Teoria do lab #12

Programowanie generyczne

Definicja

Programowanie generyczne to koncepcja w programowaniu, która pozwala na pisanie kodu, który może być używany z różnymi typami danych, bez konieczności powtarzania tego samego kodu dla każdego typu danych. Jest to szczególnie przydatne w językach programowania silnie typowanych, takich jak Java, C# czy C++, gdzie typy danych muszą być określone podczas kompilacji. Oto kluczowe aspekty programowania generycznego:

1. **Typy Parametryzowane:** Programowanie generyczne umożliwia tworzenie klas, interfejsów i metod, które działają na "typach generycznych". Te typy generyczne są określone jako parametry, zazwyczaj reprezentowane przez litery, takie jak T, E, K, V itp.
2. **Zwiększona Znacząco Bezpieczeństwo Typów:** Dzięki temu, że typy są określone podczas kompilacji, programowanie generyczne pomaga uniknąć błędów związanych z nieprawidłowym rzutowaniem typów, które mogą wystąpić w trakcie działania programu.
3. **Ograniczenia Typów:** Możliwe jest narzucenie ograniczeń na typy generyczne, tak aby akceptowały tylko klasy, które spełniają określone wymagania (np. dziedziczenie po konkretnej klasie bazowej lub implementowanie określonego interfejsu).
4. **Kod Współużytkowany:** Kod napisany w sposób generyczny może być używany z różnymi typami danych, co zmniejsza redundancję i ułatwia utrzymanie kodu.
5. **Kompilacja Typu Bezpiecznego:** Podczas kompilacji, kompilator sprawdza, czy kod generyczny jest używany poprawnie zgodnie z określonymi typami, co zapewnia wyższe bezpieczeństwo typów i pomaga w wykrywaniu błędów na wcześniejszym etapie rozwoju oprogramowania.

Przykładowe Zastosowania:

- **Kolekcje:** W językach takich jak Java, generyki są powszechnie stosowane w bibliotekach kolekcji (np. `List<T>`, `Map<K,V>`), co pozwala na tworzenie kolekcji, które mogą przechowywać elementy dowolnego typu, jednocześnie zapewniając bezpieczeństwo typów.
- **Algorytmy:** Generyki pozwalają na pisanie algorytmów, które mogą pracować na różnych typach danych.

Misz masz pojęciowy

1. **Klasa Generyczna (Generic Class)** Klasa generyczna w programowaniu obiektowym to taka, która pozwala na zdefiniowanie klasy z jednym lub więcej nieokreślonymi typami. Te typy są określone dopiero podczas tworzenia instancji klasy. Klasy generyczne są używane do tworzenia kodu, który jest niezależny od konkretnych typów, a więc może być używany w sposób bardziej elastyczny i bezpieczny pod względem typów.

- **Przykład w Javie:**

```
public class Box<T> {
    private T t; // T to typ generyczny

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

2. **Typ Parametryzowany (Parameterized Type)** Typ parametryzowany to konkretyzacja klasy generycznej z określonymi typami. Kiedy tworzysz obiekt klasy generycznej, musisz określić konkretne typy dla jej parametrów generycznych.

- **Przykład:**
  - Mając klasę generyczną `Box<T>`, możesz utworzyć jej instancję jako `Box<Integer>` lub `Box<String>`. Tutaj `Box<Integer>` i `Box<String>` są typami parametryzowanymi.

3. **Typ Generyczny (Generic Type)** Typ generyczny to termin ogólnie odnoszący się do klas, interfejsów i metod, które używają typów parametryzowanych. Obejmuje on zarówno definicję klasy generycznej (jak `Box<T>`), jak i konkretne typy parametryzowane (jak `Box<Integer>`).

4. **Szablon Klas (Class Template)** Szablon klas jest pojęciem bardziej związanym z językami programowania takimi jak C++, które stosują "templates" do osiągnięcia podobnych celów, co generyki w Javie. Szablon klasy w C++ jest schematem dla tworzenia klas lub funkcji, które mogą działać z dowolnym typem.

- **Przykład w C++:**

```
template <typename T>
class Box {
    T t;
public:
    void set(T t) { this->t = t; }
    T get() { return t; }
};
```

Przykład klasyczny z książki Horstmana

Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.

Projekt W11, pakiet: example17

```
package example17;

// Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
```

```
    public void setFirst(T newValue) {
        first = newValue;
    }

    public void setSecond(T newValue) {
        second = newValue;
    }
}
```

```
package example17;

//Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.

public class ArrayAlg {

    public static Pair<String> minmax(String[] a) {
        if (a == null || a.length == 0) {
            return null;
        }

        String min = a[0];
        String max = a[0];

        for (int i = 1; i < a.length; i++) {
            if (min.compareTo(a[i]) > 0) {
                min = a[i];
            }

            if (max.compareTo(a[i]) < 0) {
                max = a[i];
            }
        }

        return new Pair<>(min, max);
    }
}
```

```
package example17;

public class TestPair {
```

```
    public static void main(String[] args) {
        Pair<String> p = new Pair<>("Jan", "Kowalski");
        System.out.println(p.getFirst() + " " + p.getSecond());
        p.setFirst("Adam");
        p.setSecond("Nowak");
        System.out.println(p.getFirst() + " " + p.getSecond());
        String[] words = {"Ala", "ma", "kota", "i", "psa"};
        Pair<String> mm = ArrayAlg.minmax(words);
        System.out.println("min = " + mm.getFirst());
        System.out.println("max = " + mm.getSecond());
    }
}
```

Jakie nazwy?

W programowaniu generycznym w Javie stosuje się pewne konwencje dotyczące oznaczeń typów generycznych, aby ułatwić zrozumienie kodu. Oto najczęściej używane oznaczenia:

1. **E** - Element: Jest używany głównie w kolekcjach, jak `java.util.List<E>`, `java.util.Set<E>`, gdzie E oznacza typ elementów w kolekcji.
2. **K** - Key: Używany w kontekście map i wpisów mapy, gdzie K reprezentuje typ klucza. Na przykład w `java.util.Map<K, V>`.
3. **V** - Value: Również używany w mapach, gdzie V oznacza typ wartości. W `java.util.Map<K, V>`, K to klucz, a V to wartość.
4. **T** - Type: Jest to ogólny typ, który może być używany w dowolnym kontekście. Na przykład, w klasach generycznych jak `java.util.ArrayList<T>`, gdzie T oznacza typ przechowywanych elementów.
5. **N** - Number: Czasami używany do oznaczania liczbowych typów danych, szczególnie w klasach rozszerzających `java.lang.Number`.
6. **S, U, V** itd.: Te litery są używane, gdy są potrzebne dodatkowe typy generyczne, i zwykle są stosowane w kolejności alfabetycznej.

https://docs.oracle.com/javase/tutorial/java/generics/types.html

Metody generyczne

Tworzenie metod generycznych umożliwia pisaniu metod, które mogą operować na różnych typach danych, jednocześnie zapewniając bezpieczeństwo typów w czasie kompilacji. Oto jak możesz tworzyć metody generyczne:

1. **Deklaracja Typu Generycznego**: Typ generyczny jest deklarowany przed typem zwracanym metody. Używa się do tego liter jak T, E, K, V, itd., które działają jako zmienne reprezentujące typy.

   Przykład:

   ```
   public <T> void printArray(T[] array) {
       for (T element : array) {
           System.out.println(element);
       }
   }
   ```

   W tym przykładzie `<T>` przed `void` oznacza, że metoda `printArray` jest generyczna i operuje na typie `T`.

2. **Używanie Typów Generycznych w Ciele Metody**: Możesz używać tych typów generycznych jako typów zmiennych, parametrów i typów zwracanych w metodzie.

3. **Ograniczenia Typów (Type Bounds)**: Możesz ograniczyć rodzaje typów, które mogą być używane z danym typem generycznym, używając słowa kluczowego `extends` (dla klas i interfejsów) lub `super` (dla ograniczeń dolnych).

   Przykład:

   ```
   public <T extends Comparable<T>> T findMax(T[] array) {
       T max = array[0];
       for (T element : array) {
           if (element.compareTo(max) > 0) {
               max = element;
           }
       }
       return max;
   }
   ```

   W tym przypadku `<T extends Comparable<T>>` oznacza, że typ `T` musi implementować interfejs `Comparable<T>`.

4. **Wywoływanie Metod Generycznych**: Podczas wywoływania metody generycznej, kompilator zazwyczaj jest w stanie wywnioskować typ generyczny na podstawie kontekstu, ale można też jawnie podać typ generyczny.

Przykład:

```
Integer[] intArray = {1, 2, 3};
printArray(intArray); // Kompilator wywnioskuje, że T to Integer

