# Notatki - Typy generyczne

## Spis treści



W tej części notatek opowiemy o tym czym są **Generic Types**, jakim celu się je stosuje i jakie są dostępne warianty.

## Po co są typy generyczne?

Wyobraź sobie, że mamy taki przypadek, że chcemy żeby z klasy którą napisaliśmy mogły korzystać tylko określone typy. Przykładowo, wyobraź sobie, że tworzymy klasę `Bag` i chcemy w niej przechowywać tylko Stringi. Częściej spotkasz się z przykładem, w którym będziemy mieli listę i w niej mamy przechowywać np. tylko Integery.

Spójrzmy na przykład, kod poniżej skompiluje się poprawnie, ale spróbuj go uruchomić ☺.

```
public class Example {
    static void printCities(List cities) {
        for (int i = 0; i < cities.size(); i++) {
            String city = (String) cities.get(i);
            System.out.println(city);
        }
    }

    public static void main(String[] args) {
        List cities = new ArrayList();
        cities.add(1020);
        printCities(cities);
    }
}
```

Kod w trakcie działania wyrzuca `ClassCastException`. Dlaczego? Bo Java nie jest w stanie rzutować Integera na String. Z drugiej strony zdefiniowaliśmy Listę, w której można mieć cokolwiek, Integery, Stringi, Gruszki i Samochody. Natomiast w metodzie `printCities()` określamy, że każdy element, próbujemy rzutować na Stringa. Czy da się rzutować Gruszkę na Stringa? No nie da się, Integera też nie, stąd błąd.

Typy generyczne rozwiązują ten problem. Dzięki nim programista jest w stanie wymusić, żeby w danej klasie można było użyć tylko określonego typu. Czyli klasa `Bag` mogłaby przechowywać tylko Gruszki. A klasa `List` tylko krzesła. Wtedy taki przykład kodu wyglądałby w ten sposób:

```
public class Example {
    static void printCities(List<String> cities) {
        for (int i = 0; i < cities.size(); i++) {
            // W ten sposób nie musimy tutaj rzutować na typ String
            String city = cities.get(i);
            System.out.println(city);
        }
    }

    public static void main(String[] args) {
        List<String> cities = new ArrayList<>();
        // W tej linijce dostaniemy błąd kompilacji
        // dzięki wykorzystaniu typów generycznych
        cities.add(1020);
        cities.add("Wrocław");
        printCities(cities);
    }
}
```

# Własna klasa z typem generycznym

```
public class Bag<T> {

    private T element;

    public void pack(T element) {
        this.element = element;
    }

    public T empty() {
        // teoretycznie ta metoda mogłaby zwracać void,
        // a w implementacji tylko nullować element
        T tmp = element;
        this.element = null;
        return tmp;
    }
}
```

W przykładzie powyżej mamy klasę `Bag`, w której określamy, że będziemy korzystać z typu generycznego `T`. Jeżeli chcielibyśmy jej użyć to można napisać to tak:

```
public class BagCaller {
```

```
    public static void main(String[] args) {
        // W tej linijce określamy, że chcemy żeby typem T był String
        Bag<String> stringBag = new Bag<>();
        stringBag.pack("String element");
        String element = stringBag.empty();
        System.out.println(element);

        Bag<Car> carBag = new Bag<>();
        carBag.pack(new Car());
        Car car = carBag.empty();
        System.out.println(car);
    }
}
```

Tutaj dodajmy 2 dygresje:

## Pierwsza dygresja

Kiedyś definiowanie generyków wyglądało w ten sposób (zwróć uwagę na String w generyku w wywołaniu konstruktora Bag):

```
Bag<String> stringBag = new Bag<String>();
```

Po Javie 7 nie ma już takiej konieczności i możemy to zapisywać w ten sposób:

```
Bag<String> stringBag = new Bag<>();
```

Oczywiście można to zapisywać tak jak kiedyś, ale nie ma już takiej konieczności. Intellij pokaże w takiej sytuacji warning.

