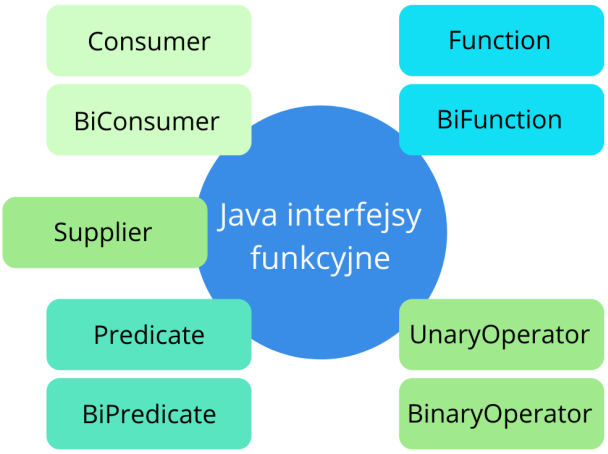
Notatki - Wbudowane Interfejsy Funkcyjne

Spis treści

Wbudowane interfejsy funkcyjne

We wcześniejszych przykładach rozmawialiśmy o własnych interfejsach funkcyjnych oraz o tym jakie warunki interface musi spełniać aby był uznany za interface funkcyjny.

Twórcy Javy starali się wyjść użytkownikom na przeciw i przewidzieć typowe interfejsy funkcyjne, które programista musiałby napisać sam. Dlatego API Javy daje nam kilka interfejsów funkcyjnych, które możemy wykorzystywać.



Obraz 1. Interfejsy funkcyjne, które oferuje Java

Wszystkie wymienione interfejsy zostaną omówione poniżej.

Predicate

Predicate, to interfejs funkcyjny, który dostarcza nam metodę `test()`, która przyjmuje dowolny obiekt jako argument i zwraca `boolean`. Stosujemy go najczęściej, gdy chcemy "odfiltrować" jakąś wartość.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class LambdaExample {

    public static void main(String[] args) {
        List<Animal> animals = List.of(
            new Animal("rabbit"),
            new Animal("dog"),
            new Animal("bird")
        );
        print(animals, a -> "rabbit".equals(a.getName())); ①
    }

    private static void print(List<Animal> animals, Predicate<Animal> checker) {
        for (Animal animal : animals) {
            if (checker.test(animal)) {
                System.out.println(checker.toString() + ": " + animal);
            }
        }
    }
}
```

`Predicate` przyjmuje typ generyczny, który jest parametrem wejściowym metody `test()`. Typem zwracanym metody `test()` jest `boolean`. Określając lambdę, która implementuje ten interface musimy podać jeden argument, którego typ jest określony typem generycznym, natomiast typem zwracanym tej lambdy musi być boolean. Oczywiście możemy też zastosować **method reference**.

Kolejne wykorzystanie `Predicate` w praktyce:

```
public class ExamplePredicate {

    public static void main(String[] args) {
        Predicate<String> predicate1 = someString -> someString.isEmpty(); ①
        Predicate<String> predicate2 = String::isEmpty; ②
        System.out.println(predicate1.test("zajavka"));
        System.out.println(predicate2.test("zajavka is cool"));
```

```
        String someValue = Optional.of("zajavka is the coolest")
            // zamiast .filter(someString -> someString.isEmpty()) ③
            .filter(String::isEmpty) ④
            .orElse("nonEmpty");
    }