String[] stringArray = {"Hello", "World"};
printArray(stringArray); // Kompilator wywnioskuje, że T to String
```

Przykład - metoda generyczna statyczna z dowolną ilością argumentów

Projekt W12, example18

```
package example18;

// //Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class ArrayAlg {

    public static <T> T getMiddle(T... a) {
        return a[a.length / 2];
    }
}
```


```
package example18;

public class TestArrayAlg {

    public static void main(String[] args) {
        String[] words = {"ABC", "DEF", "GHI", "JKL", "MNO"};
        String middle = ArrayAlg.getMiddle(words);
        System.out.println(middle);
        Integer[] numbers = {1, -2, 7, 8, 12};
        Integer middle2 = ArrayAlg.getMiddle(numbers);
        System.out.println(middle2);
        System.out.println(ArrayAlg.getMiddle("ABC", "DEF", "GHI"));
        System.out.println(ArrayAlg.getMiddle(3.4, 177.0, 3.14, -5.6, 177.1));
    }
}
```

Przykład - statyczna metoda generyczna której argumentem jest tablica

Projekt W12, example19

```
package example19;

public class Test19 {

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5};
        Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
        print(intArray);
        print(doubleArray);
        print(charArray);
    }

    public static <T> void print(T[] array) {
        for (T t : array) {
            System.out.println(t);
        }
    }
}
```

Przykład - inne kombinacje, nie zawsze zalecane

Projekt W12, example20

```
package example20;

import java.util.Optional;

public class Test20 {

    public static void main(String[] args) {
        System.out.println(foo("ABC"));
        System.out.println(foo(123));
        System.out.println(foo(3.14));
        System.out.println(foo2(123));
        System.out.println(foo2(3.14));
        //System.out.println(foo2("ABC"));
```

```
        System.out.println(Optional.ofNullable(foo3()));
        System.out.println(Optional.ofNullable(foo4(0)));
    }

    public static <T> int foo(T arg){
        return arg.hashCode();
    }

    public static <T> int foo2(T arg) {
        if (arg instanceof Number) {
            return (int) Math.pow(((Number) arg).doubleValue(), 2);
        }
        throw new IllegalArgumentException("Arg musi być liczbą");
    }

    public static <T> T foo3(){
        return null;
    }

    public static <T> T foo4(int arg) {
        if (arg == 0) {
            return (T) Integer.valueOf(arg);
        } else if (arg == 1) {
            return (T) "String";
        }
        return (T) new Object();
    }


}
```

Ograniczenia zmiennych typowych

```
package example21;

// //Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.

public class ArrayAlg {

    public static <T> T min(T... a) {
```

```
        if (a == null || a.length == 0) {
            return null;
        }

        T smallest = a[0];
        for (int i = 1; i < a.length; i++) {
            if (smallest.compareTo(a[i]) > 0) {
                smallest = a[i];
            }
        }

        return smallest;
    }
}
```

Czy czegoś tu nie brak?

Poprawna forma

Projekt W12, example21

```
package example21;

// //Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.

public class ArrayAlg {

    public static <T extends Comparable<T>> T min(T... a) {
        if (a == null || a.length == 0) {
            return null;
        }

        T smallest = a[0];
        for (int i = 1; i < a.length; i++) {
            if (smallest.compareTo(a[i]) > 0) {
                smallest = a[i];
            }
        }

        return smallest;
    }
```

typ T musi implementować Comparable<T>

```
}
```

```
package example21;

public class Person implements Comparable<Person>{

    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "[name=" + name + ",age=" + age + "]";
    }


}
```

```
package example21;

