TYPY GENERYCZNE (GENERICS)

ZAGADNIENIA:

- wprowadzenie,
- konwencje, metody, typy surowe
- parametry ograniczone
- podtypy, dziedziczenie,
- symbole wieloznaczne,
- ograniczenia.

MATERIAŁY:

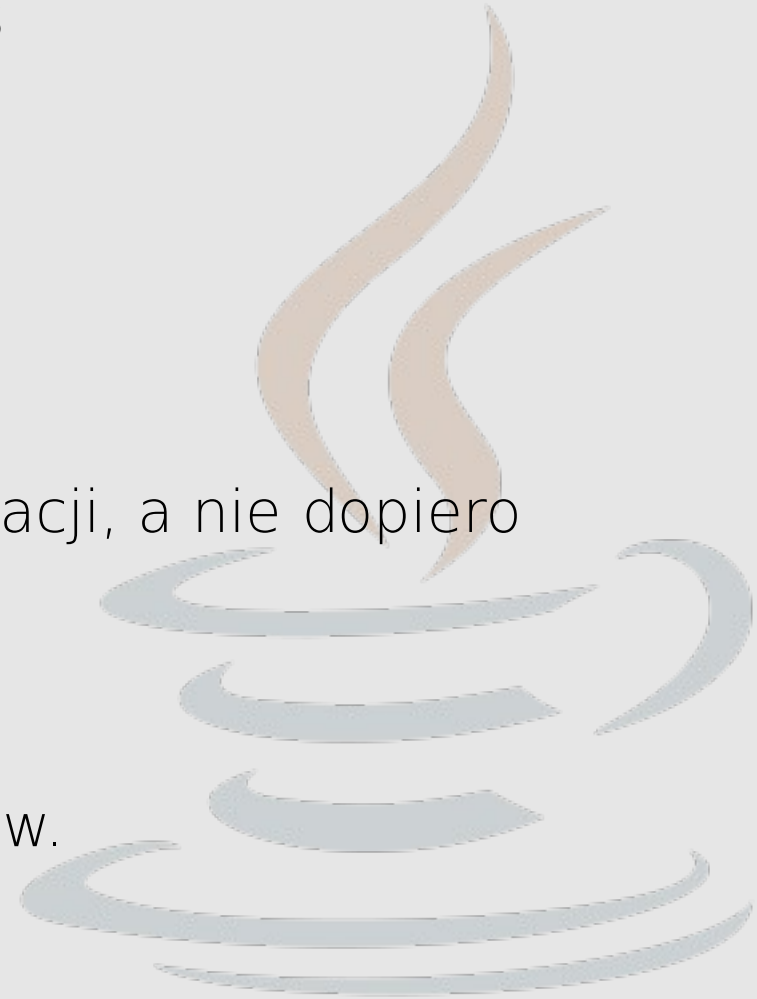
http://docs.oracle.com/javase/tutorial/java/generics/

TYPY GENERYCZNE

```
List<String> list = new ArrayList<String>();
list.add("Ala");
String s = list.get(0);
```

ZALETY:

- kontrola typów na poziomie kompilacji, a nie dopiero w trakcie działania programu
- brak potrzeby rzutowania,
- konstruowanie ogólnych algorytmów.



