Generyki

Klasy generyczne są wielką zaletą obiektowych języków programowania. Pozwalają na utworzenie uniwersalnych klas do przechowywania dowolnych typów danych. Nie ma potrzeby pisania wielu klas (np. kolekcji) dla każdej oddzielnej struktury. Wystarczy odpowiednio sparametryzować taką kolekcję podczas inicjalizacji, aby Java mogła zrozumieć jakiego typu elementy może w nim trzymać.

Ogólnie

Zobaczmy najpierw prosty przykład:

```
...
ArrayList<String> listaString = new ArrayList<String>();
listString.add("Ala");
listString.add("ma");
listString.add(new Integer(1)); // błąd kompilacji. Nie jest to
Integer
listString.remove(0); // usuń 1-wszy element
String element = listString.get(0); // pobranie pierszego elementu
System.out.println(element); // wypisanie elementu
System.out.println(listString); // wypisanie zawartości listy
...
```

W pierwszej linijce od Java 7 możemy napisać tak:

```
ArrayList<String> listaString = new ArrayList<>();
```

jest to zapis za pomocą diamentu. Java widzi, że jest to ArrayList typu String i nie trzeba powtarzać tego po prawej stronie. Dodatkowo aby uniknąć silnego typowania (w tym przypadku ArrayList) programiści często korzystają z abstrakcji i piszą po prostu tak:

```
List<String> listaString = new ArrayList<>();
```

co jest prawdą bo jak wejdziemy do klasy ArrayList to zobaczymy, że implementuje interfejs List. Dzięki temu nie przekazujemy w kodzie z jakiej konkretnej klasy skorzystaliśmy - niech innym programistom wystarczy informacja, że jest to Lista (zasada ukrywania implementacji z dobrych rad).

Szczegóły

Generyki tworzymy za pomocą labelów którymi są napisy pisane z dużych liter. Bardzo często korzysta się z litery T. Np. prosta własna klasa opakowująca dowolny inny typ.

```
public class Box<T> {
```

```
        private T item

        private Box(T item) {
                this.item = item;
        }

        public L getItem() {
                return item;
        }
        public void setItem(L item) {
                this.item = item;
        }

        public static void main(String[] args) {
                Box<String> box = new Box<>("AAA");
                System.out.println(box.getItem());
                Box<Integer> box2 = new Box<>(2);
                System.out.println(box2.getItem());
        }
}
```

Albo z dwoma typami generycznymi:

```
public class Pair<L,R> {

        private L left;
        private R right;

        protected Pair(L left, R right) {
                this.left = left;
                this.right = right;
        }
        public L getLeft() {
                return left;
        }
        public void setLeft(L left) {
                this.left = left;
        }
        public R getRight() {
                return right;
        }
        public void setRight(R right) {
                this.right = right;
        }

        public static <L,R> Pair<L,R> of(L left, R right){
                return new Pair<L,R>(left, right);
        }

        public static void main(String[] args) {
```

```
                // Pair<String, Integer> para = new Pair<>("AAA",
1); - konstruktor zrobiłem protected. Poza klasą Pair nie można
tego zrobić.
                Pair<String, Integer> pair2 = Pair.of("AAA", 1);
                String text = pair2.getLeft();
        }
}
```

Drugi przykład pokazuje dodatkowo, że można tworzyć metody statyczne z typami generycznymi. Jako, że metody statyczne są oderwane od swoich klas to trzeba metodę tą poinformować, że argumenty metody nie są klasami tylko generykami:

```
public static <L,R> Pair<L,R> of(L left, R right){
    return new Pair<L,R>(left, right);
}
```

Oznaczenie: <L,R> informuje kompilator, że są to typy generyczne więc potem możemy napisać metodę, która zwraca obiekt typu Pair<L,R> oraz przyjmuje argumenty typu R i L czymkolwiek one by nie były. W tym przypadku R i L mogą być dowolnymi obiektami z wyjątkiem typów prostych. Identyczny zabieg można również zrobić nawet w zwykłych metodach klas niegenerycznych - wystarczy tylko w nawiasach <R> poinformować kompilator, że w argumentach metod będą typy generyczne.

Co warto wiedzieć?

Jeśli w Javie jest tak, że klasa Float, Double czy Integer dziedziczą po Number i można napisać tak:

```
public static void addNumber(Number number) {
    // tutaj jakiś kod
}

// potem gdzieś w main:
Integer int1 = new Integer(1); // lub po prostu: Integer int1 = 1;
Number flo1 = new Float(4);

addNumber(int1); // ok. Integer jest Number
addNumber(flo1); // ok.
```

to w przypadku generyków nie można w ten sposób rozumieć, że List<Integer> dziedziczy po List<Number>. Najbliższy rodzic tych dwóch typów to Object. Jedyne dziedziczenie, które Java rozumie to te, które my sami zdefiniujemy jako extends. A więc wzorując się, na klasie Pair stwórzmy klasę Triple:

```
public class Triple<L,M,R> extends Pair<L,R> {

 private M middle;
```

```
 protected Triple(L left, M middle, R right){
    super(left, right);
    this.middle = middle;
 }

 public M getMiddle(){
   return middle;
 }

 public void setMiddle(M middle){
   this.middle = middle;
 }

 public static void main(String[] args) {
    // a teraz zapisamy pasy! Korzystam z abstrakcji:
    Pair<String, String> pair = new Triple<String, Integer,
String>("AAA", 1, "BBB");
    System.out.println(pair.getLeft());
 }
}
```

Kolekcje

O kolekcjach zostało napisane tyle w Internecie więc aby nie odkrywać koła na nowo odsyłam do wpisu na blogu (szczegóły w zadaniu 3.).

