Programowanie obiektowe w Javie - teoria 5

Interfejs `Comparable`

Domyślny sposób

W Javie, interfejs `Comparable` to generyczny interfejs, który umożliwia porównywanie obiektów jednego typu ze sobą. Głównym celem implementacji interfejsu Comparable jest umożliwienie sortowania zbiorów obiektów tego samego typu.

Interfejs `Comparable` znajduje się w pakiecie `java.lang` i zawiera jedną metodę, która musi być zaimplementowana przez klasę, która go implementuje:

```
public int compareTo(T o);
```

Tutaj `T` oznacza typ obiektów, które mają być porównywane. Metoda `compareTo` zwraca wartość liczb całkowitych:

1. Ujemna wartość (mniejsza niż 0), gdy obiekt jest mniejszy niż obiekt przekazany jako argument (`o`).
2. Wartość zero (0), gdy obiekt jest równy obiektowi przekazanemu jako argument (`o`).
3. Dodatnia wartość (większa niż 0), gdy obiekt jest większy niż obiekt przekazany jako argument (`o`).

Przykład implementacji interfejsu Comparable dla klasy `Person`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

```
    @Override
    public int compareTo(Person other) {
        // Porównanie wieku osób
        if (this.age < other.age) {
            return -1;
        } else if (this.age > other.age) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

W powyższym przykładzie, obiekty klasy `Person` są porównywane na podstawie ich wieku. Dzięki temu, możemy sortować listę tablicową osób według wieku:

```
ArrayList<Person> people = new ArrayList<>();
people.add(new Person("Anna", 28));
people.add(new Person("John", 11));
people.add(new Person("Michael", 22));

people.sort(null);
```

Teraz lista `people` jest posortowana rosnąco według wieku.

Implementując interfejs `Comparable`, można określić naturalny porządek dla obiektów klasy, co pozwala na sortowanie i porównywanie tych obiektów w łatwy sposób.

Inny sposób

Możemy zamienić na odejmowanie w metodzie `compareTo`. Oto zmieniona wersja klasy `Person`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

```
    @Override
    public int compareTo(Person other) {
        // Porównanie wieku osób używając odejmowania
        return this.age - other.age;
    }
}
```

W powyższym przykładzie, metoda `compareTo` zwraca wynik odejmowania wieku bieżącego obiektu `Person` (`this.age`) od wieku obiektu przekazanego jako argument (`other.age`). Jeśli różnica jest ujemna, oznacza to, że bieżący obiekt jest młodszy niż obiekt porównywany. Jeśli różnica wynosi zero, oznacza to, że oba obiekty mają ten sam wiek. Jeśli różnica jest dodatnia, oznacza to, że bieżący obiekt jest starszy niż obiekt porównywany.

Proszę jednak zauważyć, że używanie odejmowania w porównaniu może prowadzić do błędów wynikających z przekroczenia zakresu wartości typu `int` (integer overflow). Jeśli wartości liczbowe są bardzo duże i różnica pomiędzy nimi przekracza zakres dopuszczalnych wartości typu `int`, wynik porównania może być niepoprawny. W takich przypadkach lepiej korzystać z metody `Integer.compare`, która nie jest narażona na ten problem.

Inny sposób 2

Możemy zmodyfikować metodę `compareTo` w klasie `Person`, aby użyć statycznej metody `compare` z klasy `Integer`, która pozwala na domyślne porównanie wartości typu `int`. Oto zmieniona wersja klasy `Person`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    @Override
    public int compareTo(Person other) {
        // Porównanie wieku osób używając metody statycznej 'compare' z klasy 'Integer'
        return Integer.compare(this.age, other.age);
    }
}
```

Dwa kryteria

Aby porównywać obiekty po dwóch kryteriach, musisz zaimplementować logikę porównywania w metodzie `compareTo`. Najpierw porównujemy obiekty według pierwszego kryterium, a jeśli są równe, porównujemy według drugiego kryterium.