PRZYKŁAD

```
public class Box<T> { // T oznacza Typ
    private T t;

    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

ogólnie:

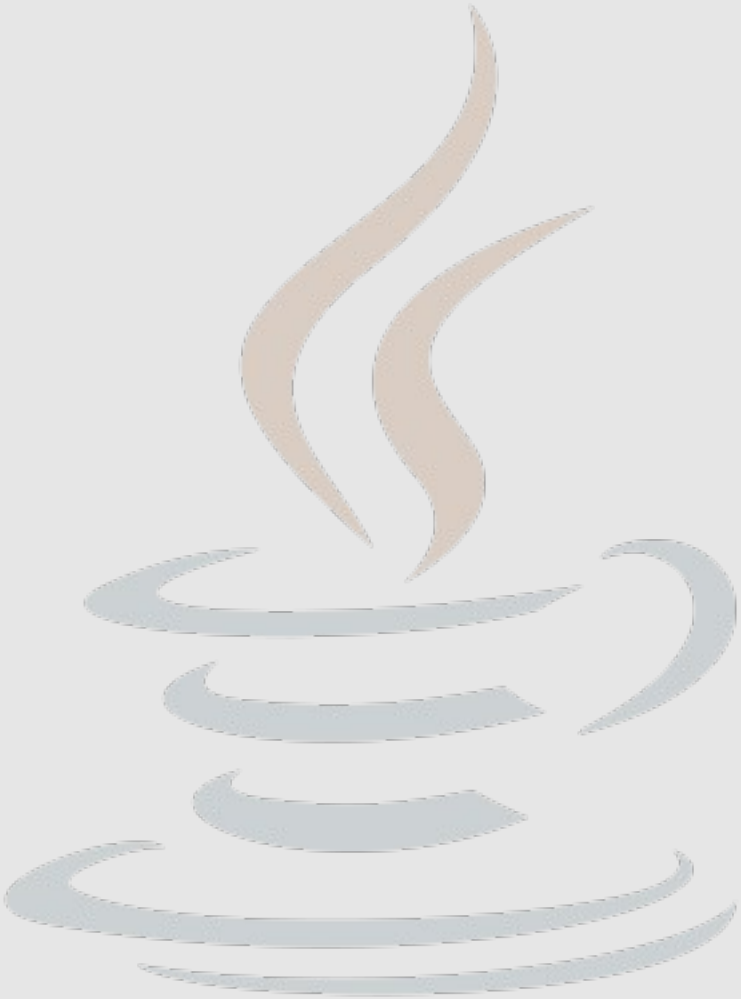
```
class NazwaKlasy<T1, T2, ..., Tn> {
...
}
```

KONWENCJE

- T – typ,
- K – klucz,
- V – wartość,
- E – element (np. kolekcji),
- N – liczba

PRZYKŁAD:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```



PRZYKŁADY

```
public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}

Pair<String, Integer> p = new OrderedPair<String, Integer>("Even", 8);
Pair<String, Integer> p = new OrderedPair<>("Even", 8);  // Java 7
```

TYPY SUROWE

```
public class Box<T> {
    public void set(T t){ ... }
    ...
}

Box<String> typedBox = new Box<String>();
Box rawBox = new Box(); // typ surowy <=> Box<Object>


Box rawBox = typedBox; // warning: przypisanie możliwe
                       // (Objct o – new String()), ale brak jawnej
                       // konwersji typów

rawBox.set(8);         // warning: w trakcie działania programu może
                       // spowodować RuntimeException
```

Ze względu na możliwość popełnienia błędów, nie zaleca się stosowania typów surowych.

METODY GENERYCZNE

```
public class Util {
    // Genericzna metoda statyczna
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

wywołanie metody generycznej:

```
Pair<Integer, String> p1 = new Pair<>(1, "jabłko");
Pair<Integer, String> p2 = new Pair<>(2, "gruszka");
boolean same = Util.<Integer, String>compare(p1, p2);
```

możliwe również prostsze wywołanie dzięki mechanizmowi wnioskowania typów (type inference).

```
boolean same = Util.compare(p1, p2);
```

PARAMETRY OGRANICZONE

```
public class Box<T> {
...
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: String to nie Number
    }
}
```

inny przykład

```
public class NaturalNumber<T extends Integer>
```

PARAMETRY OGRANICZONE

wielokrotne ograniczenia:

```
<T extends B1 & B2 & B3>
```

przykład:

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

typ **T** musi rozszerzać klasę **A** i implementować interfejsy **B** i **C**.

UWAGA: **A** musi być zadeklarowana jako pierwsza (błąd w trakcie kompilacji)

