

The background is a solid blue gradient. Overlaid on this are numerous thin, white, curved lines that flow from the left side towards the right, creating a sense of motion and depth. These lines are more densely packed in some areas, forming a wave-like pattern that peaks towards the right side of the image.

JAVA SE 8 PROGRAMMER II

ZAKRES SZKOLENIA – DZIEŃ 2

- Interfejsy funkcyjne wyrażen lambda

interfejsy z pakietu `java.util.function`: `Predicate`, `Consumer`, `Function`, `Supplier`;

interfejsy funkcyjne w wersjach do typów prostych;

użycie dwuargumentowych interfejsów funkcyjnych; interfejs `UnaryOperator`

- Java Stream API

metody: `peek` oraz `map`; wyszukiwanie danych: `findFirst`, `findAny`, `anyMatch`, `allMatch`,

`noneMatch`; klasa `Optional`; metody wyliczające oraz wyszukujące z interfejsu `Stream`;

sortowanie; zbieranie wyników z użyciem metody `collect`; podział i grupowanie danych

przy pomocy klasy `Collectors`; metoda `flatMap`

- Wyjątki i asercje

konstrukcja try-catch oraz throw; użycia catch, multi-catch oraz finally; konstrukcja

try-with-resources; tworzenie własnych zasobów `AutoCloseable`; użycie asercji



INTERFEJSY FUNKCYJNE

Java jest językiem obiektowym, który opiera się na tworzeniu obiektów i komunikacji między nimi. W tym dziale poznamy **paradygmat programowania funkcyjnego**. Programowanie funkcyjne **opiera się jedynie na funkcjach**. Główny program jest funkcją, której podajemy argumenty, a w odpowiedzi otrzymujemy wyznaczony rezultat. Główna funkcja programu składa się tylko i wyłącznie z innych funkcji, które z kolei agregują inne.

INTERFEJSY FUNKCYJNE

Wiele interfejsów w Javie składa się tylko i wyłącznie z jednej abstrakcyjnej metody, np. Runnable, Callable, Comparator. Interfejsy takie nazywane są **Single Abstract Method**. W Javie 8 wprowadzono nazwę **FunctionalInterface** dla tego typu komponentów. Zdefiniowany na slajdzie interfejs Action jest interfejsem funkcyjnym.

```
1 public interface Action {  
2     void execute(int x, int y);  
3 }
```

INTERFEJSY FUNKCYJNE

Poniższy interfejs nie może zostać zakwalifikowany jako interfejs typu SAM, ponieważ posiada **dwie metody abstrakcyjne**.

```
public interface Presenter {  
    void present(String text);  
    void present(String text, int size);  
}
```

@FUNCTIONALINTERFACE

W celu łatwiejszej identyfikacji interfejsów funkcyjnych, w Javie wprowadzona została adnotacja **@FunctionalInterface**, która informuje programistę, że wskazany interfejs **ma być z założenia interfejsem funkcyjnym**. Adnotacja ta powinna zostać umieszczona nad definicją interfejsu. Próba umieszczenia takiej informacji nad interfejsem, który ma np. zero lub dwie metody abstrakcyjne, **skończy się błędem kompilacji**.

UWAGA: Interfejs funkcyjny **NIE musi być oznaczony adnotacją @FunctionalInterface**.

```
@FunctionalInterface
public interface Executor {
    void executor(int x);
}
```

DEFAULT

W Javie 8 został wprowadzony mechanizm **metod domyślnych**, określony za pomocą słowa kluczowego **default**. Metoda domyślna jest to metoda interfejsu, która zawiera domyślną implementację (tzn. metoda ta ma ciało). Dzięki tej nowej składni, w ramach interfejsu funkcyjnego, możemy deklarować **więcej niż jedną metodę, z zachowaniem zasady pojedynczej metody abstrakcyjnej**.

Metody typu default mogą być nadpisywane w klasach implementujących, bądź też realizować domyślną funkcjonalność.

```
@FunctionalInterface
public interface Executor {
    void executor(int x);

    default void executor(int x, int y) {
        // I have a body I am a default method
    }
}
```

WYRAŻENIA LAMBDA

Dużym problemem klas anonimowych jest to, że nawet **jeśli implementujemy prosty interfejs z jedną metodą, to forma zapisu potrafi być nieczytelna**. Przykładem takiej sytuacji jest przekazanie funkcji do innej metody, np. do obsługi kliknięcia przycisku.

```
Thread thread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Implementacja interfejsu Runnable jako implementacja klasy anonimowej!");  
    }  
});  
thread.start();
```


WYRAŻENIA LAMBDA

W Javie 8 zostały wprowadzone **wyrażenia lambda**, które pozwalają traktować klasę anonimową jak **zwykłą funkcję**, znacząco skracając przy tym składnię samego zapisu.

```
Thread thread = new Thread(() -> {  
    System.out.println("Runnable example using lambda!");  
});  
thread.start();
```

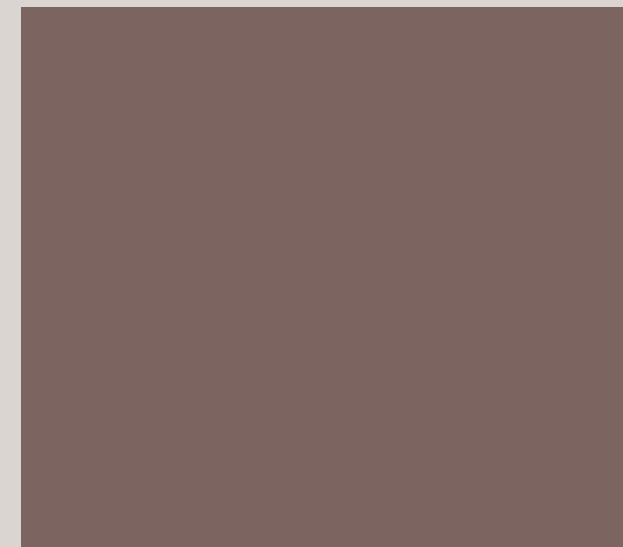
Jak widać wyrażenie lambda **zwalnia nas z pisania słowa kluczowego new i wykorzystywania adnotacji Override**.