Załóżmy, że chcemy porównywać obiekty klasy `Person` najpierw według wieku, a następnie według imienia, gdy wiek jest taki sam. Oto zmodyfikowana wersja klasy `Person`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    @Override
    public int compareTo(Person other) {
        // Porównanie wieku osób używając metody 'compare' z klasy 'Integer'
        int ageComparison = Integer.compare(this.age, other.age);

        // Jeśli wiek jest taki sam, porównujemy imiona
        if (ageComparison == 0) {
            // Porównanie imion używając metody 'compareTo' z klasy 'String'
            return this.name.compareTo(other.name);
        }

        // W przeciwnym razie zwracamy wynik porównania wieku
        return ageComparison;
    }
}
```

W powyższym przykładzie, metoda `compareTo` najpierw porównuje wiek obiektów używając metody `Integer.compare`. Jeśli wiek obiektów jest taki sam (wynik porównania wynosi 0), porównuje imiona używając metody `compareTo` z klasy `String`. W przeciwnym razie, metoda zwraca wynik porównania wieku.

Interfejs `Comparable` a dziedziczenie

W kontekście dziedziczenia, można zaimplementować generyczny interfejs `Comparable` w klasie bazowej, aby umożliwić porównywanie obiektów w klasach dziedziczących. Kluczową kwestią jest definiowanie metody `compareTo` w taki sposób, aby była wystarczająco ogólna dla wszystkich klas dziedziczących. Możemy osiągnąć to, używając metod abstrakcyjnych, które będą zaimplementowane przez klasy dziedziczące.

Rozważmy przykład klasy bazowej `Animal` oraz klas dziedziczących `Dog` i `Cat`. Chcemy, aby obiekty tych klas były porównywane według ich masy.

1. Zdefiniuj klasę bazową `Animal` z abstrakcyjną metodą `getWeight` i implementacją interfejsu `Comparable<Animal>`:

```
public abstract class Animal implements Comparable<Animal> {

    protected abstract int getWeight();

    @Override
    public int compareTo(Animal other) {
        // Porównanie zwierząt według masy
        return Integer.compare(this.getWeight(), other.getWeight());
    }
}
```

2. Zdefiniuj klasę `Dog` dziedziczącą po klasie `Animal`:

```
class Dog extends Animal {
    private int weight;

    public Dog(int weight) {
        this.weight = weight;
    }

    @Override
    protected int getWeight() {
        return this.weight;
    }
}
```

3. Zdefiniuj klasę `Cat` dziedziczącą po klasie `Animal`:

```
class Cat extends Animal {
    private int weight;

    public Cat(int weight) {
        this.weight = weight;
    }

    @Override
    protected int getWeight() {
        return this.weight;
    }
}
```

Teraz, gdy mamy zdefiniowane klasy `Dog` i `Cat` dziedziczące po klasie `Animal`, możemy porównywać i sortować obiekty tych klas według ich masy, korzystając z metody `compareTo` zaimplementowanej w klasie bazowej `Animal`:

```
ArrayList<Animal> animals = new ArrayList<>();
animals.add(new Dog(30));
animals.add(new Cat(10));
animals.add(new Dog(20));

Collections.sort(animals);
```

W powyższym przykładzie, lista `animals` zostanie posortowana rosnąco według masy zwierząt. Dzięki zastosowaniu interfejsu `Comparable` w klasie bazowej `Animal`, zapewniamy, że wszystkie klasy dziedziczące będą miały zaimplementowaną metodę `compareTo`.

Inny sposób

```
class Vehicle implements Comparable<Vehicle> {
    private int maxSpeed;

    public Vehicle(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }
```

```
    @Override
    public int compareTo(Vehicle other) {
        return Integer.compare(this.maxSpeed, other.maxSpeed);
    }
}
```

Aby wymusić implementację interfejsu `Comparable<Vehicle>` w klasie `Car`, należy dodać generyczny typ parametru do klasy `Comparable`:

```
class Car implements Comparable<Vehicle> {
    private int maxSpeed;
    private String brand;

    public Car(int maxSpeed, String brand) {
        this.maxSpeed = maxSpeed;
        this.brand = brand;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public String getBrand() {
        return brand;
    }

    @Override
    public int compareTo(Vehicle other) {
        // Porównanie samochodów według prędkości maksymalnej
        if (other instanceof Car) {
            Car otherCar = (Car) other;
            return Integer.compare(this.maxSpeed, otherCar.maxSpeed);
        }
        return -1;
    }
}
```

W powyższym przykładzie, klasa `Vehicle` nadal implementuje interfejs `Comparable<Vehicle>`, a klasa `Car` implementuje ten sam interfejs z parametrem generycznym `Vehicle`. W metodzie `compareTo` klasy `Car`, porównujemy samochody tylko wtedy, gdy przekazany obiekt jest instancją klasy `Car`.

Dzięki takiemu rozwiązaniu, możemy porównywać obiekty typu `Vehicle` w ogólny sposób, a

jednocześnie wymuszać implementację porównywania obiektów klasy `Car` w bardziej szczegółowy sposób.

Porównywanie napisów

Oto przykład implementacji interfejsu `Comparable` w wersji generycznej dla klasy `Person`, gdzie porównywanie obiektów jest oparte na polu `name` typu `String`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
}
```

W powyższym przykładzie, klasa `Person` implementuje interfejs generyczny `Comparable` z parametrem typu `Person`. W metodzie `compareTo` porównujemy obiekty klasy `Person` na podstawie pola `name` typu `String` przy użyciu metody `compareTo` klasy `String`. Dzięki temu, możemy porównywać obiekty klasy `Person` na podstawie wartości w polu `name` w sposób naturalny dla typu `String`. Możemy to wykorzystać, na przykład, do sortowania listy obiektów typu `Person` w porządku alfabetycznym według ich imion.

Porównywanie w typie `Double`

```
class Product implements Comparable<Product> {
    private String name;
    private Double price;

    public Product(String name, Double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public Double getPrice() {
        return price;
    }

    @Override
    public int compareTo(Product other) {
        return this.price.compareTo(other.price);
    }
}
```

Porównywanie w typie `Date`

Oto przykład implementacji interfejsu `Comparable` w wersji generycznej dla klasy `Employee`, gdzie porównywanie obiektów jest oparte na polu `hireDate` typu `Date`:

```
import java.util.Date;

public class Employee implements Comparable<Employee> {
    private String name;
    private Date hireDate;

    public Employee(String name, Date hireDate) {
        this.name = name;
        this.hireDate = hireDate;
    }
```

```
    public String getName() {
        return name;
    }

    public Date getHireDate() {
        return hireDate;
    }

    @Override
    public int compareTo(Employee other) {
        return this.hireDate.compareTo(other.hireDate);
    }
}
```

W powyższym przykładzie, klasa `Employee` implementuje interfejs generyczny `Comparable` z parametrem typu `Employee`. W metodzie `compareTo` porównujemy obiekty klasy `Employee` na podstawie pola `hireDate` typu `Date` przy użyciu metody `compareTo` klasy `Date`. Dzięki temu, możemy porównywać obiekty klasy `Employee` na podstawie wartości w polu `hireDate` w sposób naturalny dla typu `Date`. Możemy to wykorzystać, na przykład, do sortowania listy obiektów typu `Employee` w porządku rosnącym według ich dat zatrudnienia.

