Definicja klasy generycznej

Przyjrzyjmy się następującemu kodowi:

```
List<Integer> ints = Arrays.asList(1, 2, 3);
int sum = 0;
for (int n : ints) {
    sum += n;
}
System.out.println(sum);
```

Intuicyjnie wiemy że chodzi o to, że w liście znajdują się obiekty typu Integer. Jeśli jednak spojrzymy w definicję interfejsu List, zobaczymy:

```
public interface List<E> extends Collection<E>
```

w takim przypadku mówimy o liście typu generycznego, typie E. W momencie kiedy w definicji funkcji lub klasy użyjemy nawiasów, oznaczamy ją jako generyczną, to znaczy dowolnego typu który zostanie wstawiony w miejsce znaku E.

przed wprowadzeniem generyków w javie 5, powyższy przykład wyglądał następująco:

```
List ints = Arrays.asList(new Integer[]{new Integer(1), new Integer(2), new Integer(3
)});
int sum = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
    int n = ((Integer) it.next()).intValue();
    sum += n;
}
System.out.println(sum);
```

Weźmy pod uwagę inny przykład:

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
System.out.println(s.equals("Hello world!"));
```

oraz

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1));
System.out.println(s.equals("Hello world!"))
```

Jak widać można używać kolekcji bez podania typu generycznego, jednakże przed użyciem typu właściwego dla przechowywanego obiektu potrzeba wykonać rzutowanie wprost, explicite. Jeśli przyjrzymy się wynikowi kompilacji dwóch ostatnich bloków, zauważymy że instrukcje bytecode są identyczne, a w czasie działania programu (runtime) typ jest reprezentowany przez po prostu List.

```
List<String> list = new ArrayList<>();
System.out.println(list.getClass().getName())
```

Aby stworzyć klasę generyczną, wzbogacamy ją o *formal type parameter*, który umieszczamy w ostrych nawiasach, tzw. *diamond operator* (czyli wspomniane wcześniej E)

```
import java.util.Objects;

public class Crate <T> {
    private T content;

    public T emptyCrate() {
        T toReturn = content;
        content = null;
        return toReturn;
    }

    public void packCrate(T contents) {
        if(Objects.isNull(this.content)){
            this.contents = content;
        } else {
            System.err.println("cannot pack if something inside");
        }
    }
}
```

konwencja nazewnicza typów generycznych:

- E - element (to oznaczenie wykorzystywane jest najczęściej we frameworku Collections)
- K - klucz
- N - liczba
- T - typ
- V - wartość
- S, U, V - dla następnych typów

jednakże nic nie stoi na przeszkodzie, by móc korzystać ze swoich nazw, jeżeli będą bardziej sugestywne aniżeli E,K,V itd.

typ generyczny podajemy podczas tworzenia obiektu, dzięki czemu wiadomo jakie typy mają być używane do operacji na danej klasie, jednakże reguły te obowiązują przed kompilacją, ponieważ później następuje tzw. *type erasure*, czyli wszystkie typy generyczne zostają sprowadzone do klasy Object, wobec czego powyższa klasa po kompilacji wygląda następująco:

```
public class Crate {
    private Object content;

    public Object emptyCrate() {
        Object toReturn = content;
        content = null;
        return toReturn;
    }

    public void packCrate(Object contents) {
        this.contents = content;
    }
}
```

Typami generycznymi mogą być tylko typu referencyjnego (obiektowego), dlatego bardzo przydatny jest mechanizm boxingoraz unboxing typu prymitywnego na referencyjny i odwrotnie. Przypomnijmy sobie typy prymitywne i odpowiadające im typy referencyjne

Primitive	**Reference**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Poniższe zapisy są sobie równe

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
int n = ints.get(0);
```

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Co więcej, parametrami typu generycznego mogą być inne typy generyczne! np.