    private static void voidMethod() {}

    interface SomeInterface {
        void someMethod(String abc);
    }
}
```

① Implementacja `Predicate` wykorzystując lambdę.

② Implementacja `Predicate` wykorzystując **method reference**, która robi to samo co przykład 1.

③ Implementacja `Predicate` wykorzystując lambdę w metodzie `Optional.filter()`.

④ Implementacja `Predicate` wykorzystując **method reference**, która robi to samo co przykład 3.

Consumer

Consumer, to interfejs funkcyjny, który dostarcza nam metodę `accept()`, która przyjmuje dowolny obiekt jako argument i nic nie zwraca. Stosujemy go najczęściej, gdy chcemy "skonsumować" jakąś wartość i nic nie zwrócić.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleConsumer {

    public static void main(String[] args) {
        Consumer<String> consumer1 = value -> {return;}; ①
        //System.out.println(consumer1.accept("ain't gonna work")); ②

        Consumer<String> consumer2 = value -> {
            System.out.println("printing: " + value);
            return;
        }; ③
        consumer2.accept("this shall work"); ④

        Consumer<String> consumer3 = value -> someMethod(); ⑤
        consumer3.accept("this shall work"); ⑥

        Consumer<String> consumer4 = value -> {
```

```
            someMethod2();
            return;
        }; ⑦
        consumer4.accept("this shall work"); ⑧

        Consumer<String> consumer5 = value -> someMethod2(); ⑨
        consumer5.accept("this shall work"); ⑩

        Consumer<String> consumer6 = ExampleConsumer::someMethodReference; ⑪
        consumer6.accept("this shall work"); ⑫
    }

    private static void someMethod() {}

    private static boolean someMethod2() {
        return true;
    }

    private static void someMethodReference(String arg) {}
}
```

① Implementacja interfejsu przy wykorzystaniu lambdy, możemy zastosować tutaj słówko `return` na takiej samej zasadzie jak robimy to w metodach zwracających `void`.

② Wywołanie, które same w sobie jest poprawne, natomiast nie możemy przekazać rezultatu tego wywołania do metody `println()`, gdyż wywołanie zwraca `void`.

③ Implementacja interfejsu przy wykorzystaniu lambdy, gdzie najpierw drukujemy tekst na ekranie, a potem wychodzimy z lambdy jak w przykładzie 1.

④ Poprawne wywołanie metody.

⑤ Implementacja interfejsu przy wykorzystaniu lambdy, możemy zapisać to w ten sposób, zwyczajnie w ciele lambdy wywołujemy metodę, która nic nie przyjmuje, a zwraca `void`.

⑥ Poprawne wywołanie metody.

⑦ Implementacja interfejsu przy wykorzystaniu lambdy, możemy to zapisać w ten sposób, co jest trochę analogiczne do przykładu 5, na koniec wychodzimy słówkiem `return` jak w przykładzie 1.

⑧ Poprawne wywołanie metody.

⑨ Implementacja interfejsu przy wykorzystaniu lambdy, natomiast zapis ten może wydawać się nieintuicyjny ze względu na to, że metoda `someMethod2()` zwraca `true`. Ten zapis jest jednak analogiczny do zapisu 7, zatem jest poprawny.

⑩ Poprawne wywołanie metody.

⑪ Zastosowanie mechanizmu method reference do implementacji interfejsu.

⑫ Poprawne wywołanie metody.

Supplier

Supplier, to interfejs funkcyjny, który dostarcza nam metodę `get()`, która nic nie przyjmuje i zwraca nam obiekt dowolnego typu. Stosujemy go najczęściej, gdy chcemy "dostarczyć" jakąś wartość z niczego.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface Supplier<T> {

    public T get();

}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleSupplier {

    public static void main(String[] args) {
        Supplier<String> supplier1 = () -> "value";
        String supplied1 = supplier1.get();
        System.out.println(supplied1); ①

        //Supplier<String> supplier2 = ExampleSupplier::provideInt; ②

        Supplier<String> supplier3 = () -> ExampleSupplier.provideString();
        String supplied3 = supplier3.get();
        System.out.println(supplied3); ③