SKŁADNIA WYRAŻENIA LAMBDA

Wyrażenie lambda składa się z trzech części:

- listy argumentów, która:
 - musi znaleźć się **w nawiasie, jeżeli ilość argumentów jest różna od 1**
 - **może, ale nie musi**, podawać **typy argumentów**
- operatora **->**, który jest używany po liście argumentów
- ciała implementowanej metody, które znajduje się **za operatorem ->**
 - jeżeli ciało składa się z **jednego wyrażenia** to:
 - ciało to **nie musi** być wewnątrz klamer, tzn. { i }
 - w przypadku, gdy to wyrażenie **zwraca obiekt**, możemy **pomiąć słowo kluczowe return**



SKŁADNIA WYRAŻENIA LAMBDA

- jeżeli ciało składa się z **wielu wyrażeń**, to:
 - ciało to musi znaleźć się **wewnątrz klamer**, tzn. { i }.

UWAGA: jeżeli zadaniem lambdy jest **wyrzucenie wyjątku**, to musi to zostać zrobione **wewnątrz klamer**, pomimo tego, że jest to pojedyncze wyrażenie.

UWAGA: nazwy argumentów mogą różnić się od tych zdefiniowanych w interfejsie.

UWAGA: definiując lambdę zupełnie **nie interesuje nas nazwa metody abstrakcyjnej w interfejsie**.



LAMBDA – PRZYKŁAD 1

```
public interface Action {  
    String execute(int x, int y);  
}
```

```
Action action = (int x, int y) -> {  
    return x + "-" + y;  
};
```

```
Action action = (x, y) -> x + "-" + y;
```



LAMBDA – PRZYKŁAD 2

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

```
Runnable runnableExample = () -> {
    System.out.println("Hello from runnable");
    System.out.println("{ and } cannot be omitted");
};
```



LAMBDA – PRZYKŁAD 3

```
@FunctionalInterface
public interface FruitEater<T> {
    void consume(T t);
}
```

```
FruitEater<String> fruitEater = fruit ->
    System.out.println(String.format("eating %s... omnomnom", fruit));
```


REFERENCJE METOD

Niektóre proste wyrażenia lambda, **możemy zapisać za pomocą referencji metody**. Zamiast pisać wywołanie metody, możemy podać jedynie jej nazwę. W takim przypadku **nazwę klasy i metody musimy oddzielić od siebie znakami '::'**.

Z referencji do metody możemy korzystać, gdy **lambda posiada jeden argument oraz jest spełniony jeden z warunków**:

- argument wejściowy lambdy jest argumentem metody z pewnej klasy
- na argumencie wejściowym wywoływana jest bezargumentowa metoda.



REFERENCJE METOD

```
// wykorzystanie lambdy
Consumer<String> consumerExample = someString -> System.out.println(someString);
// identyczny zapis, jak w linijce powyżej, użycie referencji
Consumer<String> consumerExampleReference = System.out::println;

// wykorzystanie lambdy w metodzie map
List.of("someString").stream().map(str -> str.toUpperCase());
// wykorzystanie referencji do metody, zapis równoważny w linijce kodu wyżej
List.of("someString").stream().map(String::toUpperCase);
```

INTERFEJSY FUNKCYJNE W JAVIE 8

W Javie 8 wprowadzonych zostało **wiele przydatnych interfejsów funkcyjnych**, które można wykorzystywać m.in. w **Stream API**. Wszystkie interfejsy, które zostaną omówione, znajdują się w pakiecie **java.util.function**. Najczęściej wykorzystywane to:

- **Supplier<T>**
- **Function<T, R>**
- **Consumer<T>**
- **UnaryOperator<T, T>**
- **Predicate<T>**



INTERFEJSY FUNKCYJNE W JAVIE 8

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
Supplier<T>	0	T	get
Consumer<T>	1 (T)	void	accept
BiConsumer<T, U>	2 (T, U)	void	accept
Predicate<T>	1 (T)	boolean	test
BiPredicate<T, U>	2 (T, U)	boolean	test
Function<T, R>	1 (T)	R	apply
BiFunction<T, U, R>	2 (T, U)	R	apply
UnaryOperator<T>	1 (T)	T	apply
BinaryOperator<T>	2 (T, T)	T	apply

FUNCTION BIFUNCTION

Function jest generycznym interfejsem funkcyjnym, który **na wejście przyjmuje obiekt dowolnego typu (T)** i **zwraca obiekt dowolnego typu (R)**. Metoda **apply** jest odpowiedzialna za wywołanie zaimplementowanej przez nas akcji.

```
public class FunctionExample {
    public static void main(String[] args) {
        Function<Employee, String> employeeToString = (employee) -> employee.getName(); // (1)

        List<Employee> employees = Arrays.asList(new Employee("test"), new Employee("test2"));
        showEmployee(employees, employeeToString);
    }

    static void showEmployee(List<Employee> employees, Function<Employee, String> showFunction) {
        for (Employee employee : employees) {
            System.out.println(showFunction.apply(employee));
        }
    }
}
```

```
@FunctionalInterface public class Function<T, R> {
    R apply(T t);
}

@FunctionalInterface public class BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

```
Function<String, Integer> f1 = String::length;
Function<String, Integer> f2 = x -> x.length();

System.out.println(f1.apply("cluck")); // 5
System.out.println(f2.apply("cluck")); // 5
```