## Druga dygresja

Istnieje pewna konwencja nazywania typów generycznych, tak jak ja napisałem `Bag<T>`:

*Tabela 1. Przykładowe nazwy typów generycznych*

| Litera | Użycie |
|---|---|
| *K* | Klucz w mapie |
| *V* | Wartość w mapie |
| *E* | Jakiś element |
| *N* | Jakiś numer |
| *T* | Typ generyczny |
| *R* | Typ zwracany |
| *S, U, V* | Gdy mamy kilka typów generycznych w klasie |

# Type erasure

Wiemy już, że narzucając typ generyczny wymuszamy na użytkowniku naszej klasy określenie konkretnego typu generycznego. Czyli tak jak widzieliśmy, zamiast `T` wstawialiśmy przykładowo String. Natomiast to się dzieje na etapie kompilacji.

Pytanie co się dzieje w trakcie działania programu? Kompilator podczas kompilacji zamienia wszystkie typy generyczne `T, U, R, E` itp. na klasę Object. Czyli po wykonaniu kompilacji, wszędzie typy generyczne zostają zamienione na typ Object. Czyli na konkretnym przykładzie, wyglądałoby to tak:

```
public class Bag {

    private Object element;

    public void pack(Object element) {
        this.element = element;
    }

    public Object empty() {
        Object tmp = element;
        this.element = null;
        return tmp;
    }

}
```

Natomiast w miejscu, w którym odwołujemy się do typu Bag, wyglądałoby to w ten sposób:

```
Car car = (Car) bag.empty();
```

Czasem pojawiają się pytania, czy typ generyczny oznacza, że Java stworzy sobie kopię klasy z każdym typem generycznym oddzielnie. Czyli jak zdefiniujemy Bag ze Stringiem i Integerem to mielibyśmy oddzielną kopię klasy ze Stringiem i oddzielną z Integerem. Nie działa to tak ☺.

To zjawisko, proces, o którym tutaj piszę nazywa się **type erasure**.

Dlaczego zostało to zrobione w ten sposób? Bo Java jest kompatybilna wstecznie. Typy generyczne zostały wprowadzone w wersji 5 Javy (upraszczam tutaj konwencję nazewnictwa). Z racji, że Java jest kompatybilna wstecznie, jej twórcy obeszli problem kompilowania starego kodu nowszym kompilatorem właśnie w ten sposób. Bo założeniem Javy jest bycie kompatybilnym ze starszymi wersjami Javy, które nie wykorzystują typów generycznych.

# Czego z typami generycznymi zrobić się nie da?

Większość ograniczeń wynika z procesu type erasure.

Nie możemy zrobić takich rzeczy:

- Wywołać konstruktora typu generycznego, czyli np. `new T()`. Nie jest to możliwe, bo w trakcie działania programu byłoby to `new Object()`,

- Nie możemy stworzyć tablicy typu generycznego, powodem jest to, że w trakcie działania programu otrzymalibyśmy tablicę typu Object,
- Wywołać `instanceof`, nie można tego zrobić bo w trakcie działania programu `Bag<Car>` i `Bag<Apple>` byłoby tym samym przez type erasure,
- Nie możemy używać prymitywów jako typów generycznych, natomiast wiemy już, że istnieje coś takiego jak Autoboxing i mamy klasy takie jak np. Integer,
- Stworzyć zmiennej statycznej typu generycznego. Można to wytłumaczyć w ten sposób, że typ generyczny jest powiązany z instancją klasy, a static nie jest.

# Generic w metodach statycznych

Powiedzieliśmy sobie, że nie można stworzyć zmiennej statycznej typu generycznego.

Można to obejść w ten sposób, że napiszemy metodę statyczną, w której określimy typ generyczny na potrzeby tylko tej metody. Oczywiście metody instancyjne (niestatyczne) też możemy zapisać w ten sposób.

```
// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie, a metoda zwraca void
public static <T> void method1(T element) {
    System.out.println(element);
}

// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie, a metoda zwraca String
public static <T> String method2(T element) {
    System.out.println(element);
    return element.toString();
}

// Tutaj określamy typ generyczny na potrzeby metody,
// który jest używany w argumencie i jednocześnie jest typem zwracanym
public static <T> T method3(T element) {
    System.out.println(element);
    method1(element);
    return element;
}
```

Dodam w tym miejscu, że typy generyczne są bardzo ważną tematyką, bo na co dzień są używane bardzo często. Odkąd zostały wprowadzone, w nowym kodzie nie pisze się kolekcji, które mogą trzymać cokolwiek. Zawsze określa się konkretny przewidziany typ.