        Supplier<String> supplier4 = ExampleSupplier::provideString;
        String supplied4 = supplier4.get();
        System.out.println(supplied4); ④
    }

    private static int provideInt() {
        return 23;
    }

    private static String provideString() {
        return "23";
    }

}
```

① Implementacja `Supplier` wykorzystując lambdę. Najprostszy przykład, w którym z niczego dostarczamy `String`.

② Implementacja `Supplier` wykorzystując method reference, typ generyczny określa `String`, natomiast metoda `provideInt()` dostarcza `int`, stąd mamy błąd kompilacji.

③ Implementacja `Supplier` wykorzystując lambdę. Lambda nie przyjmuje żadnego parametru, natomiast możemy w niej wywołać metodę zwracającą `String`. Intellij sam podpowiada aby zamienić to wywołanie na to w przykładzie 4.

④ Implementacja `Supplier` wykorzystując **method reference**, przykład robi dokładnie to samo co przykład 3.

BiPredicate

Oprócz interfejsów takich jak `Predicate` lub `Consumer` zobaczymy jeszcze interfejsy z przedrostkiem `Bi`. Oznacza to najczęściej, że coś dzieje się w takim interfejsie

> podwójnie.

BiPredicate, to interfejs funkcyjny, który dostarcza nam metodę `test()`, która przyjmuje dwa parametry dowolnego typu i zwraca `boolean`. Stosujemy go najczęściej, gdy chcemy "odfiltrować" jakąś wartość przyjmując dwa argumenty.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiPredicate {

    public static void main(String[] args) {
        BiPredicate<String, Dog> biPredicate1 = (str, dog) -> dog.likes(str);
        System.out.println(biPredicate1.test("abc", new Dog())); ①

        BiPredicate<String, Dog> biPredicate2 = ExampleBiPredicate::doggoLikes;
        System.out.println(biPredicate2.test("doggo", new Dog())); ②
    }

    private static boolean doggoLikes(String string, Dog dog) {
        return string.isEmpty();
    }
}
```

```
class Dog {

    public boolean likes(final String str) {
            return "doggo".equals(str);
    }
}
```

① Implementacja `BiPredicate` wykorzystując lambdę. Wywołując zdefiniowany interface przekazujemy dwa argumenty i wartością zwracaną jest `boolean`.

② Implementacja `BiPredicate` wykorzystując method reference. Definicja metody `doggoLikes()` pasuje (dwa parametry i typ zwracany boolean) do definicji metody `test()` z interfejsu `BiPredicate`, dlatego możemy wykorzystać metodę `doggoLikes()`.

BiConsumer

BiConsumer, to interfejs funkcyjny, który dostarcza nam metodę `accept()`, która przyjmuje dwa parametry dowolnego typu i nic nie zwraca. Stosujemy go najczęściej, gdy chcemy "skonsumować" jakieś dwie wartości i nic nie zwrócić.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    void accept(T t, U u);

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiConsumer {

    public static void main(String[] args) {
        ExampleBiConsumer example = new ExampleBiConsumer();
        example.run();
    }

    private void run() {
        Cat cat = new Cat("Mruczek");
        Car car = new Car();

        BiConsumer<Cat, Car> biConsumer = (ct, cr) -> cr.enter(ct); ①

        System.out.println(car); ②
        biConsumer.accept(cat, car); ③
        System.out.println(car); ④
    }

}
```

```
class Cat {
    private final String name;

    public Cat(final String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

```
class Car {

    private Cat cat;

    public void enter(final Cat cat) {
        this.cat = cat;
    }
```

```
    @Override
    public String toString() {
        return "Car{" +
            "cat=" + cat +
            '}';
    }
}
```

① Implementacja `BiConsumer` wykorzystując lambdę, w której określamy, że `Cat` wsiada do `Car`. Lambda nic nie zwraca, konsumujemy dwa obiekty.

② Zawartość obiektu klasy `Car` bez kota w środku.

③ Kot wsiada do samochodu.

④ Zawartość obiektu klasy `Car` z kotem w środku.

BiSupplier

Ten jest szybki bo nie ma czegoś takiego 😁.