public class TestPerson {

    public static void main(String[] args) {
        Double[] numbers = {1.0, 12.0, -3.0};
        System.out.println(ArrayAlg.min(numbers));
        Person[] people = {new Person("Jan", 12), new Person("Anna", 10), new Person("Piot
        System.out.println(ArrayAlg.min(people));
    }
}
```

Interpretacja

`<T extends typ_graniczny>` jest składnią używaną w programowaniu generycznym do określenia górnej granicy dla typu generycznego `T`. Oznacza to, że `T` musi być podtypem (klasą pochodną) klasy określonej jako `typ_graniczny` lub sama być tym typem.

Wymazywanie typów

Wymazywanie typów (ang. type erasure) to proces stosowany w Javie w kontekście programowania generycznego, który zapewnia kompatybilność wsteczną z wcześniejszymi wersjami Javy, które nie obsługiwały generyków. Kiedy kod zawierający generyki jest kompilowany, kompilator usuwa (wymazuje) wszelkie informacje o typach generycznych, zastępując je ich ograniczeniami lub, jeśli takie nie istnieją, obiektem najbardziej ogólnym (często `Object`).

Jak Działa Wymazywanie Typów?

1. **Zastępowanie Typów Generycznych:**
    - Kompilator zastępuje wszystkie typy generyczne ich ograniczeniami lub `Object`, jeśli brak jest ograniczeń. Na przykład, dla `class Box<T>`, `T` zostanie zastąpione przez `Object` podczas kompilacji.
2. **Usuwanie Metadanych o Typach:**
    - Informacje o typach generycznych są usuwane, więc w czasie wykonania (runtime) nie ma dostępu do tych informacji. Na przykład, nie można sprawdzić czy lista jest typu `List<String>` czy `List<Integer>` w czasie wykonania.
3. **Mostowanie Metod:**
    - W niektórych przypadkach kompilator może dodać metody mostowe (bridge methods) w celu utrzymania polimorfizmu dla dziedziczonych klas generycznych.

Wymazywanie typów ma kilka konsekwencji: - **Brak Możliwości Przeciążania Metod:** Metody różniące się jedynie typem generycznym nie mogą być przeciążone, ponieważ po wymazaniu będą miały ten sam sygnaturę.

- **Konieczność Rzutowania:** W czasie wykonania trzeba czasami ręcznie rzutować obiekty na odpowiedni typ, co może prowadzić do błędów `ClassCastException`.
- **Brak Możliwości Sprawdzenia Typu Generycznego w Runtime:** Nie można używać refleksji do dokładnego ustalenia typu generycznego w czasie wykonania, ponieważ informacje te są wymazane.

Przykład wymazywania dla klasy generycznej

```
// Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }

    public void setFirst(T newValue) {
        first = newValue;
    }

    public void setSecond(T newValue) {
        second = newValue;
    }
}
```

jest wymazywany na:

```
public class Pair {

    private Object first;
    private Object second;
```

```
    public Pair() {
        first = null;
        second = null;
    }

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() {
        return first;
    }

    public Object getSecond() {
        return second;
    }

    public void setFirst(Object newValue) {
        first = newValue;
    }

    public void setSecond(Object newValue) {
        second = newValue;
    }
}
```

Przykład wymazywania dla metody generycznej

```
public static <T extends Comparable<T>> T min(T... a) {
    if (a == null || a.length == 0) {
        return null;
    }

    T smallest = a[0];
    for (int i = 1; i < a.length; i++) {
        if (smallest.compareTo(a[i]) > 0) {
            smallest = a[i];
        }
    }
```

```
    return smallest;
}
```

jest wymazywany na:

```
public static Comparable min(Comparable... a) {
    if (a == null || a.length == 0) {
        return null;
    }

    Comparable smallest = a[0];
    for (int i = 1; i < a.length; i++) {
        if (smallest.compareTo(a[i]) > 0) {
            smallest = a[i];
        }
    }

    return smallest;
}
```

Typy generyczne a typy proste (prymitywne)

Typy proste (ang. primitive types), takie jak `int`, `double`, `char`, itd., nie mogą być używane jako typy generyczne. Wynika to z kilku powodów, głównie związanych z tym, jak generyki są implementowane w Javie oraz jak działają typy proste:

1. **Wymazywanie Typów (Type Erasure):**
   - Generyki w Javie są implementowane za pomocą mechanizmu zwanego "wymazywaniem typów" (type erasure), co oznacza, że informacje o typach generycznych są usuwane w czasie kompilacji, a zamiast nich stosowane są ograniczenia lub typ `Object`.
   - Typy proste nie są obiektami i nie mogą być zastąpione przez typ `Object` w procesie wymazywania typów.
2. **Pudełkowanie (Boxing) i Rozpakowywanie (Unboxing):**
   - Java oferuje mechanizm automatycznego pudełkowania i rozpakowywania (boxing i unboxing) dla typów prostych, co pozwala na konwersję między typami prostymi a ich odpowiednikami w postaci klas opakowujących (wrapper classes), takimi jak `Integer` dla `int`, `Double` dla `double` itp.
   - Dzięki temu, zamiast używać typów prostych w generykach, można używać ich klas opakowujących. Na przykład, zamiast `List<int>`, używa się `List<Integer>`.

3. **Kompatybilność wsteczna:**
    - Generyki zostały wprowadzone do Javy w wersji 5, z zachowaniem kompatybilności wstecznej. Aby to osiągnąć, generyki musiały być zaimplementowane w sposób, który nie wymagał zmian w maszynie wirtualnej Javy (JVM). Użycie typów prostych w generykach wymagałoby głębokich zmian w JVM.
4. **Złożoność i Wydajność:**
    - Włączenie typów prostych do systemu generyków znacząco zwiększyłoby złożoność języka i kompilatora. Ponadto, operacje na typach prostych wewnątrz generyków mogłyby być mniej wydajne ze względu na konieczność ciągłego pudełkowania i rozpakowywania.

Dziedziczenie a typy generyczne

Projekt W12, example22