```
BiFunction<String, String, String> b1 = String::concat;
BiFunction<String, String, String> b2 = (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick")); // baby chick
System.out.println(b2.apply("baby ", "chick")); // baby chick
```

```
class Employee {
    private String name;

    public Employee(String name) {
        this.name = name;
    }

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

UNARYOPERATOR BINARYOPERATOR

UnaryOperator oraz **BinaryOperator** są specjalnymi przypadkami interfejsów **Function** oraz **BiFunction**. Podstawowym wymaganiem jest fakt, iż **wszystkie typy argumentów muszą być jednakowe**.

```
@FunctionalInterface public class UnaryOperator<T>
    extends Function<T, T> { }
@FunctionalInterface public class BinaryOperator<T>
    extends BiFunction<T, T, T> { }
```

```
T apply(T t);
T apply(T t1, T t2);
```

```
UnaryOperator<String> u1 = String::toUpperCase;
UnaryOperator<String> u2 = x -> x.toUpperCase();

System.out.println(u1.apply("chirp"));
System.out.println(u2.apply("chirp"));
```

```
BinaryOperator<String> b1 = String::concat;
BinaryOperator<String> b2 = (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick")); // baby chick
System.out.println(b2.apply("baby ", "chick")); // baby chick
```


SUPPLIER

Supplier jest generycznym interfejsem funkcyjnym, którego **odpowiedzialnością jest dostarczenie wartości obiektu typu T**. Metoda **get** zwraca wartość zaimplementowaną w interfejsie.

```
@FunctionalInterface public class Supplier<T> {  
    public T get();  
}
```

```
public class SupplierExample {  
    public static void main(String[] args) {  
        getValue(() -> "supplier test!");  
    }  
  
    static void getValue(Supplier<String> supplier){  
        System.out.println(supplier.get());  
    }  
}
```

```
Supplier<LocalDate> s1 = LocalDate::now;  
Supplier<LocalDate> s2 = () -> LocalDate.now();
```

```
LocalDate d1 = s1.get();  
LocalDate d2 = s2.get();
```

```
System.out.println(d1);  
System.out.println(d2);
```

```
Supplier<StringBuilder> s1 = StringBuilder::new;  
Supplier<StringBuilder> s2 = () -> new StringBuilder();  
  
System.out.println(s1.get());  
System.out.println(s2.get());
```

```
Supplier<ArrayList<String>> s1 = ArrayList<String>::new;  
ArrayList<String> a1 = s1.get();  
System.out.println(a1);
```

CONSUMER BICONSUMER

Consumer jest kolejnym generycznym interfejsem funkcyjnym. Reprezentuje **operację, która akceptuje pojedynczy argument wejściowy i nie zwraca żadnego wyniku**. Generyczny typ T jest argumentem metody. Interfejs wykorzystywany jest, gdy istnieje potrzeba „konsumpcji” obiektu. **Metoda accept jest odpowiedzialna za wywołanie implementowanej metody.**

```
@FunctionalInterface public class Consumer<T> {  
    void accept(T t);  
}  
  
@FunctionalInterface public class BiConsumer<T, U> {  
    void accept(T t, U u);  
}
```

```
public class ConsumerExample {  
    public static void main(String[] args) {  
        Consumer<String> stringTrim = (s) -> {  
            s = s.trim();  
            System.out.println(s);  
        }; // implementacja Consumera za pomocą wielolinijkowej lambdy  
  
        trimValue(stringTrim, "  text  ");  
    }  
  
    static void trimValue(Consumer<String> trimAction, String s) {  
        trimAction.accept(s);  
    }  
}
```

```
Consumer<String> c1 = System.out::println;  
Consumer<String> c2 = x -> System.out.println(x);  
  
c1.accept("Annie");  
c2.accept("Annie");
```

```
Map<String, Integer> map = new HashMap<>();  
BiConsumer<String, Integer> b1 = map::put;  
BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);  
  
b1.accept("chicken", 7);  
b2.accept("chick", 1);  
  
System.out.println(map);
```

PREDICATE BIPREDICATE

Interfejsy funkcyjny **Predicate** reprezentuje operację, która akceptuje pojedynczy argument i zwraca wartość logiczną na podstawie przekazanego parametru. Jest więc specjalizacją interfejsu Function. Generyczny typ T jest typem argumentu wejściowego metody. Metoda **test** jest odpowiedzialna za zwrócenie wartości logicznej testu.

```
public class PredicateExample {  
    public static void main(String[] args) {  
        Predicate<Integer> predicate = (value) -> {  
            return value >= 0;  
        };  
        checkTest(predicate);  
    }  
  
    static void checkTest(Predicate<Integer> predicate) {  
        System.out.println(predicate.test(-1));  
    }  
}
```

```
@FunctionalInterface public class Predicate<T> {  
    boolean test(T t);  
}  
  
@FunctionalInterface public class BiPredicate<T, U> {  
    boolean test(T t, U u);  
}
```

```
Predicate<String> p1 = String::isEmpty;  
Predicate<String> p2 = x -> x.isEmpty();
```

```
BiPredicate<String, String> b1 = String::startsWith;  
BiPredicate<String, String> b2 = (string, prefix) -> string.startsWith(prefix);  
  
System.out.println(b1.test("chicken", "chick"));  
System.out.println(b2.test("chicken", "chick"));
```

INTERFEJSY FUNKCYJNE - PODSUMOWANIE

```
6:  ____<List> ex1 = x -> "".equals(x.get(0));  
7:  ____<Long> ex2 = (Long l) -> System.out.println(l);  
8:  ____<String, String> ex3 = (s1, s2) -> false;
```

```
6:  Function<List<String>> ex1 = x -> x.get(0); // DOES NOT COMPILE  
7:  UnaryOperator<Long> ex2 = (Long l) -> 3.14; // DOES NOT COMPILE  
8:  Predicate ex4 = String::isEmpty; // DOES NOT COMPILE
```

INTERFEJSY FUNKCYJNE - PRYMITYWY

Istnieją specjalne interfejsy dedykowane obsłudze typów prostych (prymitywnych), tj. **int**, **long**, **double**, **boolean**.