Porównywanie w typie `LocalDate`

Oto przykład implementacji interfejsu `Comparable` w wersji generycznej dla klasy `Task`, gdzie porównywanie obiektów jest oparte na polu `deadline` typu `LocalDate`:

```
import java.time.LocalDate;

public class Task implements Comparable<Task> {
    private String title;
    private LocalDate deadline;

    public Task(String title, LocalDate deadline) {
        this.title = title;
        this.deadline = deadline;
    }

    public String getTitle() {
        return title;
    }
```

```
    public LocalDate getDeadline() {
        return deadline;
    }

    @Override
    public int compareTo(Task other) {
        return this.deadline.compareTo(other.deadline);
    }
}
```

W powyższym przykładzie, klasa `Task` implementuje interfejs generyczny `Comparable` z parametrem typu `Task`. W metodzie `compareTo` porównujemy obiekty klasy `Task` na podstawie pola `deadline` typu `LocalDate` przy użyciu metody `compareTo` klasy `LocalDate`. Dzięki temu, możemy porównywać obiekty klasy `Task` na podstawie wartości w polu `deadline` w sposób naturalny dla typu `LocalDate`. Możemy to wykorzystać, na przykład, do sortowania listy obiektów typu `Task` w porządku rosnącym według ich terminów.

Przykład sortowania listy tablicowej `tasks` zawierającej obiekty typu `Task` na podstawie pola `deadline`:

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        ArrayList<Task> tasks = new ArrayList<>();
        tasks.add(new Task("Write report", LocalDate.of(2022, 4, 30)));
        tasks.add(new Task("Prepare presentation", LocalDate.of(2022, 5, 15)));
        tasks.add(new Task("Read book", LocalDate.of(2022, 6, 1)));

        // Sortowanie listy na podstawie pola deadline
        Collections.sort(tasks);

        // Wyświetlenie posortowanej listy
        for (Task task : tasks) {
            System.out.println(task.getTitle() + " - " + task.getDeadline());
        }
    }
}
```

W powyższym przykładzie, lista `tasks` jest sortowana przy użyciu metody `sort` z klasy `Collections`, która wykorzystuje metodę `compareTo` klasy `Task` do porównywania obiektów

na podstawie pola `deadline`. Następnie posortowana lista jest wyświetlana na ekranie. W przeciwieństwie do poprzedniego przykładu, użyliśmy tu klasy `ArrayList` zamiast interfejsu `List`.

Interfejs `Comparator`

W Javie istnieje również generyczny interfejs `Comparator<T>`, który służy do porównywania obiektów klasy `T` na podstawie określonego kryterium. W przeciwieństwie do interfejsu `Comparable`, który wymaga implementacji metody `compareTo` w klasie porównywanej, interfejs `Comparator` pozwala na zdefiniowanie własnej metody porównującej obiekty danej klasy.

Interfejs `Comparator` zawiera jedną metodę o nazwie `compare`, która przyjmuje dwa argumenty typu `T` i zwraca wartość typu `int`. Metoda ta porównuje dwa obiekty klasy `T` i zwraca:

- wartość mniejszą od zera, jeśli pierwszy argument jest mniejszy niż drugi,
- zero, jeśli oba argumenty są równe,
- wartość większą od zera, jeśli pierwszy argument jest większy niż drugi.

Dzięki temu interfejsowi możemy określić dowolną liczbę metod porównujących dla danej klasy, co jest przydatne w przypadku, gdy chcemy sortować obiekty tej klasy w różny sposób w zależności od kontekstu.

Poniżej przedstawiony jest przykład zastosowania interfejsu `Comparator` do sortowania obiektów klasy `Person` na podstawie pola `age`:

```
import java.util.ArrayList;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        ArrayList<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 25));
        people.add(new Person("Bob", 30));
        people.add(new Person("Charlie", 20));

        // Sortowanie listy na podstawie pola age z użyciem Comparator
        people.sort(new AgeComparator());

        // Wyświetlenie posortowanej listy
        for (Person person : people) {
            System.out.println(person.getName() + " - " + person.getAge());
        }
```