```
package example22;

// Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }
```

```
    public void setFirst(T newValue) {
        first = newValue;
    }

    public void setSecond(T newValue) {
        second = newValue;
    }
}
```

```
package example22;

public class Animal {
}
```

```
package example22;

public class Dog extends Animal{
}
```

```
package example22;

public class Test22 {

    public static void main(String[] args) {
        //Pair<Animal> obj = new Pair<Dog>(); // to nie jest możliwe
        var obj2 = new Pair<Dog>();
        // reszta kodu nie jest zalecana
        Pair obj3 = obj2;
        obj3.setFirst(new Dog());
        obj3.setSecond(new Animal());
    }
}
```

Typy wieloznaczne

Termin "typ wieloznaczny" (ang. wildcard type) odnosi się do typów generycznych, które nie są dokładnie określone, czyli używają symbolu zapytania `?` jako zastępczego oznaczenia typu. Typy wieloznaczne pozwalają na większą elastyczność w definiowaniu i wykorzystywaniu generycznych struktur danych i metod, ponieważ mogą reprezentować szeroki zakres różnych typów.

Rodzaje Typów Wieloznacznych:

1. **Nieograniczony Typ Wieloznaczny (`?`):**
   - Oznacza dowolny typ. Na przykład, `List<?>` może być listą dowolnego typu obiektów.
2. **Ograniczony Górnie Typ Wieloznaczny (`? extends T`):**
   - Ogranicza typ do klasy `T` lub dowolnej jej podklasy. Na przykład, `List<? extends Number>` może być listą obiektów typu `Number` lub dowolnego typu, który jest podklasą `Number` (jak `Integer` czy `Double`).
3. **Ograniczony Dolnie Typ Wieloznaczny (`? super T`):**
   - Ogranicza typ do klasy `T` lub dowolnej jej nadklasy. Na przykład, `List<? super Integer>` może być listą obiektów typu `Integer` lub dowolnego typu, który jest nadklasą `Integer` (jak `Number` czy `Object`).

Przykład prosty

Projekt W12, example23

```
package example23;
// Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }
```

```
    public T getSecond() {
        return second;
    }

    public void setFirst(T newValue) {
        first = newValue;
    }

    public void setSecond(T newValue) {
        second = newValue;
    }
}
```

```
package example23;

public class Animal {

    @Override
    public String toString() {
        return getClass().getSimpleName();
    }
}
```

```
package example23;

public class Dog extends Animal{
}
```

```
package example23;

public class Test23 {

    public static void main(String[] args) {

        Pair<Animal> animals = new Pair<>(new Animal(), new Animal());
        printAnimals(animals);
        Pair<Dog> dogs = new Pair<>(new Dog(), new Dog());
        //printAnimals(dogs);
        printAnimalsFix(animals);
        printAnimalsFix(dogs);
```

```
        printAnimalsFix2(animals);
        printAnimalsFix2(dogs);
        printAnimalsFix3(animals);
        printAnimalsFix3(dogs);

    }

    public static void printAnimals(Pair<Animal> animals) {
        System.out.println(animals.getFirst().toString() + " " + animals.getSecond().toStr
    }

    public static void printAnimalsFix(Pair<? extends Animal> animals) {
        System.out.println(animals.getFirst().toString() + " " + animals.getSecond().toStr
    }

    public static void printAnimalsFix2(Pair<? super Dog> animals) {
        System.out.println(animals.getFirst().toString() + " " + animals.getSecond().toStr
    }

    public static void printAnimalsFix3(Pair<?> animals) {
        System.out.println(animals.getFirst().toString() + " " + animals.getSecond().toStr
    }
}
```

Przykład zaawansowany

Projekt W12, example24

```
package example24;