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
DoubleSupplier	0	double	getAsDouble
IntSupplier		int	getAsInt
LongSupplier		long	getAsLong
DoubleConsumer	1 (double)	void	accept
IntConsumer	1 (int)		
LongConsumer	1 (long)		
DoublePredicate	1 (double)	boolean	test
IntPredicate	1 (int)		
LongPredicate	1 (long)		
DoubleFunction<R>	1 (double)	R	apply
IntFunction<R>	1 (int)		
LongFunction<R>	1 (long)		
DoubleUnaryOperator	1 (double)	double	applyAsDouble
IntUnaryOperator	1 (int)	int	applyAsInt
LongUnaryOperator	1 (long)	long	applyAsLong
DoubleBinaryOperator	2 (double, double)	double	applyAsDouble
IntBinaryOperator	2 (int, int)	int	applyAsInt
LongBinaryOperator	2 (long, long)	long	applyAsLong

INTERFEJSY FUNKCYJNE

- PRYMITYWY

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
<code>ToDoubleFunction<T></code>	1 (T)	double	<code>applyAsDouble</code>
<code>ToIntFunction<T></code>		int	<code>applyAsInt</code>
<code>ToLongFunction<T></code>		long	<code>applyAsLong</code>
<code>ToDoubleBiFunction<T, U></code>	2 (T, U)	double	<code>applyAsDouble</code>
<code>ToIntBiFunction<T, U></code>		int	<code>applyAsInt</code>
<code>ToLongBiFunction<T, U></code>		long	<code>applyAsLong</code>
<code>DoubleToIntFunction</code>	1 (double)	int	<code>applyAsInt</code>
<code>DoubleToLongFunction</code>	1 (double)	long	<code>applyAsLong</code>
<code>IntToDoubleFunction</code>	1 (int)	double	<code>applyAsDouble</code>
<code>IntToLongFunction</code>	1 (int)	long	<code>applyAsLong</code>
<code>LongToDoubleFunction</code>	1 (long)	double	<code>applyAsDouble</code>
<code>LongToIntFunction</code>	1 (long)	int	<code>applyAsInt</code>
<code>ObjDoubleConsumer<T></code>	2 (T, double)	void	<code>accept</code>
<code>ObjIntConsumer<T></code>	2 (T, int)		
<code>ObjLongConsumer<T></code>	2 (T, long)		



INTERFEJSY FUNKCYJNE

- BOOLEANSUPPLIER

```
boolean getAsBoolean()
```

```
12: BooleanSupplier b1 = () -> true;  
13: BooleanSupplier b2 = () -> Math.random() > .5;  
14: System.out.println(b1.getAsBoolean());  
15: System.out.println(b2.getAsBoolean());
```



OPTIONAL

Zamiast tego możemy wykorzystać mechanizm wprowadzony w Javie 8. **Optional**, znajdujący się w pakiecie **java.util** jest obiektem, który **opakowuje obiekt docelowy**, tzn., jest pewnego rodzaju pudełkiem, które taki obiekt może zawierać lub nie.

Klasa Optional daje nam dostęp do wielu metod:

- **of**
- **ofNullable**
- **isPresent**
- **ifPresent**
- **orElse**
- **orElseGet**



OPTIONAL - KREACJA

Za tworzenie obiektów odpowiedzialna jest metoda statyczna **of** lub **ofNullable**. Różnica między pierwszą, a drugą metodą polega m.in. na tym, że pierwsza z metod **nie pozwala na przyjmowanie wartości typu null**.

```
public class CreatingOptionals {  
    public static void main(String[] args) {  
        // stworzenie pudełka z obiektem typu String  
        final Optional<String> stringOptional = Optional.of("This is Java course!!!");  
  
        String value = null;  
        if ((Integer.parseInt(args[0]) % 2 == 0)) {  
            value = "I am even";  
        }  
        // wykorzystanie metody ofNullable, ponieważ value może być nullem.  
        // W takim przypadku Optional będzie pustym pudełkiem  
        final Optional<String> optionalThatCanBeEmpty = Optional.ofNullable(value);  
    }  
}
```

OPTIONAL – SPRAWDZANIE WARTOŚCI

Klasa **Optional** udostępnia metody, które pozwalają na **warunkowe wykonanie operacji**, jeśli faktycznie w jego wnętrzu znajduje się referencja do obiektu.

- metoda **isPresent** zwraca wartość true/false w zależności od tego, czy Optional przechowuje jakiś obiekt
- metoda **ifPresent** przyjmuje argument w postaci interfejsu funkcyjnego Consumer. Consumer ten **wywoła się tylko i wyłącznie w przypadku, gdy Optional przechowuje referencję do obiektu.**

```
public class OptionalsPresenceExample {
    public static void main(String[] args) {
        final Optional<String> optional = getStringForEvenNumber(3);
        if (optional.isPresent()) {
            System.out.println("I am optional with a value, I am non empty box");
        } else if (optional.isEmpty()) { // warunek zawsze prawdziwy w tym momencie
            System.out.println("I am an empty optional");
        }

        // wypisanie wartości w pudełku na ekranie, tylko jeżeli jest dostępna
        optional.ifPresent(System.out::println);
    }

    private static Optional<String> getStringForEvenNumber(final int number) {
        if (number % 2 == 0) {
            return Optional.of("even");
        }
        return Optional.empty();
    }
}
```

POBIERANIE WARTOŚCI

Klasa Optional udostępnia kilka metod umożliwiających pobieranie wartości:

- metoda **get** zwraca wartość przechowywaną w obiekcie, bądź też **w przypadku nulla wyrzuca wyjątek: NoSuchElementException**
- metoda **orElse** zwraca wartość przechowywaną w Optionalu **lub wartość wskazaną jako argument metody w przypadku pustego Optionala**
- metoda **orElseGet** zwraca wartość przechowywaną w Optionalu **lub wartość wskazaną przez Supplier, który jest argumentem wejściowym.**

```
public class OptionalOrElseExample {
    public static void main(String[] args) {
        String object = null;
        String name = Optional.ofNullable(object).orElse("john");
        System.out.println(name); // na ekranie zostanie wypisana wartość john
    }
}

public class OptionalOrElseGetExample {
    public static void main(String[] args) {
        String object = null;
        String name = Optional.ofNullable(object).orElseGet(() -> "john");
        System.out.println(name); // na ekranie ponownie zostanie wyświetlony String john
    }
}
```

POBIERANIE WARTOŚCI

UWAGA: Chcąc **wykorzystać metodę get**, powinniśmy najpierw **sprawdzić dostępność obiektu za pomocą metody isPresent**.

UWAGA: Metoda **orElseGet** w przeciwieństwie do **orElse** wyliczy wartość zastępczą **tylko, gdy Optional jest pusty**. Oznacza to, że wydajność metody **orElseGet** jest lepsza, niż metody **orElse**.



POBIERANIE WARTOŚCI

Method	When Optional Is Empty	When Optional Contains a Value
<code>get()</code>	Throws an exception	Returns value
<code>ifPresent(Consumer c)</code>	Does nothing	Calls Consumer c with value
<code>isPresent()</code>	Returns false	Returns true
<code>orElse(T other)</code>	Returns other parameter	Returns value
<code>orElseGet(Supplier s)</code>	Returns result of calling Supplier	Returns value
<code>orElseThrow(Supplier s)</code>	Throws exception created by calling Supplier	Returns value

JAVA STREAM API

Klasy dające dostęp do interfejsu **Stream** z pakietu **java.util.stream** umożliwiają funkcyjny sposób przetwarzania danych. Strumienie (tzw. **streamy**) **reprezentują** sekwencyjny zestaw elementów i pozwalają na **wykonywanie** różnych **operacji na tych elementach**.

STRUMIENIE A KOLEKCJE

Strumienie są pewnego **rodzaju reprezentacją kolekcji**, jednak są między nimi pewne różnice:

- **Strumień nie jest strukturą danych**, która przechowuje elementy. Przekazuje on tylko referencje elementów ze źródła, którym może być np.: struktura danych, tablica, kolekcja, metoda generatora
- Operacje na strumieniu **zwracają wynik**, ale **nie modyfikują źródła** strumienia, np. **filtrowanie** strumienia uzyskanego z kolekcji powoduje utworzenie nowego strumienia **bez przefiltrowanych elementów**, natomiast **żaden element nie jest usuwany z oryginalnej kolekcji**.



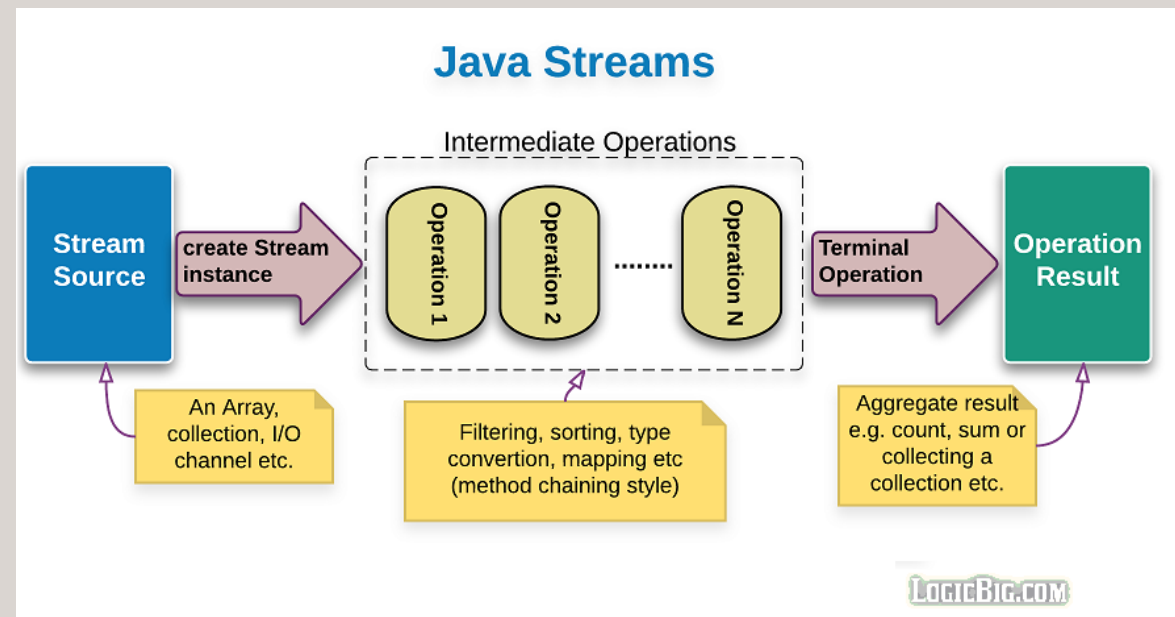
STRUMIENIE A KOLEKCJE

- **Kolekcje mają skończony rozmiar**, strumienie **nie muszą posiadać skończonego rozmiaru**. Operacje takie jak **limit(n)** lub **findFirst()** pozwalają na zakończenie obliczeń w strumieniach w skończonym czasie.
- Elementy w strumieniach **odwiedzane są tylko raz podczas jego trwania**. Podobnie jak Iterator, **niezbędne jest wygenerowanie nowego strumienia**, aby ponownie odwiedzić te same elementy źródła, np. kolekcji.