```
List<List<String>> listOfListOfStrings = new ArrayList<>();
ArrayList<String> listInsideOfList = new ArrayList<>();
listInsideOfList.add("first!");
listOfListOfStrings.add(listInsideOfList);

System.out.println(listOfListOfStrings.get(0).get(0)); //<-- "first!
```

tworzenie obiektu generycznego:

```
Crate<Zebra> zebraCrate = new Crate<>(); //lub new Crate<Zebra>();
zebraCrate.packCrate(new Zebra());
Zebra zebra = emptyCrate();
```

Ograniczanie deklarowanego typu generycznego

Przy pracy z typami generycznymi mamy dostęp również do mechanizmu ograniczenia typu generycznego:

```
abstract class Animal{
    abstract void roar();
}

class AnimalWrapper<T extends Animal>{

    T animal;

    public AnimalWrapper(T animal) {
        this.animal = animal;
    }

    public void animalRoar(){
        animal.roar();
    }

}
```

jeśli wiemy że możemy umieścić jako typ generyczny wyłącznie klasy które dziedziczą po klasie Animal... będziemy mieć pewność że mają dostęp do tych metod które są tam zdefiniowane .

Realizujemy to poprzez dodanie słówka extends w typie generycznym.

Teraz obiekty typu T maja dostęp do metod klasy Animal (domyślnie tylko do metod klasy Object)

generyczne interfejsy

Podobnie jak w przypadku klas, również interfejsy mogą być typu generycznego

```
public interface Shippable < T > {
    void ship(T t);
}
```

wówczas, gdy chcemy zaimplementować dany interfejs, podczas wykonywania instrukcji implements podajemy docelowy typ, np.

```
class ShippableZebraCrate implements Shippable <Zebra> {

    public void ship(Zebra t){
        ...
    }
}
```

użycie:

```
public static void main(String[] args){
  ShippableZebraCrate zebraCrate = new ShippableZebraCrate();
  zebraCrate.ship(new Zebra()); //nie przyjmie niczego poza zebrą... lub obiektem
dziedziczącym po Zebra
}
```

lub zachować generyczność tego rozwiązania, wiążąc to z typem generycznym powiązanym z klasą implementującą:

```
class ShippableGenericCrate <U> implements Shippable <U> {

    public void ship(U t){
        ...
    }
}
```

użycie:

```
public static void main(String[] args){
  ShippableGenericCrate<Zebra> zebraCrate = new ShippableGenericCrate();
  zebraCrate.ship(new Zebra());
}
```

Kiedy używamy generyków ?

np. jeśli nie chcemy się martwić czy ktoś nie wrzucił do listy czegoś czego się nie spodziewamy.

tak wygląda sytuacja w której korzystamy z ogólnego typu List (de facto List<Object>) - tutaj trzeba się martwić :D

```
static void printNames(List list) {
    for (int i = 0; i < list.size(); i++) {
        String name = (String) list.get(i); // class cast exception here
        System.out.println(name);
    }
}

public static void main(String[] args) {
    List names = new ArrayList();
    names.add(new StringBuilder("Webby"));
    printNames(names);
}
```

natomiast jeśli użyjemy typów generycznych...

```
List<String> names = new ArrayList<String>();
names.add(new StringBuilder("Webby")); // DOES NOT COMPILE
```

Czy w danej klasie może być tylko jeden typ generyczny ?

Nie, klasa może zawierać 0 lub więcej typów generycznych :)

```
public class TwoPartsCrate<T, U> {
    private T first;
    private U second;

    public SizeLimitedCrate(T firstContent, U secondContent) {
        this.first = firstContent;
        this.second = secondContent;
    }
}
```

Metody generyczne

możliwe jest utworzenie metody typu generycznego (nie musimy deklarować wówczas poziomu generycznego na poziomie klasy)

```
public static <T> Crate<T> ship(T t) {
    System.out.println("Preparing " + t);
    return new Crate<T>();
}
```

przykłady metod:

```
public static <T> void sink(T t) { }
public static <T> T identity(T t) { return t; }
public static T noGood(T t) { return t; } // DOES NOT COMPILE, brak definicji typu w
ramach <>
```

wywołania metod generycznych

Metody generyczne możemy wywołać tak jak normalne metody, kompilator może się wówczas domyślić zwracanego typu np. na podstawie parametru (jeśli argument typu generycznego znajduje się na liście parametrów). Możemy również podać typ wprost, umieszczając go w nawiasach klamrowych **przed** wywołaniem funkcji, np. dla metody ship oba poniższe wywołania są poprawne:

```
public class Box {

   public static <T> T ship(T input){
       return input;
   }

    public static void main(String[] args) {
        String stringBox = Box.<String>ship("package");
        String[] stringArrayBox = Box.<String[]>ship(args);