Function

Function, to interfejs funkcyjny, który dostarcza nam metodę `apply()`, która przyjmuje parametr dowolnego typu i zwraca parametr dowolnego typu. Stosujemy go najczęściej, gdy chcemy przekształcić jeden obiekt w drugi.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    // reszta metod
}
```

Warto zwrócić tutaj uwagę, że pierwszy typ generyczny oznacza przyjmowany przez metodę `apply()` parametr, natomiast drugi typ generyczny oznacza typ zwracany z metody `apply()`.

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleFunction {

    public static void main(String[] args) {
        ExampleFunction example = new ExampleFunction();
        example.run();
    }

    private void run() {
        Function<String, Cat> function1 = name -> new Cat(name); ①
        Function<String, Cat> function2 = Cat::new; ②
        System.out.println(function2.apply("mruczek"));
```

```
        Function<String, Integer> function3 = input -> input.length(); ③
        Function<String, Integer> function4 = String::length; ④
        System.out.println(function4.apply("this string length"));

        Function<String, Integer> function5 = s -> s.length() + 123; ⑤
        System.out.println(function5.apply("abc"));
    }
}
```

```
class Cat {
    private final String name;

    public Cat(final String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

① Implementacja `Function` wykorzystując lambdę. Wywoływany jest tutaj konstruktor, w którym tworzymy obiekt klasy `Cat` na podstawie przekazanego `Stringa`.

② Implementacja `Function` wykorzystując method reference, która robi to samo co linijka 1.

③ Implementacja `Function` wykorzystując lambdę. Wywoływana jest tutaj metoda `length()` z klasy String, dlatego definicja `Function<String, Integer>` określa najpierw `String`, bo jest to typ wejściowy do metody `apply()`, a potem `Integer` bo jest to typ zwracany z tej metody.

④ Implementacja `Function` wykorzystując method reference, która robi to samo co linijka 3.

⑤ Implementacja `Function` wykorzystując lambdę. W tym przypadku lambda wyciąga długość przekazanego `Stringa` i dodaje do niej `123`. Nie jest możliwe zapisanie tego samego przy wykorzystaniu method reference.

BiFunction

BiFunction, to interfejs funkcyjny, który dostarcza nam metodę `apply()`, która przyjmuje dwa parametry dowolnego typu i zwraca parametr dowolnego typu. Stosujemy go najczęściej, gdy chcemy przekształcić dwa obiekty w jakiś trzeci obiekt.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    // reszta metod
```

```
}
```

Warto zwrócić tutaj uwagę, że pierwsze dwa typy generyczne oznaczają przyjmowane przez metodę `apply()` parametry, natomiast trzeci typ generyczny oznacza typ zwracany z metody `apply()`.

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBiFunction {

    public static void main(String[] args) {
        BiFunction<String, Integer, Long> function1 = (a, b) -> (long) (a.length() + b);
        System.out.println(function1.apply("string1", 10)); ①

        BiFunction<String, String, Dog> function2 = (a, b) -> new Dog(a + b);
        System.out.println(function2.apply("mruczek", "mruczaasty")); ②

        BiFunction<String, String, Dog> function3 = Dog::new;
        System.out.println(function3.apply("kiciek","kiciasty")); ③
    }
}
```

```
class Dog {
    private final String name;

    public Dog(final String name) {
        this.name = name;
    }

    public Dog(final String name1, final String name2) { ④
        this.name = name1 + name2;
    }