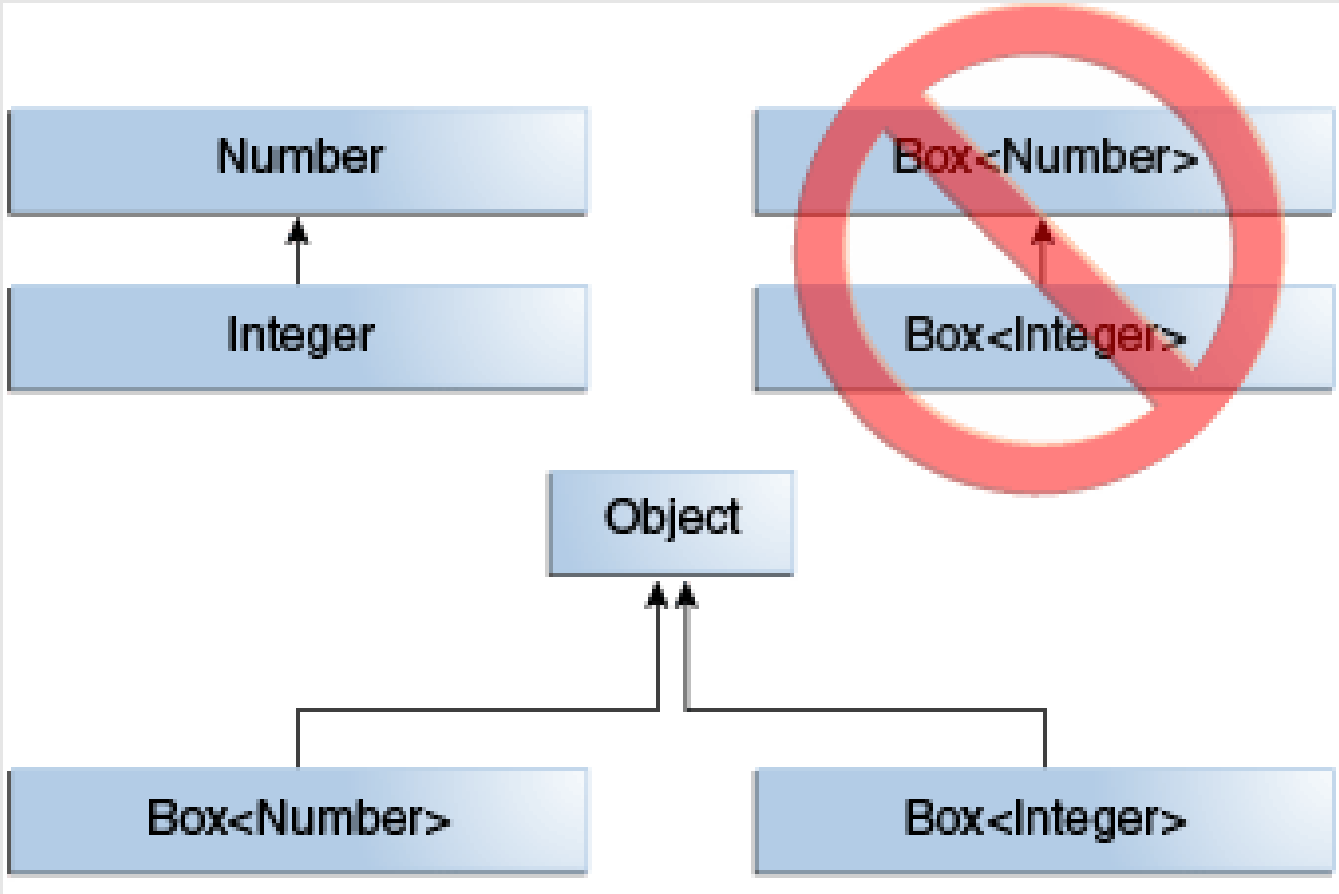
PRZYKŁAD

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // błąd kompilacji. Co oznacza '>'?
            ++count;
    return count;
}
```

poprawnie:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray,
                                                             T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

PODTYPY, DZIEDZICZENIE

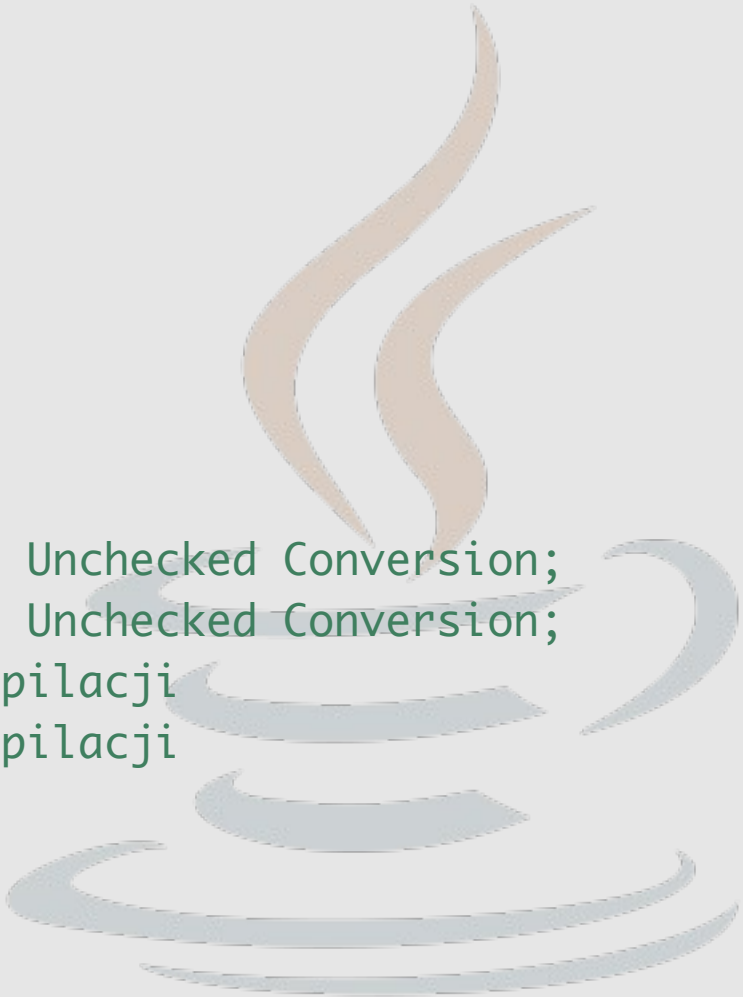


http://docs.oracle.com/javase/tutorial/figures/java/generics-subtypeRelationship.gif

DZIEDZICZENIE PRZYKŁAD

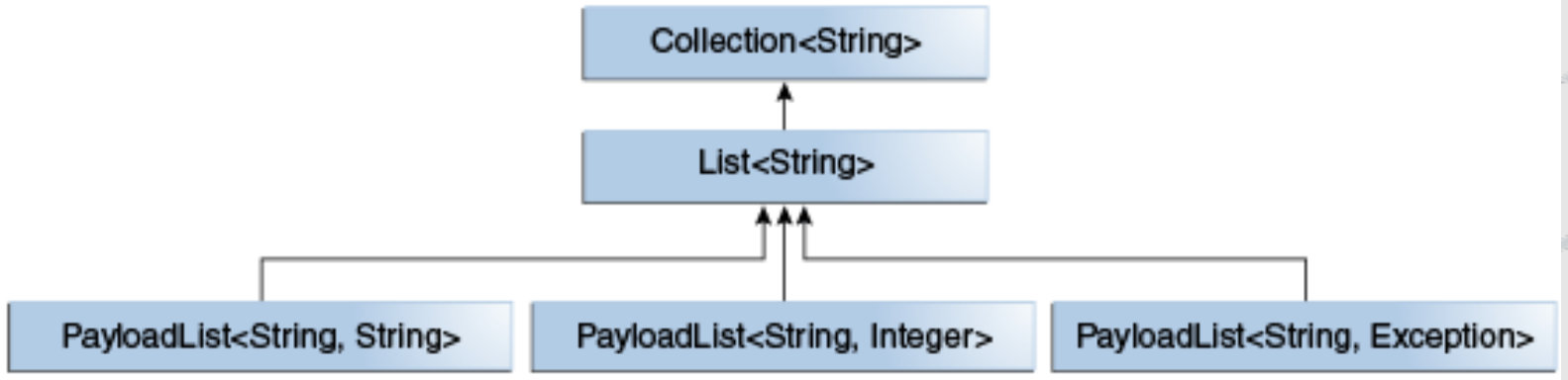
```
Box<Integer> boxInteger;
Box<Number> boxNumber;
Box box;

box        = boxInteger;  // ok
box        = boxNumber;   // ok
boxNumber  = box;         // warning: Unchecked Conversion;
boxInteger = box;         // warning: Unchecked Conversion;
boxInteger = boxNumber;   // błąd kompilacji
boxNumber  = boxInteger;  // błąd kompilacji
```



PODTYPY, DZIEDZICZENIE

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```



http://docs.oracle.com/javase/tutorial/figures/java/generics-payloadListHierarchy.gif

SYMBOLE WIELOZANCZNE

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Znak '?' ogranicza 'od góry' rodzaj parametru.

UWAGA: deklaracja / definicja metody