        String stringBox2 = Box.ship("package");
        String[] stringArayBox2 = Box.ship(args);
    }
}
```

Przykład metody wykorzystującej ograniczenie typów:

funkcja konwertująca na inny typ:

```
public static <T, R extends T> T convert(Class<T> clazz, R toConvert){
    return clazz.cast(toConvert);
}

Number number = convert(Number.class, 5);
System.out.println(number); // 5
```

Czego typy generyczne nie potrafią ?

Większość ograniczeń dotyczących typów generycznych (type parameter), wynika z ich właściwości *type erasure*, tj. wymazywania typu w trakcie kompilacji, i zastępowanie go typem Object. W związku z tym:

- nie można wywoływać konstruktora - wynika to z faktu, że w czasie uruchomienia programu (*runtime*), jest on zamieniony na konstruktor klasy Object (new Object()), a co za tym idzie jedyny konstruktor jest bezargumentowy, jeśli klasa będzie posiadać inny konstruktor, konstrukcja nie zadziała
- tworzyć tablicy typu T[] - w przeciwieństwie do typów generycznych, tablica przechowuje informację na temat przechowywanego typu w trakcie działania programu, a w momencie wymazywania typów po kompilacji. Jako że nie wiemy jakiego typu jest T po kompilacji, nie możemy stworzyć jej instancji. (dyskusja na ten temat tutaj: https://stackoverflow.com/questions/2927391/whats-the-reason-i-cant-create-generic-array-types-in-java) są obejścia, np.

```
T t; //zakładając że obiekt jest w jakiś sposób zainicjalizowany lub przekazany jako
parametr
int length = 5; //przykladowo, rozwniez jako np. parametr
T[] ts = (T[]) Array.newInstance(t.getClass(), length); //rozwiazanie wykorzystujace
mechanizm refleksji
```

```
public class Box<S> {

    S[] ts;

    public Box(S first, S ... objects){
        // this.ts = new S[objects.length+1]; //tak nie mozemy
        this.ts = (S[]) Array.newInstance(first.getClass(), objects.length > 0 ?
objects.length+1 : 1);
        //rozwiazanie wykorzystujace mechanizm refleksji, uwaga ze strony kompilatora
        ts[0] = first;
        for(int i=0; i<objects.length; i++){
            ts[i+1] = objects[i];
        }
    }

    public void printAll(){
        for (S t : ts) {
            System.out.println(t);
        }
    }

    public static void main(String[] args) {
        Box<String> box = new Box<>("first", "second", "third", "fourth");
        box.printAll();
    }
}
```

- wykorzystywać mechanizmu instanceof - skoro typ jest wymazywany, to List<String> i List<Integer> są z punktu widzenia programu identyczne (obj instanceof Object) ←- zawsze prawda
- używać typów prymitywnych jako typu generycznego - co nie jest dużym problemem, biorąc pod uwagę mechanizm autoboxingu typów prymitywnych
- tworzyć list obiektów typu genertycznego(np. new List<String>[3] jest niepoprawne), dopóki nie użyjemy unbounded wildcard (o tym za chwilę, new List<?>[3] jest poprawne)
- definiowania zmiennych statycznych jako pól klasy - niemożliwe ponieważ są one powiązane z klasą a nie konkretnym obiektem, a określenie typu zmiennej możliwe jest podczas tworzenia obiektu

ostatni punkt; gdyby było możliwe

```
public class MobileDevice<T> {
    private static T os;
}
```

wówczas jakiego typu byłaby zmienna os:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();

MobileDevice.os; // ??? jaki typ ? pewnie najlepszy bylby Object ;)
```

niebezpieczny (acz kompilowalny!) kod

```
public class LegacyUnicorns {
    public static void main(String[] args) {
        java.util.List<Unicorn> unicorns = new java.util.ArrayList<>();
        addUnicorn(unicorns);
        Unicorn unicorn = unicorns.get(0); // ClassCastException
    }

    private static void addUnicorn(List unicorn) {
        unicorn.add(new Dragon());
    }
}
```

Granice (bounds)

Typy generyczne oprócz definiowania symboli oznaczających poszczególne typy, posiadają również możliwość określenia zakresów typów. *Wildcard*, to tzw. maska / zaślepka, określająca wyrażenie ograniczające daną wartość

Bounded parameter type to specjalny rodzaj generyka.