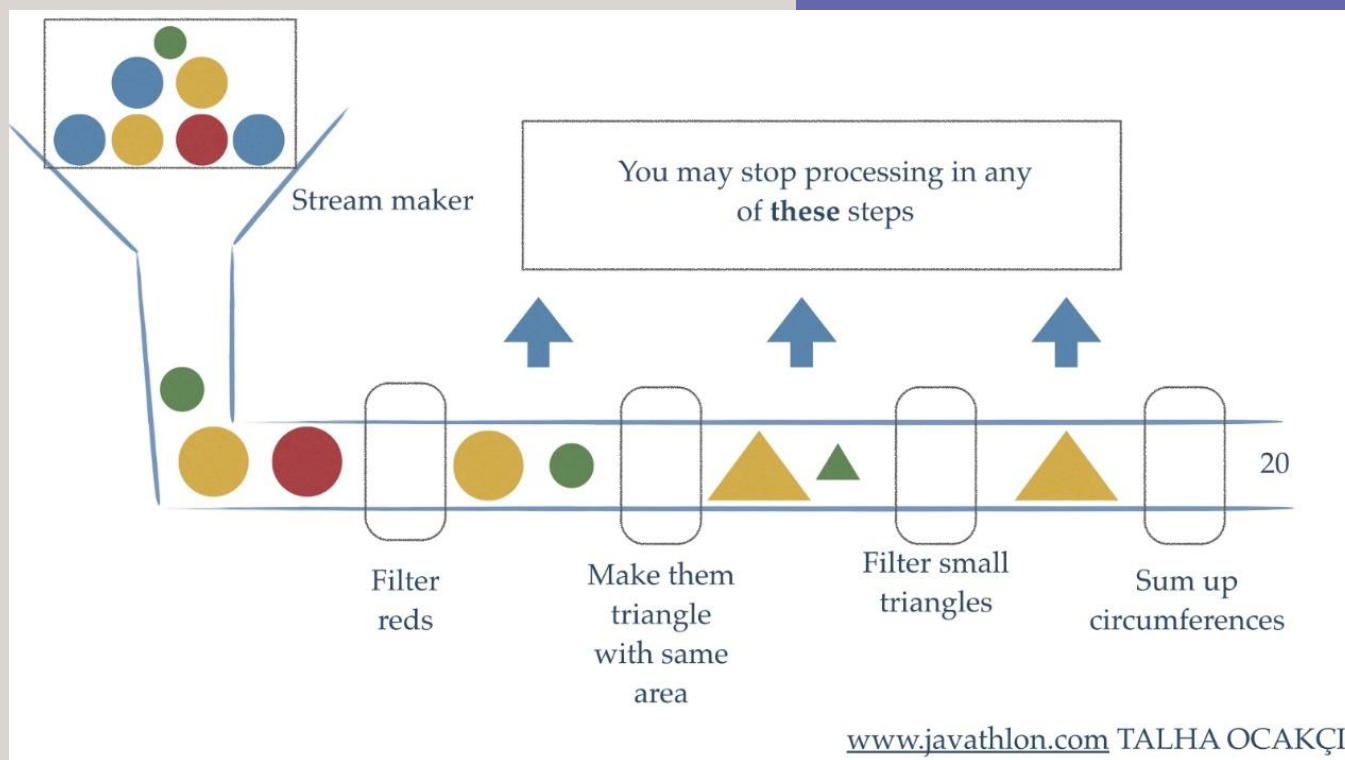
STRUMIENIE A KOLEKCJE

- **Strumienie są przetwarzane w sposób leniwy (lazy)**, to znaczy żadne z metod pośrednich nie są wywoływane, **aż do momentu wywołania jednej z metod kończących**.



STRUMIENIE A KOLEKCJE

- PRZYKŁAD



STRUMIENIE - KREACJA

W Javie istnieje wiele sposobów otrzymywania strumieni na podstawie wskazanych kolekcji, tablic, obiektów:

- metoda **stream()** zwraca strumień dla klas dostępnych w Collection API
- metoda **Arrays.stream(Object[])** umożliwia tworzenie strumieni na podstawie tablic
- metoda statyczna **Stream.of(T ...values)** umożliwia tworzenie strumieni na podstawie tablic, obiektów
- metoda statyczna **Stream.generate()** tworzy Stream elementów na podstawie wejściowego Suppliera
- strumienie plików mogą być zwracane na podstawie klasy Files.