    @Override
    public String toString() {
        return "Dog{" +
            "name='" + name + '\'' +
            '}';
    }
}
```

① Implementacja `BiFunction` wykorzystując lambdę. Przekazujemy dwa parametry `String` oraz `Integer` i zwracamy sumę długości `Stringa` i wartości `Integer`. Typem zwracanym jest `Long` stąd rzutowanie. Wywołując metodę `apply()` przekazujemy dwa argumenty.

② Implementacja `BiFunction` wykorzystując lambdę. Do wywołania konstruktora przekazujemy skonkatenowany `String` (dlatego pierwsze dwa generyki definicji `BiFunction` to `String`) i zwracamy `Dog`.

③ Implementacja `BiFunction` wykorzystując method reference, która robi to samo co linijka 2, tyle, że w tym przypadku korzystamy z drugiego konstruktora, który przyjmuje dwa argumenty, a nie jeden tak jak w poprzednim przypadku. Gdyby nie konstruktor oznaczony jako 4, to method reference nie byłoby możliwe.

UnaryOperator

UnaryOperator, to interfejs funkcyjny, który jest specjalnym rodzajem interfejsu `Function`. `UnaryOperator` dziedziczy z `Function`. Określa tę samą metodę `apply()` co `Function`, przy czym **Unary** oznacza, że i typ wejściowy i typ wejściowy tej metody są takie same. Stosujemy go najczęściej, gdy chcemy przekształcić obiekt w obiekt tej samej klasy ale dokonać przy okazji pewnych zmian.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

    // metoda accept jest dziedziczona z Function

    // metoda identity() ma implementację jak pokazano w przykładzie
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleUnaryOperator {

    public static void main(String[] args) {
        UnaryOperator<String> unaryOperator1 = a -> a + "!enriched";
        System.out.println(unaryOperator1.apply("someString")); ①

        UnaryOperator<String> unaryOperator2 = b -> b;
        System.out.println(unaryOperator2.apply("someString")); ②

        UnaryOperator<String> unaryOperator3 = UnaryOperator.identity();
        System.out.println(unaryOperator3.apply("someString")); ③
    }

}
```

① Implementacja `UnaryOperator` wykorzystując lambdę. Typ wejściowy i wyjściowy to `String`.

② Implementacja `UnaryOperator` wykorzystując lambdę. Ponownie typ wejściowy i wyjściowy to `String`, lambda nie zmienia wejściowego `Stringa` tylko przekazuje go w takiej formie w jakiej został on do niej przekazany.

③ Implementacja `UnaryOperator` wykorzystując metodę `identity()`, której implementacja jest pokazana w przykładzie definicji interfejsu `UnaryOperator`.

BinaryOperator

BinaryOperator, to interfejs funkcyjny, który jest specjalnym rodzajem interfejsu `BiFunction`. `BinaryOperator` dziedziczy z `BiFunction`. Określa tę samą metodę `apply()` co `BiFunction`, przy czym **Binary** oznacza, że i typ dwóch parametrów wejściowych i typ wyjściowy tej metody są takie same. Stosujemy go najczęściej, gdy chcemy przekształcić dwa obiekty tej samej klasy w obiekt tej samej klasy ale np, je skleić.

Definicja tego interfejsu wygląda w ten sposób:

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {

    // metoda accept jest dziedziczona z BiFunction

    // reszta metod
}
```

Wykorzystanie tego interfejsu w praktyce:

```
public class ExampleBinaryOperator {

    public static void main(String[] args) {
        BinaryOperator<String> binaryOperator1 = (a, b) -> a + ", " + b;
        System.out.println(binaryOperator1.apply("string1", "string2")); ①

        BinaryOperator<String> binaryOperator2 = ExampleBinaryOperator::concat;
        System.out.println(binaryOperator2.apply("string1", "string2")); ②
    }

    public static String concat(String arg1, String arg2) {
        return arg1 + ", " + arg2;
    }

}
```

① Implementacja `BinaryOperator` wykorzystując lambdę. W tym przykładzie sklejamy ze sobą dwa `Stringi` przy wykorzystaniu przecinka.

② Implementacja `BinaryOperator` wykorzystując method reference. Efekt tego wywołania jest taki sam jak efekt wywołania 1.

Gdzie możemy znaleźć wbudowane interfejsy funkcyjne

Poniżej znajdziesz przykład, w którym pokazane są metody klas Javy, które korzystają z wymienionych interfejsów funkcyjnych.