// Cay S. Horstmann, Java. Podstawy. Wydanie XII , Wyd. Helion, 2021.
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }
```

```
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public T getSecond() {
        return second;
    }

    public void setFirst(T newValue) {
        first = newValue;
    }

    public void setSecond(T newValue) {
        second = newValue;
    }
}
```

```
package example24;

public class Person implements Comparable<Person>{

    private String name;
    private int age;

    public Person(String name, int age) {
        if (name == null || name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be null or blank");
        }
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        this.name = name;
        this.age = age;
    }

    public String getName() {
```

```
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        this.age = age;
    }

    public void setName(String name) {
        if (name == null || name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be null or blank");
        }
        this.name = name;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + ": name=" + name + ", age=" + age;
    }

    @Override
    public int compareTo(Person o) {
        int base = this.name.compareTo(o.name);
        if (base != 0) {
            return base;
        }
        return Integer.compare(this.age, o.age);
    }
}
```

```
package example24;

public class Student extends Person implements Comparable<Person>{

    private int studentId;
```

```
    public Student(String name, int age, int studentId) {
        super(name, age);
        if (studentId <10000 || studentId > 999999) {
            throw new IllegalArgumentException("Wrong student ID");
        }
        this.studentId = studentId;
    }

    public int getStudentId() {
        return studentId;
    }

    public void setStudentId(int studentId) {
        if (studentId <10000 || studentId > 999999) {
            throw new IllegalArgumentException("Wrong student ID");
        }
        this.studentId = studentId;
    }

    @Override
    public String toString() {
        return super.toString() + ", studentId=" + studentId;
    }

    @Override
    public int compareTo(Person o) {
        if (o instanceof Student) {
            Student student = (Student) o;
            int base = super.compareTo(student);
            if (base != 0) {
                return base;
            }
            return Integer.compare(this.studentId, student.studentId);
        }
        return super.compareTo(o);
    }
}
```

```
package example24;

public class Test24 {
```

```
public static void main(String[] args) {
    Person[] people = new Person[4];
    people[0] = new Person("John", 20);
    people[1] = new Person("John", 30);
    people[2] = new Person("Adam", 20);
    people[3] = new Person("Adam", 16);
    System.out.println("Case 1");
    Pair<Person> pair = minmaxOld(people);
    System.out.println(pair.getFirst());
    System.out.println(pair.getSecond());
    Student[] students = new Student[4];
    students[0] = new Student("John", 20, 125478);
    students[1] = new Student("John", 30, 122278);
    students[2] = new Student("Adam", 20, 125433);
    students[3] = new Student("Adam", 16, 165478);
    //Pair<Student> pair2 = minmaxBad(students); // to nie jest możliwe
    System.out.println("Case 2");
    Pair<Student> pair2 = minmax(students);
    System.out.println(pair2.getFirst());
    System.out.println(pair2.getSecond());
    Person[] people2 = new Person[6];
    people2[0] = new Person("John", 20);
    people2[1] = new Person("John", 30);
    people2[2] = new Person("Adam", 20);
    people2[3] = new Student("John", 20, 125478);
    people2[4] = new Student("John", 30, 122278);
    people2[5] = new Student("Adam", 20, 125433);
    System.out.println("Case 3");
    Pair<Person> pair3 = minmaxOld(people2);
    System.out.println(pair3.getFirst());
    System.out.println(pair3.getSecond());
    System.out.println("Case 4");
    Pair<Person> pair4 = minmax(people2);
    System.out.println(pair4.getFirst());
    System.out.println(pair4.getSecond());
}

public static <T extends Comparable<T>> Pair<T> minmaxOld(T[] a) {
    if (a == null || a.length == 0) {
        return null;
    }
```

```
        T min = a[0];
        T max = a[0];
        for (int i=1; i<a.length; i++) {
            if (min.compareTo(a[i]) > 0) {
                min = a[i];
            }
            if (max.compareTo(a[i]) < 0) {
                max = a[i];
            }
        }
        return new Pair<>(min, max);
    }

    public static <T extends Comparable<? super T>> Pair<T> minmax(T[] a) {
        if (a == null || a.length == 0) {
            return null;
        }
        T min = a[0];
        T max = a[0];
        for (int i=1; i<a.length; i++) {
            if (min.compareTo(a[i]) > 0) {
                min = a[i];
            }
            if (max.compareTo(a[i]) < 0) {
                max = a[i];
            }
        }
        return new Pair<>(min, max);
    }
}
```