# Wildcards

Tak, wiem, jak to brzmi oraz że ciężko będzie te nazwy spamiętać.

W pewnym momencie zauważysz, że z racji wymazywania typów (type erasure) typy generyczne mają pewien minus, który ogranicza ich stosowanie.

```
public class Bag<T> {
```

```
    private T element;

    public void run(T element) {
        // Taki zapis nie jest możliwy,
        // bo przecież w trakcie działania programu zamiast <T> będziemy mieli Object.
        // Object nie ma przecież zdefiniowanej metody someMethod().
        // Java nie pozwala też do klasy Object takiej metody dopisać.
        // Wtedy każda klasa w programie miałaby przecież metodę someMethod().
        // Ale są języki, które pozwalają na coś podobnego.
        element.someMethod();
    }

}
```

Bounded wildcards (bo tak się tę grupę określa) pozwalają na obejście tego problemu poprzez określenie, że typ generyczny ma być pewnego typu.

*Tabela 2. Wildcards podsumowanie*

| Wildcard | Syntax | Przykład |
|---|---|---|
| *Unbounded wildcard* | ? | List<?> l = new ArrayList<>(); |
| *Upper-Bounded Wildcard* | ? extends type | List<? extends Animal> l = new ArrayList<Cat>(); |
| *Lower-Bounded Wildcard* | ? super type | List<? super Animal> l = new ArrayList<Object>(); |

Znak zapytania w tych zapisach oznacza, nieznany typ generyczny i przechodząc po kolei.

# Unbounded wildcard

Unbounded wildcard można rozumieć jako "jakikolwiek typ danych". Czyli pisząc ? mówimy, "daj mi cokolwiek i OK". Na przykładach:

```
public class Example {

    public static void printCities(List<Object> cities) {
        for (int i = 0; i < cities.size(); i++) {
            final Object x = cities.get(i);
            System.out.println(x);
        }
    }
    public static void main(String[] args) {
        List<String> cities = new ArrayList<>();
        cities.add("Wrocław");
        printCities(cities); // Tutaj się nie kompiluje
    }

}
```

Przykład powyżej się nie kompiluje, bo metoda `printCities()` oczekuje List<Object>, a my przekazujemy List<String>. Żeby obejść taki problem, możemy zapisać to tak:

```
class Example {
    public static void printCities(List<?> cities) {
        for (int i = 0; i < cities.size(); i++) {
            // Tutaj wtedy należy przypisać wartość do zmiennej Object.
            // Jeżeli spróbujemy do typu String, dostaniemy błąd kompilacji.
            Object x = cities.get(i);
            System.out.println(x);
        }
    }
}
```

# Upper-Bounded Wildcards

```
class Animal {}

class Cat extends Animal {}

class Dog extends Animal {}

public class Example {

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        List<Cat> cats = new ArrayList<>();
        List<Dog> dogs = new ArrayList<>();
        print(animals);
        // Tutaj dostaniemy błąd kompilacji, bo metoda oczekuje listę List<Animal>,
        // a my przekazujemy List<Cat>
        print(cats);
        // Tutaj dostaniemy błąd kompilacji, bo metoda oczekuje listę List<Animal>,
        // a my przekazujemy List<Dog>
        print(dogs);
    }

    public static void print(List<Animal> animals) {
        for (Animal animal : animals) {
            System.out.println(animal);
        }
    }
}
```

W tym momencie, wjeżdża całe na biało Upper-Bounded Wildcards i wystarczy, że metodę print zapiszemy tak:

```
public static void print(List<? extends Animal> animals) {
    for (Animal animal : animals) {
        System.out.println(animal);
    }
}
```