Wildcard generic type, to nieznany rodzaj typu generycznego, określany przez typ < *?* >. Używamy go wówczas, kiedy każdy typ jest dla nas "OK"

Typ ograniczenia	Składnia	Przykład
Unbounded Wildcard (dowolny typ)	?	List<?> l = new ArrayList<String>();
Wildcard ograniczona odgórnie (upper bound)	? extends type	List<? extends Exception> l = new ArrayList<RuntimeException>();
Wildcard ograniczona oddolnie (lower bound)	? super type	List<? super Exception> l = new ArrayList<Object>();

Granice dotyczą tylko typów referencyjnych, a nie samych obiektów!

Unbounded Wildcards

Zwróćmy uwagę na następujący kod:

```
    public static void printList(List<Object> list) {
        for (Object x : list){
            System.out.println(x);
        }
    }

    public static void main(String[] args) {
        List<String> keywords = new ArrayList<>();
        keywords.add("java");
        printList(keywords); // DOES NOT COMPILE
    }
```

co tu jest nie tak ? Przecież String jest podklasą Object, więc jeśli metoda przyjmuje List typu sparametryzowanego Object, to List<String> nie spełnia tego warunku ? No niestety nie... nie możemy przypisać listy typu List<String> do referencji typu List<Object>. Tak, to brzmi nielogicznie. Więc dlaczego ? Spójrzmy na inny kod:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(new Integer(42));

List<Object> objects = numbers; // DOES NOT COMPILE
objects.add("forty two");

System.out.println(numbers.get(1));
```

Jak widzimy przypisanie listy List<Integer> do typu List<Object> mogłoby się źle dla nas skończyć. Kompilator obiecał nam że do listy numbers będziemy mogli dodawać tylko obiekty typu Integer. Gdybyśmy przypisali tą listę do referencji typu List<Object> wówczas moglibyśmy dodawać dowolne obiekty dziedziczące po Object, czyli obietnica kompilatora poszła by... :) Jak więc możemy rozwiązać nasz problem ? Możemy powiedzieć że to czego tak naprawdę potrzebujemy, to nie lista obiektów typu Object, a raczej lista obiektów typu dowolnego List<?>. Wówczas każdy element będzie traktowany jak Object:

```
public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords);

    List<Integer> integers = new ArrayList<>();
    integers.add(40);
    printList(integers);
}

public static void printList(List<?> list) {
    for (Object x: list){
        System.out.println(x);
    }
    //list.add("pies"); tej operacji nie wykonamy, ogranicza nas wildcard ?
}

public static void printListOfObjects(List<Object> list) { //przypominam, referencja
to "sposob patrzenia na obiekt"
    for (Object x: list){
        System.out.println(x);
    }
}
```

Upper-Bounded Wildcards

Jak ustaliśmy wyżej, poniższy przykład się nie skompiluje:

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
```

Jako że byłoby to sytuacją naruszenia kontraktu, w którym do listy deklarującej trzymanie obiektów typu Integer, można by dodać np. obiekt typu Double

Zamiast tego, musimy użyć następującej kontrukcji:

```
List<? extends Number> list = new ArrayList<Integer>();
```

Oznacza to, że do referencji List<? extends Number> możemy przypisać dowolną kolekcje, której elementy będą typu dziedziczącego po Number. Możemy to zapisać również następująco:

```
List<Number> list = new ArrayList<Number>();
```

stracimy jednak wówczas możliwość umieszczania w tej liście wyłącznie obiektów typu Integer (a może nam na tym zależeć).

Mając do dyspozycji typ List<? extends Number> możemy go użyć w definicji metody, sumującej wartości wszystkich obiektów typów liczbowych (typów dziedziczących po Number), tj. niezależnie czy przekażemy listę typu List<Integer>, List<Double> czy nawet List<Number>, program będzie działać poprawnie:

```
public static long total(List<? extends Number> list) {
    long count = 0;
    for (Number number: list)
        count += number.longValue();
    return count;
}
```

na powyższy kod nadal działa zasada *type erasure*, w związku z tym gdybyśmy chcieli uzyskać tą funkcjonalność bez typów generycznych, wyglądałoby to następująco:

```
public static long total(List list) {
    long count = 0;
    for (Object obj: list) {
        Number number = (Number) obj;
        count += number.longValue();
    }
    return count;
}
```

it's a trap!