```
    }
}

class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}
```

W powyższym przykładzie, klasa `Person` reprezentuje osobę z polami `name` i `age`. Następnie tworzymy klasę `AgeComparator`, która implementuje interfejs `Comparator<Person>` i definiuje metodę `compare` porównującą dwa obiekty klasy `Person` na podstawie ich wieku. W metodzie `main` tworzymy listę obiektów klasy `Person` i sortujemy ją przy użyciu obiektu klasy `AgeComparator`. Ostatecznie posortowana lista jest wyświetlana na ekranie.

Interfejs `Comparator` umożliwia również łą…łączenie wielu kryteriów sortowania za pomocą metody `thenComparing`, która umożliwia określenie kolejnych kryteriów sortowania w przypadku, gdy dwa obiekty mają takie same wartości w pierwszym kryterium. Metoda ta zwraca nowy obiekt klasy `Comparator`, który łączy dwa kryteria sortowania.

Poniżej przedstawiony jest przykład zastosowania metody `thenComparing` do sortowania obiektów klasy `Person` na podstawie pola `age`, a następnie na podstawie pola `name`:

```java
import java.util.ArrayList;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        ArrayList<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 25));
        people.add(new Person("Bob", 30));
        people.add(new Person("Charlie", 20));
        people.add(new Person("Alice", 30));

        // Sortowanie listy na podstawie pola age i name z użyciem Comparator
        people.sort(new AgeComparator().thenComparing(new NameComparator()));

        // Wyświetlenie posortowanej listy
        for (Person person : people) {
            System.out.println(person.getName() + " - " + person.getAge());
        }
    }
}

class Person {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

class AgeComparator implements Comparator<Person> {
    @Override
```

```
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
}

class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

W powyższym przykładzie, klasa `Person` reprezentuje osobę z polami `name` i `age`. Następnie tworzymy dwie klasy `AgeComparator` i `NameComparator`, które implementują interfejs `Comparator<Person>` i definiują odpowiednie metody `compare` porównujące dwa obiekty klasy `Person` na podstawie wieku i imienia. W metodzie `main` tworzymy listę obiektów klasy `Person` i sortujemy ją przy użyciu obiektu klasy `AgeComparator`, a następnie wywołujemy metodę `thenComparing` i przekazujemy obiekt klasy `NameComparator`, który definiuje kolejne kryterium sortowania. Ostatecznie posortowana lista jest wyświetlana na ekranie.

Różnice

W Javie istnieją dwa generyczne interfejsy, które umożliwiają porównywanie obiektów: `Comparable` i `Comparator`.

Główne różnice między nimi to:

1. Implementacja interfejsu `Comparable` wymaga zdefiniowania metody `compareTo` w klasie porównywanej, podczas gdy implementacja interfejsu `Comparator` polega na utworzeniu osobnej klasy porównującej, która implementuje interfejs `Comparator`.
2. Interfejs `Comparable` pozwala na naturalne porównanie obiektów klasy zgodnie z ich wewnętrzną kolejnością, podczas gdy interfejs `Comparator` pozwala na dowolne porównanie obiektów na podstawie określonego kryterium.
3. Implementacja interfejsu `Comparable` jest przydatna w przypadku, gdy chcemy sortować obiekty klasy w sposób naturalny, natomiast interfejs `Comparator` umożliwia określenie niestandardowych kryteriów sortowania.
4. Klasa implementująca interfejs `Comparable` jest automatycznie sortowalna przy użyciu metody `sort` z klasy `Collections`, podczas gdy w przypadku użycia interfejsu `Comparator` należy przekazać obiekt klasy porównującej do metody `sort`.

Przykładowo, jeśli chcemy sortować obiekty klasy `Person` na podstawie pola `age`, możemy wykorzystać zarówno interfejs `Comparable`, jak i `Comparator`:

1. Implementacja interfejsu `Comparable`:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

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

2. Implementacja interfejsu `Comparator`:

```
class PersonAgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
}
```

Obydwa rozwiązania umożliwiają sortowanie obiektów klasy `Person` na podstawie pola `age`. W przypadku użycia interfejsu `Comparable`, sortowanie jest realizowane automatycznie przez metodę `sort` z klasy `Collections`. W przypadku użycia interfejsu `Comparator`, należy utworzyć obiekt klasy `PersonAgeComparator` i przekazać go do metody `sort`.