W tym momencie określamy, że pasuje nam każdy typ generyczny w Liście, który dziedziczy z klasy Animal. Tutaj należy tylko pamiętać, że robi się ciekawa kwestia, która wychodzi, jeżeli przykładowo do listy, określonej jako `List<? extends Animal>` będziemy chcieli dodać jakieś elementy (wyobraźmy sobie, że klasy Animal, Cat i Dog są zdefiniowane jak w poprzednim przykładzie):

```
public class Example {

    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        List<Cat> cats = new ArrayList<>();
        List<Dog> dogs = new ArrayList<>();

        // Tutaj jest wszystko w porządku,
        // referencja upperBoundAnimals może wskazywać na listę List<Dog>,
        // to oznacza zapis List<? extends Animal>
```

```
        // Zwróć też uwagę, że na tym etapie, wiemy też,
        // jaka konkretnie jest zawartość listy dogs i że są tam same psy, nie ma tam kotów.
        List<? extends Animal> upperBoundAnimals = dogs;

        // Czegoś takiego zapisać nie możemy,
        // bo nie wiadomo czy finalnie ma to wtedy być lista List<Animal>,
        // czy List<Cat>, czy List<Dog>.
        // Zatem na wszelki wypadek, żaden z poniższych zapisów nie jest dozwolony
        upperBoundAnimals.add(new Dog());
        upperBoundAnimals.add(new Cat());
        upperBoundAnimals.add(new Animal());
    }
}
```

# Lower-Bounded Wildcard

Wyobraźmy sobie teraz taką sytuację:

```
class Animal {}

class Cat extends Animal {}

class Dog extends Animal {}

public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        List<Animal> animals = new ArrayList<>(cats);

        addCat1(cats);
        addCat1(animals);

        addCat2(cats);
        addCat2(animals);

        addCat3(cats);
        addCat3(animals);
    }

    public static void addCat1(List<?> list) {
        list.add(new Cat());
    }

    public static void addCat2(List<Object> list) {
        list.add(new Cat());
    }

    public static void addCat3(List<? extends Animal> list) {
        list.add(new Cat());
    }

}
```

Próbujemy w metodach addCat1, addCat2, addCat3 dodać Cat do listy i jednocześnie zapisać to w taki sposób, żeby parametr tej metody mógł przyjąć listę Cat i listę Animal jednocześnie. Tutaj uprzedzę, że jest to skrajny przypadek, żeby robić coś takiego w praktyce, ale należy wiedzieć jak podobną sytuację rozwiązać.

I kolejno:

- addCat1 - do listy `List<?>` list nie możemy dodać `new Cat()`, bo dostaniemy błąd kompilacji. Ma sens, w sumie w tej liście może być cokolwiek przez `?`,
- addCat2 - wprawdzie do listy `List<Object>` list możemy dodać `new Cat()`, bo przecież Cat rozszerza klasę Animal, ale do metody `addCat2()` nie możemy przekazać ani listy `List<Cat>`, ani listy `List<Animal>`, bo metoda spodziewa się od nas listy `List<Object>`. Rozwiązaniem byłoby zdefiniowanie w metodzie addCat2 parametru w sposób `List<?> list`, ale wtedy w ciele metody nie moglibyśmy dodać `new Cat()`, jak w przykładzie addCat1,

- addCat3 - jeżeli parametr w metodzie addCat3 określimy jako `List<? extends Animal> list` to możemy do wywołania tej metody przekazać listy `List<Cat>` oraz `List<Animal>`, natomiast dostaniemy błąd kompilacji tak jak wyjaśnione zostało w przykładzie Upper-Bounded Wildcard.

Widać zatem, że żaden z poznanych sposobów nie działa ☹.

I w tym momencie na białym koniu wjeżdża Lower-Bounded Wildcard:

```
public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        List<Animal> animals = new ArrayList<>(cats);

        addCat(cats);
        addCat(animals);
    }

    public static void addCat(List<? super Cat> list) {
        list.add(new Cat());
    }

}
```

W ten sposób mówimy Javie, że będzie to lista `List<Cat>` albo lista obiektów, które są superklasą klasy `Cat`. Czyli klas, z których `Cat` dziedziczy. Jednocześnie dajemy znać, że jest okej dodać `Cat` do tej listy, bo przecież, będą w tej liście zawsze "przynajmniej" obiekty klasy `Cat`, więc spokojnie można do tej listy dodać kolejnego Kota. Albo obiekty dziedziczące z klasy `Cat` (np. `SuperCat`).

```
class SuperCat extends Cat {}
// gdzieś w kodzie
public static void addCat(List<? super Cat> list) {
    list.add(new Cat());
    list.add(new SuperCat());
    list.add(new Animal()); // Ale tak już nie można, dostaniemy błąd kompilacji
}
```

Nie zadziała wtedy również coś takiego:

```
List<SuperCat> superCats = new ArrayList<>();
addCat(superCats);
```

Nie zadziała dlatego, że zapisem `List<? super Cat>` mówimy, że chcemy tutaj listę z obiektami `Cat` albo z klasami, które są superklasami dla `Cat`. `SuperCat` jest subklasą `Cat`, a nie jego superklasą, dlatego to nie zadziała.