```
public class ExampleSummingUp {

    public static void main(String[] args) {
        Optional.of("someValue")
            .map(value -> "value" + 192) ①
            .filter(value -> value.length() > 2) ②
            .ifPresent(value -> System.out.println(value + "another")); ③

        List<String> strings = new ArrayList<>();
        strings.removeIf(value -> value.equals("12")); ④

        Map<String, String> map = new HashMap<>();
        map.put("1", "someValue1");
```

```
        map.put("2", "someValue2");
        map.put("3", "someValue3");
        System.out.println(map);
        map.replaceAll((key, value) -> value + "!new"); ⑤
        System.out.println(map);
    }
}
```

① Metoda `Optional.map()` przyjmuje jako argument interface `Function`, który określa logikę zamiany jednego typu danych w drugi.

② Metoda `Optional.filter()` przyjmuje jako argument interface `Predicate`, który zostawia obiekt w `Optional` tylko gdy wynik `Predicate` zwróci `true`.

③ Metoda `Optional.ifPresent()` przyjmuje jako argument interface `Consumer`, który "konsumuje" przekazaną wartość o ile jest ona obecna w `Optional`.

④ Metoda `List.removeIf()` przyjmuje jako argument interface `Predicate`, który definiuje, że element ma być usunięty z `Listy` jeżeli wynik wywołania `Predicate` jest `true`.

⑤ Metoda `Map.replaceAll()` przyjmuje jako argument interface `BiFunction`, który określa logikę jak zamienić wszystkie wpisy w mapie.

Lambdy a obsługa wyjątków

Gdy korzystamy z lambd, najwygodniej jest używać wyjątków **unchecked**. Wynika to z tego, że w lambdzie nie mogą być wywoływane metody, które deklarują wyrzucenie wyjątku checked. Jedynym wyjściem jest albo obsłużenie tego wyjątku wewnątrz takiej metody aby nie deklarowała ona możliwości wyrzucenia takiego wyjątku, albo korzystanie z wyjątków **unchecked**. Demonstruje to poniższy przykład.

```
public class LambdaWithException {

    public static void main(String[] args) {
        //Supplier<String> stringSupplier1 = () -> stringCreatorChecked(); ①
        //Supplier<String> stringSupplier2 = LambdaWithException::stringCreatorChecked; ②

        Supplier<String> stringSupplier3 = () -> wrapper(); ③
        Supplier<String> stringSupplier4 = LambdaWithException::wrapper; ④

        Supplier<String> stringSupplier5 = () -> stringCreatorUnChecked(); ⑤
        Supplier<String> stringSupplier6 = LambdaWithException::stringCreatorUnChecked; ⑥
    }

    private static String wrapper() {
        try {
            return stringCreatorChecked();
        } catch (Exception e) {
            return "nope";
        }
    }

    private static String stringCreatorChecked() throws IOException {
        throw new IOException();
    }
```

```
    private static String stringCreatorUnChecked() throws RuntimeException {
        throw new RuntimeException();
    }
}
```

① Błąd kompilacji, gdyż metoda `stringCreatorChecked()` wyrzuca **checked exception**.

② Błąd kompilacji, gdyż metoda `stringCreatorChecked()` wyrzuca **checked exception**.

③ Kod kompiluje się poprawnie gdyż metoda `wrapper()` wrapuje wyrzucenie wyjątku **checked exception** i obsługuje go wewnątrz.

④ Kod kompiluje się poprawnie gdyż metoda `wrapper()` wrapuje wyrzucenie wyjątku **checked exception** i obsługuje go wewnątrz.

⑤ Kod kompiluje się poprawnie gdyż metoda `stringCreatorUnChecked()` wyrzuca wyjątek **unchecked**, a to już jest dozwolone.

⑥ Kod kompiluje się poprawnie gdyż metoda `stringCreatorUnChecked()` wyrzuca wyjątek **unchecked**, a to już jest dozwolone.