Różnice a dziedziczenie

Różnice w dziedziczeniu między interfejsem `Comparable` i `Comparator` wynikają z faktu, że interfejs `Comparable` jest implementowany bezpośrednio w klasie, podczas gdy interfejs `Comparator` jest implementowany w osobnej klasie porównującej.

W przypadku interfejsu `Comparable` dziedziczenie polega na tym, że klasa dziedzicząca implementuje interfejs `Comparable` i jednocześnie korzysta z metod porównujących zdefiniowanych w klasie bazowej. Dzięki temu, klasy dziedziczące dziedziczą również zachowanie dotyczące porównywania obiektów.

W przypadku interfejsu `Comparator` dziedziczenie polega na tym, że klasa dziedzicząca korzysta z obiektu klasy porównującej zdefiniowanego w klasie bazowej. Dzięki temu, klasy dziedziczące nie muszą implementować interfejsu `Comparator`, lecz korzystają z istniejącej implementacji.

Przykładowo, jeśli mamy klasę `Vehicle` implementującą interfejs `Comparable<Vehicle>` oraz klasę `Car` dziedziczącą po klasie `Vehicle`, to klasa `Car` dziedziczy również implementację metody `compareTo` z klasy `Vehicle`. Można jednak nadpisać tę metodę w klasie `Car`, aby zmienić zachowanie porównywania obiektów.

Natomiast jeśli mamy klasę `VehicleComparator` implementującą interfejs `Comparator<Vehicle>` oraz klasę `Car` dziedziczącą po klasie `Vehicle`, to klasa `Car` może korzystać z obiektu klasy `VehicleComparator`, ale nie musi implementować interfejsu `Comparator`. Można również utworzyć nową klasę porównującą dziedziczącą po klasie `VehicleComparator`, aby zmienić zachowanie porównywania obiektów.

Różnice w tablicach

W przypadku tablic w Javie, implementacja interfejsu `Comparable` dotyczy elementów wewnątrz tablicy, podczas gdy implementacja interfejsu `Comparator` dotyczy samej tablicy.

Jeśli chodzi o interfejs `Comparable`, to każdy typ elementów w tablicy musi implementować interfejs `Comparable` i zawierać metodę `compareTo`, która porównuje dwa elementy. Następnie sortowanie tablicy może być wykonane za pomocą metody `Arrays.sort`, która automatycznie wykorzystuje implementację metody `compareTo`.

```
public class Main {
    public static void main(String[] args) {
        Integer[] arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

W powyższym przykładzie sortujemy tablicę `arr` typu `Integer[]` przy użyciu metody `Arrays.sort`. Ponieważ klasa `Integer` implementuje interfejs `Comparable`, metoda `sort` wykorzystuje implementację metody `compareTo` dla klasy `Integer`.

Jeśli chodzi o interfejs `Comparator`, to należy utworzyć osobną klasę porównującą, która implementuje interfejs `Comparator` i zawiera metodę `compare`, która porównuje dwa elementy. Następnie sortowanie tablicy może być wykonane za pomocą metody `Arrays.sort`, której jako drugi argument przekazujemy obiekt klasy porównującej.

```
public class Main {
    public static void main(String[] args) {
        Integer[] arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
        Arrays.sort(arr, new IntegerComparator());
        System.out.println(Arrays.toString(arr));
    }
}

class IntegerComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}
```

W powyższym przykładzie sortujemy tablicę `arr` typu `Integer[]` przy użyciu metody `Arrays.sort` i obiektu klasy `IntegerComparator`, która implementuje interfejs `Comparator`. W metodzie `main` przekazujemy obiekt klasy `IntegerComparator` jako drugi argument do metody `sort`, co pozwala na sortowanie tablicy zgodnie z porównaniem zdefiniowanym w metodzie `compare`.