Ograniczanie swobody typów

Czasem stosowanie typu T jest dość kłopotliwe. O takim obiekcie nic nie wiemy więc nie jesteśmy w stanie wykonać żadnej jego metody. Wiemy tylko tyle, że dziedziczy po Object więc tylko te metody na pewno posiada. Czasem trzeba ograniczyć swobodę typu aby móc z takim obiektem wykonać. Przykładowo:

```
import java.util.*;

public class NumberBox<T extends Number> {

    List<T> list = new ArrayList<>();
    double sum = 0;

    public void addToBox(T t){
      sum+=t.doubleValue();
      list.add(t);
    }
}

// gdzieś w main:
```

```
NumberBox<Integer> box = new NumberBox();
box.addToBox(1);
```

Skoro mam nałożone jakieś ograniczenia na T to już mogę z abstrakcji, że posiada już jakieś metody które to ograniczenie na nim wymusza. Dzięki temu mogę sumować wartości elementów w trakcie ich dodawania do NumberBox. Należy tutaj mieć na względzie, że zapis T extends Number oznacza, że pasują tutaj wszystkie klasy dziedziczące po Number oraz zakłada się, że klasa Number sama po sobie też dziedziczy a więc też może być tutaj zaaplikowana.
Ograniczenie takie nie musi być klasą. Może być one interfejsem. W takim przypadku nie stosuje się słowa implements a nadal extends. Mogą być to nawet zespoły ograniczeń typu:

```
import java.util.*;

public class NumberBox<T extends Number & Printable & Valuable> {
    // ...
}
```

Z czego trzeba mieć na względzie, że jako pierwsza musi być użyta klasa a następnie można wymieniać interfejsy (jak w przykładzie).
Inne ograniczenia
Nie tylko extends można zastosować do ograniczania typu. Można zastosować rozumienie odwrotne w postaci operatora super. Super oznacza wszystkie nadtypy (np prawda dla: Number super Integer albo: Object super String). Mało tego, zamiast T można zastosować po prostu znak ? który oznacza dowolny typ - najczęściej stosujemy je w metodach kiedy nie chce nam się pisać w nazwie metody <T> by poinformować o tym, że T jest typem generycznym. W przykładzie:

```
public void printList(List<? extends Number> numbers) {  // użycie
? z extends
    for(Number number: numbers){
        System.out.println(number);
    }
}

//zamiast:
public <T> void printList(List<T extends Number> numbers) {
    for(Number number: numbers){
        System.out.println(number);
    }
}

public void addToList(List<? super Number> numbers) {  // użycie ?
z super
      numbers.add(new Integer(1));
      numbers.add(new Double(1));
}
```

Niby wszystko jasne. Ale czy na pewno? I teraz uwaga kilka rzeczy, które trzeba sobie uzmysłowić:

```
// niech:
public class Person {...}
// oraz:
public class VipPerson extends Person { ... }
// oraz:
public class SuperPerson extends VipPerson { ... }

// i teraz:

// przykład 1:
void addPersonToList(List<Person> list) {
        list.add(new Person());                 // ok
        list.add(new VipPerson());      // ok. VipPerson jest
Person
}

// przykład 2:
void addAnyToList(List list) {           // List bez typu =
List<Object>
        list.add(new Person());                 // ok
        list.add(new VipPerson());      // ok
        list.add(new Integer(1));       // ok :)
}

// przykład 3:
void printList(List list) { // dowolna lista
        System.out.println(list.toString());
}

// przykład 4:
void printFromList(List<? extends Person> list) {
        Person person = list.get(0);
        System.out.println(person);
}

// przykład 5:
void addPersonToList2(List<? extends Person> list) {
        list.add(new Person());// błąd bo nie wiadomo czy to lista
VipPerson, Person,
                                // czy jakaś inna klasa
dziedzicząca po Person :)
                                // gdyby to była List<SuperPerson>
to nie mogę przecież dodać Person
                                // bo Person nie jest SuperPerson.
To SuperPerson jest Person.
        list.add(new VipPerson());      // też błąd
}
```

```
// przykład 6:
void addPersonToList3(List<? super VipPerson> list) {
        list.add(new Person());         // błąd! Co jeśli jest to
List<VipPerson>? Wtedy nie można dodać Person
        list.add(new VipPerson());      // ok
        list.add(new SuperPerson());    // ok
        list.add(new Object());                 // błąd!
        Object object = list.get(0);    // typu Object. Nie wiemy
jakiego typu jest obiekt w liście.
                                        // Może VipPerson, może
Person... kto wie!? Jedyna prawda to Object.
}
```