```
public static double sumOfList(List<T extends Number> list)
```

jest niepoprawna (składnia). Taka składnia jest douszczalna dla deklaracji klas lub wartości zwracanych (por. parametry ograniczone).

SYMBOLE WIELOZNACZNE

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Znak '?' dopuszcza dowolny typ jako zawartość listy, np.

`List<String>`, `List<Integer>`, itd.

UWAGA: Metoda

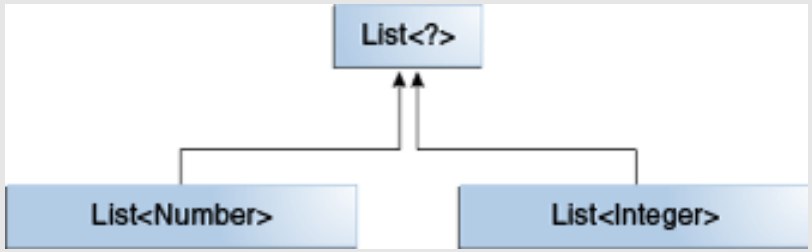
```
public static void printList(List<Object> list)
```

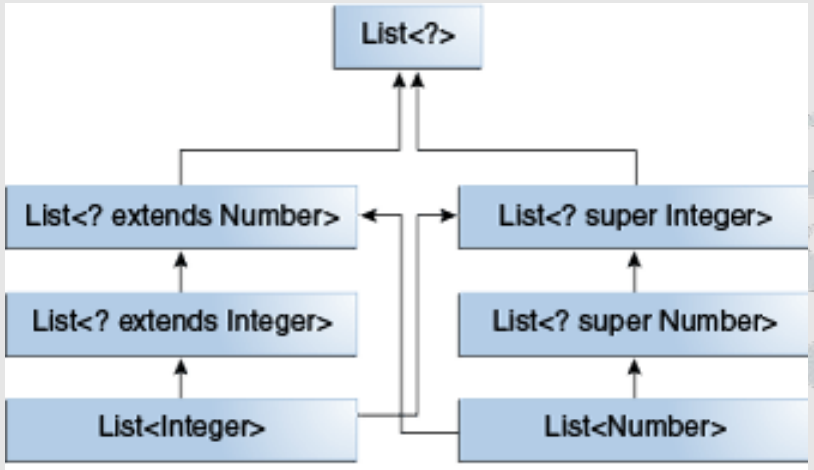
nie obsłuży `List<String>` bo `List<String>` nie jest podklasą

`List<Object>`.

SYMBOLE WIELOZNACZNE

Dziedziczenie a symbole wieloznaczne





http://docs.oracle.com/javase/tutorial/figures/java/generics-wildcardSubtyping.gif

SYMBOLE WIELOZNACZNE

Przechwytywanie symboli wieloznacznych (wildcard capture):

```
public class WildcardError {

    void doSomething(List<?> l) {
        l.set(0, l.get(0)); // blad kompilacji
    }
}
```

Kompilator traktuje `l` jako listę dowolnych obiektów, podczas gdy wywołanie metody `l.set()` wstawia do listy konkretny obiekt, który niekoniecznie musi do niej pasować.

SYMBOLE WIELOZNACZNE

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }

    // metoda pomocnicza, która rozwiązuje problem
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Metoda wykorzystuje mechanizm wnioskowania (inference). Tutaj lista jest procesowana jak lista zawierająca konkretne elementy (typu **T**) więc mogą to być obiekty **Integer**.

SYMBOLE WIELOZNACZNE

Wskazówki:

- zmienne IN – górne ograniczenie **extends**,
- zmienne OUT – dolne ograniczenie **super**,
- zmienne IN, mogące być używane jako obiekty – brak ograniczeń,
- zmienne IN/OUT – nie używamy symboli wieloznacznych,
- nie używamy symboli wieloznacznych jako wartości zwracanych.

Listy zadeklarowane z użyciem symboli wieloznacznych można traktować jako obiekty read-only

SYMBOLE WIELOZNACZNE