Lower-Bounded Wildcard były dla mnie osobiście najtrudniejsze do przyswojenia, musiałem przez to przejść dobre parę razy. W praktyce używa się ich natomiast bardzo rzadko ☹.

# Podsumowanie

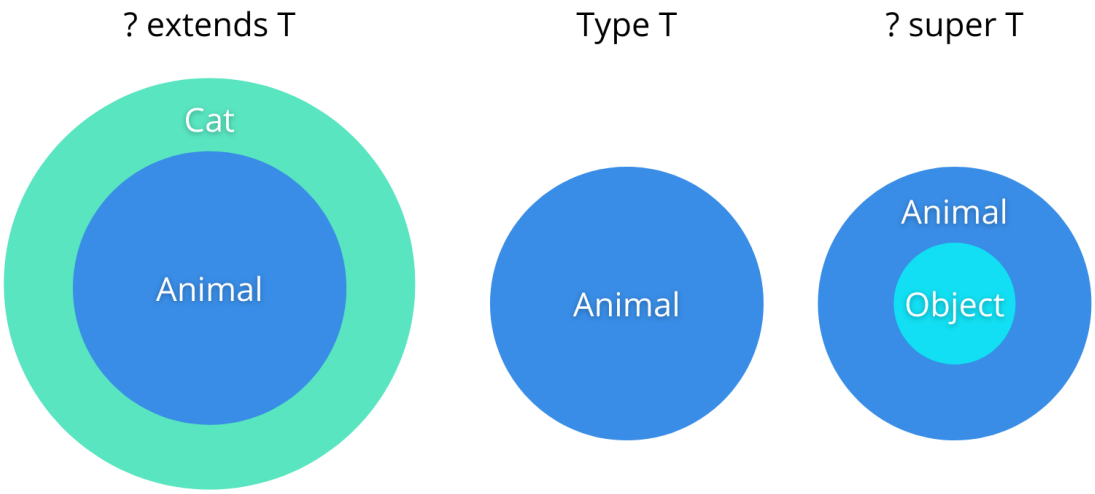
Wiemy już na tym etapie czym są typy generyczne, do czego służą i w jaki sposób mają nam pomóc programować. Powinniśmy już rozumieć, że typ generyczny może być wykorzystany z klasą, interfejsem, a nawet metodą. Na tym etapie powinniśmy mieć również zrozumienie czego z generykami zrobić się nie da. Powinniśmy również rozumieć, że na koniec dnia i tak wszystko jest obiektem 😄.

Dowiedzieliśmy się również, że oprócz "prostego" stosowania generyków, można takie przykłady mocno skomplikować wykorzystując mechanizmy takie jak **Upper-Bounded Wildcards** lub **Lower-Bounded Wildcard**. Jeżeli mielibyśmy problem z zapamiętaniem co i kiedy to z pomocą może przyjść **pecs**.

# PECS

Jeżeli będziemy mieli problem z zapamiętaniem, kiedy stosować `upper` bound, a kiedy `lower` bound to pomocne może być przypomnienie sobie zasady **pecs**, której proste wyjaśnienie znajdziemy na stackoverflow. Parafrazując:

Zasada **pecs** oznacza patrzenie z punktu widzenia kolekcji. Jeżeli wyjmujemy generyczne dane z kolekcji - kolekcja działa wtedy jak producent danych i powinniśmy stosować słówko `extends`. Jeżeli natomiast wkładamy dane do kolekcji - kolekcja zachowuj się wtedy jak konsument i stosujemy słówko `super`. Jeżeli robimy obie rzeczy jednocześnie, nie powinniśmy używać ani `extends` ani `super`.




*Obraz 1. PECS*

Biorąc pod uwagę powyższe wyjaśnienie, możemy teraz łatwo rozwinąć skrót **pecs** - ***p**roducer **e**xtends **c**onsumer **s**uper*.