STRUMIENIE - KREACJA

```
Stream<Integer> streamOfInts = Arrays.asList(1, 2, 3).stream();
Stream<String> streamOfStrings = Set.of("one", "two", "three").stream();
Stream<Map.Entry<String, Integer>> stream = Map.of("someKeyA", 1, "someKeyB", 2).entrySet().stream();
IntStream arraysStream = Arrays.stream(new int[]{1, 2, 3});
Stream<Double> ofStream = Stream.of(1.1, 2.2, 3.3);
Stream<Integer> generateStream = Stream.generate(() -> new Random().nextInt());
Stream<String> fileLinesStream = Files.lines(Path.of("/tmp/1.txt"));
```

```
1: Stream<String> empty = Stream.empty();    // count = 0
2: Stream<Integer> singleElement = Stream.of(1);    // count = 1
3: Stream<Integer> fromArray = Stream.of(1, 2, 3);    // count = 2
```

```
4: List<String> list = Arrays.asList("a", "b", "c");
5: Stream<String> fromList = list.stream();
6: Stream<String> fromListParallel = list.parallelStream();
```

```
7: Stream<Double> randoms = Stream.generate(Math::random);
8: Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```


OPERACJE NA STRUMIENIACH

Operacje na strumieniach dzielą się na dwa rodzaje - **pośrednie** oraz **kończące**. Operacje pośrednie zawsze zwracają nowy strumień, są one też przetwarzane w sposób lazy (leniwy). Zaliczamy do nich m.in.:

- **filter**
- **map**
- **flatMap**
- **peek**
- **distinct**
- **sorted**
- **limit**



OPERACJE NA STRUMIENIACH

Operacje kończące są operacjami, które zwracają końcowy wynik. Ich wywołanie powoduje **wykonanie się wszystkich poprzedzających funkcji pośrednich**. Do funkcji kończących zaliczamy:

- **toArray**
- **collect**
- **count**
- **reduce**
- **forEach, forEachOrdered**
- **min, max**
- **anyMatch, allMatch, noneMatch**
- **findAny, findFirst**



OPERACJE NA STRUMIENIACH



Scenario	For Intermediate Operations?	For Terminal Operations?
Required part of a useful pipeline?	No	Yes
Can exist multiple times in a pipeline?	Yes	No
Return type is a stream type?	Yes	No
Executed upon method call?	No	Yes
Stream valid after call?	Yes	No

OPERACJE POŚREDNIE

- MAP

Metoda **map** oczekuje na wejście obiektu **Function<T, R>**. Jej zadaniem jest **przekonwertowanie elementu** strumienia na nowy element, który dodatkowo może mieć inny typ.

```
// stworzenie streamu i przetworzenie elementów wejściowych typu Integer  
// na wartość trzy razy większą o typie Double  
List.of(1, 2, 3).stream()  
    .map(streamElem -> streamElem * 3.0);
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");  
s.map(String::length)
```



OPERACJE POŚREDNIE -FLATMAP

Metoda **flatMap** umożliwia spłaszczenie zagnieżdżonej struktury danych. Oznacza to, że jeżeli każdy przetwarzany element posiada element z którego jesteśmy w stanie stworzyć nowy Stream, to **wynikiem operacji flatMap będzie nowy, pojedynczy Stream**, który powstał przez ich połączenie w jeden. Operacja flatMap przyjmuje w parametrze interfejs funkcyjny **Function<T, ? extends Stream<? extends R>>**. Przykład pokazuje, w jaki sposób z obiektów typu Statistics i pól values stworzyć pojedynczy strumień.

```
public class FlatMapDemo {  
  
    public static void main(String[] args) {  
        final Statistics statisticsA = new Statistics(2.0, List.of(1, 2, 3));  
        final Statistics statisticsB = new Statistics(2.5, List.of(2, 3, 2, 3));  
        // Otrzymujemy stream wartości 1, 2, 3, 2, 3, 2, 3  
        Stream.of(statisticsA, statisticsB)  
            .flatMap(statistics -> statistics.getValues().stream());  
    }  
}  
  
class Statistics {  
    private double average;  
    private List<Integer> values;  
  
    public Statistics(final double average, final List<Integer> values) {  
        this.average = average;  
        this.values = values;  
    }  
  
    public double getAverage() {  
        return average;  
    }  
  
    public List<Integer> getValues() {  
        return values;  
    }  
}
```

OPERACJE POŚREDNIE -FILTER

Operacja **filter** umożliwia **wyrzucenie ze strumienia tych elementów**, które nie spełniają pewnego predykatu, który jest argumentem wejściowym metody. Przykład poniżej pokazuje w jaki sposób usunąć liczby nieparzyste ze strumienia liczb.

```
final int[] idx = { 0 };  
Stream.generate(() -> idx[0]++)  
    .limit(10)  
    .filter(elem -> elem % 2 == 0); //w strumieniu zostają: 0, 2, 4, 6, 8
```



OPERACJE POŚREDNIE -SORTED

Metoda **sorted** posortuje elementy w strumieniu. Dostępna jest wersja bezargumentowa, która **sortuje elementu w sposób naturalny**. Jeżeli chcemy posortować elementy wg innej reguły, powinniśmy wykorzystać przeciążenie, które **wykorzystuje interfejs funkcyjny `Comparator<T>`**.

```
Arrays.asList(6, 3, 6, 21, 20, 1).stream()  
    .sorted(Comparator.reverseOrder());  
// w streamie znajdziemy: 21, 20, 6, 6, 3, 1 - w tej kolejności
```

```
Stream<T> sorted()  
Stream<T> sorted(Comparator<? super T> comparator)
```

```
Stream<String> s = Stream.of("brown-", "bear-");  
s.sorted().forEach(System.out::print); // bear-brown-
```

```
Stream<String> s = Stream.of("brown bear-", "grizzly-");  
s.sorted(Comparator.reverseOrder())  
    .forEach(System.out::print); // grizzly-brown bear-
```

```
s.sorted(Comparator::reverseOrder); // DOES NOT COMPILE
```

OPERACJE POŚREDNIE -DISTINCT

Operacja **distinct** umożliwia stworzenie strumienia, w którym **wszystkie elementy są unikalne**, tzn. pozbywamy się powtórzeń, np.:

```
Arrays.asList(3, 6, 6, 20, 21, 21).stream()  
    .distinct(); // w streamie zostaną elementy: 3, 6, 20, 21
```



OPERACJE POŚREDNIE

-LIMIT, SKIP

Metody **limit** oraz **skip** pozwalają na ograniczenie strumienia.

