    public void addRectangle(List<? extends Shape> shapes) {
        shapes.add(0, new Rectangle());(1)
    }
}
```

(1) - **błąd kompilacji** - ponieważ nie znamy ostatecznego typu ?

Metoda generyczna - podejście pierwsze

```
public class Main {
    static void fromArrayToCollection(Object[] a, Collection<?> c) {
        for (Object o : a) {
            c.add(o);❶
        }
    }
}
```

❶ - błąd kompilacji

Metoda generyczna - podejście drugie

```
public class Main {
    static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
        for (T o : a) {
            c.add(o);
        }
    }
}
```

Metoda generyczna - przykład użycia

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();

String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();

Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
```

```
fromArrayToCollection(oa, co);1
fromArrayToCollection(sa, cs);2
fromArrayToCollection(sa, co);3
fromArrayToCollection(ia, cn);4
fromArrayToCollection(fa, cn);5
fromArrayToCollection(na, cn);6
fromArrayToCollection(na, co);7
fromArrayToCollection(na, cs);8
```

Generyczna klasa Collections

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) {(1)
        // ...
    }

    public static <T, S extends T> void copy(List<T> dest, List<S> src) {(2)
        // ...
    }
}
```

(1) - wersja z użyciem wildcard

(2) - wersja bez użycia wildcard

ANY QUESTIONS?