```
List<Integer> li = new ArrayList<Integer>();
List<? extends Number> ln = li; // OK. ( Object obj = new String() )
ln.add(new Integer(35));  // Blad kompilacji, ( obj.charAt(7) )
```

`List<Integer>` jest podtypem `List<? extends Number>`. A klasy nie można traktować tak jak podklasy. Jedyne operacje które można wykonać na takiej liście to:

- dodanie null
- wyczyszczenie clear()
- pobranie iteratora i wywołanie remove()
- zapisać elementy odczytane z listy z wykorzystaniem 'wildcard capture'.

DZIAŁANIE

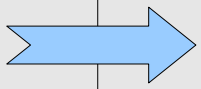
Typy generyczne zostały wprowadzone aby lepiej kontrolować typy na etapie kompilacji i aby umożliwić programowanie generyczne. W generowanym bytecodzie otrzymujemy zwykłe klasy i interfersy. Aby to zrealizować kompilator:

- zastępuje typy przez ich ograniczenia a nieograniczone typy przez **Object**,
- dodaje niezbędne operacje rzutowania,
- tworzy metody pomostowe implementujące polimorfizm w rozszerzonych typach generycznych.

DZIAŁANIE

```
public class Box<T> {

    private T t;
    public void set(T t) {
        this.t = t;
    }
}
```

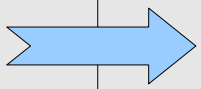


```
public class Box {

    private Object t;
    public void set(Object t) {
        this.t = t;
    }
}
```

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

    private T t;
    public void set(T t) {
        this.t = t;
    }
}
```



```
public class Box {

    private Number t;
    public void set(Number t) {
        this.t = t;
    }
}
```

METODY POMOSTOWE

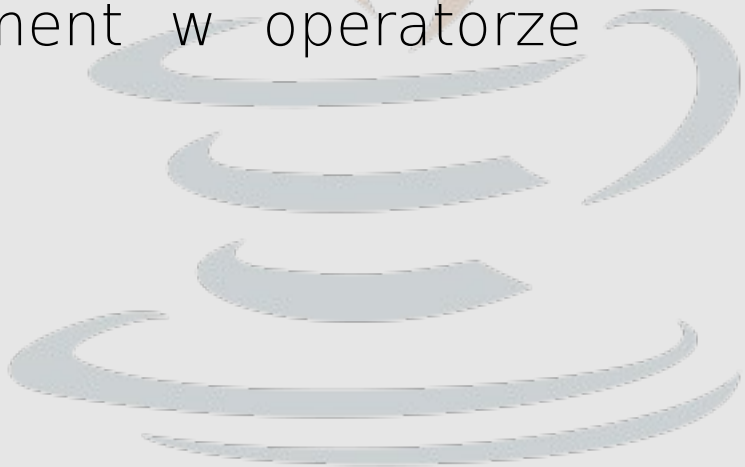
```
public class MyBox extends Box<String>{
    public void set(String s){
        super.set(s);
    }
}
```

metoda **set()** w klasie **Box** przyjmuje argument Object podczas gdy metoda w klasie potomnej przyjmuje **String**. To powodowałoby brak polimorfizmu. Dlatego kompilator automatycznie dodaje metodę pomostową (do **MyBox**):

```
public void setData(Object data) {
    setData((Integer) data);
}
```

OGRANICZENIA

- nie można używać typów prymitywnych (**Box<int>**),
- nie można używać operatora **new** (**E e = new E();**),
- nie można deklarować typów statycznych (**private static T v;**)
- typów parametryzowanych (**List<Integer>**) nie można rzutować ani używać jako argument w operatorze `instanceof`,



OGRANICZENIA

- nie można używać tablic typów parametryzowanych (**List<Integer>[] arrayOfLists = new List<Integer>[2]**),
- typ generyczny nie może rozszerzać (bezpośrednio lub pośrednio) **Throwable**,
- nie można przeciążać metod, których argumenty sprowadzają się do tego samego typu.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

DZIĘKUJĘ ZA UWAGĘ