W sytuacji gdy używamy referencji typu generycznego używającego *unbounded wildcard* lub *upper_bound*, obiekty powiązane z tym typem referencji stają się automatycznie praktycznie niemodyfikowalne (tzw. **immutable**). Właściwość ta oznacza, że obiekt nie może być modyfikowany (no akurat w tym przypadku można usuwać, ale np. dodawać już nie).

```
abstract class Bird{};
class Hawk extends Bird {};
class Duck extends Bird {};

public static void main(String args[]){
    List<? extends Bird> birds = new ArrayList<Bird>(); //OK
    birds.add(new Hawk()); //nie kompiluje sie!
    birds.add(new Duck()); //nie kompiluje sie!
}
```

Dlaczego ?

z racji że działamy na typie referencji, czyli List<? extends Bird>, nie wiemy czy lista będzie typu Hawk, Duck czy jeszcze innego typu który jeszcze nie powstał.

W pierwszym przypadku - `birds.add(new Hawk())` - nie możemy dodać obiektu typu Hawk do listy

która może być potencjalnie typu List<Duck>,

Analogicznie w przypadku `birds.add(new Duck())` - nie możemy dodać obiektu typu Duck do listy, która może być potencjalnie typu List<Hawk>

Z punktu widzenia javy, powyższe scenariusze są jak najbardziej logiczne (□■_■)

Lower-Bounded Wildcards

Zadanie jest następujące: zaimplementuj metodę addCar(...), która dodaje do list typu Object oraz String, obiekt typu String "car";

```
List <String> strings = new ArrayList <String>();
strings.add("motorbike");
List <Object> objects = new ArrayList <Object>(strings);

addCar(strings);
addCar(objects);
```

Aby rozwiązać ten problem, używamy mechanizmu *lower bound*

```
public static void addSound(List<? super String> list){
    list.add("car");
}
```

Dlaczego to działa ? używając *lower bound* mówimy javie, że parametrem metody danego typu może być lista obiektów typu String, lub każdej klasy będącej nadklasą typu String. (a jeśli jest nadklasą Stringa, to możemy tego Stringa dodać, bo String JEST zgodnie z zasadą polimorfizmu równocześnie każdym typem ponad nim)

it's a trap, part2

```
1: List <? super IOException > exceptions = new ArrayList <Exception>();
2: exceptions.add(new Exception()); // DOES NOT COMPILE
3: exceptions.add(new IOException());
4: exceptions.add(new FileNotFoundException());
```

Definiując w pierwszej linii typ List<? super IOException>, mówimy javie, że do zmiennej tego typu możemy przypisać następujące kolekcje: List<IOException>, List<Exception>, List<Object>.

Linia nr. 2 nie kompiluje się, ponieważ może dojść do sytuacji, w której dodajemy obiekt typu Exception do listy typu List<IOException>.

Linia nr. 3 jest ok, ponieważ niezależnie od typu kolekcji, IOException spełnia zasadę IS-A względem typu przechowywanego w liście (IOException instanceof IOException, IOException instanceof Exception, IOException instanceof Object)

Linia nr. 4 jest ok, mimo że ogranicznik typu mówi o nadklasie, to mimo to FileNotFoundException jako podklasa IOException, spełnia warunki IS-A podobnie do obiektu typu IOException

Ponadto, warto wspomnieć że wildCards, mogą być używane tylko w przypadku typów referencyjnych, niemożliwym jest następująca konstrukcja, ponieważ ostateczny typ musi być znany podczas tworzenia obiektu:

```
List <?> list6 = new ArrayList <? extends A>();
```

Również poniższy kod jest nieprawidłowy; *wildcards* zawsze musi być używany w połączeniu z → **?**

```
<X> void method5(List <X super B> list) { } // DOES NOT COMPILE
```

Typy generyczne są skomplikowane (zaskakujący wniosek, nieprawdaż ?)

Nie zostały one dodane do Javy od samego początku. W związku z tym, że twórcy chcieli zachować kompatybilność wstecz istnieje wiele kruczków, które nie są trywialne.