```
Stream<T> limit(int maxSize)  
Stream<T> skip(int n)
```

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);  
s.skip(5).limit(2).forEach(System.out::print);    // 67
```



OPERACJE POŚREDNIE -PEEK

Metoda **peek** jest metodą najczęściej wykorzystywaną w celach analizy błędów, diagnostyki problemów, debugowania. Metoda ta pozwala na wykonywanie pewnych operacji bez modyfikacji strumienia. Najczęstszym przypadkiem zastosowanie tej metody jest wypisywanie elementów strumienia.

```
Stream<T> peek(Consumer<? super T> action)
```

```
Stream<String> stream = Stream.of("black bear", "brown bear", "grizzly");  
long count = stream.filter(s -> s.startsWith("g"))  
    .peek(System.out::println).count();    // grizzly  
System.out.println(count);    // 1
```



OPERACJE TERMINALNE -FOREACH

Operacja **forEach** reprezentuje **funkcyjną wersję pętli for**. Funkcja ta wywołuje dowolną operację zaimplementowaną za pomocą interfejsu funkcyjnego **Consumer<T>** na **każdym elemencie strumienia**.

```
List.of(1, 2, 3, 4, 5).stream()  
    .forEach(System.out::println);
```



```
void forEach(Consumer<? super T> action)
```

```
Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo");  
s.forEach(System.out::print);    // MonkeyGorillaBonobo
```

```
Stream s = Stream.of(1);  
for (Integer i: s) {} // DOES NOT COMPILE
```

OPERACJE TERMINALNE

- COLLECT

Metoda **collect** pozwala **zebrać elementy strumienia do pewnego obiektu docelowego**. W celu zebrania elementów musimy wykorzystać interfejs **Collector**, który **nie jest interfejsem funkcyjnym**. Z pomocą przychodzi klasa **Collectors**, która zawiera statyczne metody odpowiedzialne za kumulację elementów strumienia do wskazanej struktury, np. **List**, bądź **Set**.

```
final List<Integer> listCreatedFromCollectMethod
= Stream.generate(() -> new Random().nextInt())
    .limit(10)
    .distinct()
    .filter(elem -> Math.abs(elem) < 1000)
    .collect(Collectors.toUnmodifiableList());

final String sentence = Stream.of("This", "will", "be", "single",
    "sentence", "but", "without", "some", "words")
    .filter(word -> word.length() > 2)
    .collect(Collectors.joining(" "));
System.out.println(sentence);
// outputem będzie "This will single sentence but without some words"

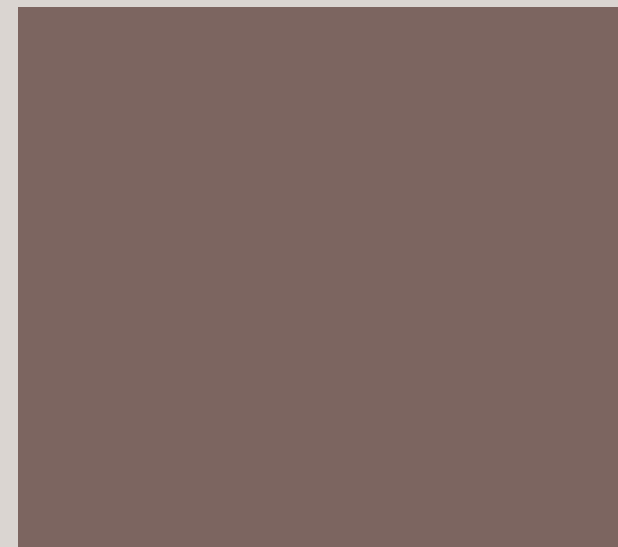
final Map<String, String> wordToUppercasedVersion
= Stream.of("Hello", "from", "Stream", "api")
    .collect(Collectors.toMap(Function.identity(), String::toUpperCase));
```

OPERACJE TERMINALNE

- COLLECT

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator,  
BiConsumer<R, R> combiner)  
<R,A> R collect(Collector<? super T, A,R> collector)
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");  
StringBuilder word = stream.collect(StringBuilder::new,  
    StringBuilder::append, StringBuilder::append)
```



OPERACJE TERMINALNE - COLLECT - GRUPOWANIE

Collector uzyskany za pomocą **groupingBy** pozwala nam ze strumienia stworzyć obiekt **Map**. Metoda **groupingBy** oczekuje Function, który staje się pewnego rodzaju własnością strumienia. Wynikiem jest mapa, której kluczami są wartości wcześniej wspomnianej własności, natomiast wartością - lista elementów spełniających tę własność. Najlepiej to zobrazować na przykładzie, np. kod poniżej tworzy mapę, która grupuje w strumieniu słowa po ich długości:

```
Stream.of("This", "is", "Java", "the", "best", "academy", "in", "the", "universe")  
    .collect(Collectors.groupingBy(String::length))  
    .forEach((key, value) -> System.out.println(key + " " + value));
```

```
/* wynik programu to:  
2 [is, in]  
3 [the, the]  
4 [This, Java, best]  
7 [academy]  
8 [universe]  
*/
```

OPERACJE TERMINALNE

- COLLECT - PARTYCJONOWANIE

W ramach operacji terminalnej **collect** mamy możliwość partycjonowania (podziału na dwa rozłączne zbioru) elementów strumienia. Służy do tego kolektor uzyskiwany przy pomocy metody **partitioningBy** dostępnej na klasie **Collectors**.

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");  
Map<Boolean, List<String>> map = ohMy.collect(  
    Collectors.partitioningBy(s -> s.length() <= 5));  
System.out.println(map); // {false=[tigers], true=[lions, bears]}
```

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");  
Map<Boolean, List<String>> map = ohMy.collect(  
    Collectors.partitioningBy(s -> s.length() <= 7));  
System.out.println(map); // {false=[], true=[lions, tigers, bears]}
```

OPERACJE TERMINALNE

- FINDFIRST

Metoda **findFirst** umożliwia zakończenie przetwarzania strumienia i pobrania pierwszego dostępnego elementu. Ponieważ taką metodę możemy również wywołać **na pustym strumieniu**, stąd zwracanym typem jest **Optional**, np.:

```
List.of("kto", "będzie", "pierwszym", "elementem").stream()
    .sorted()
    .findFirst() // zwraca Optional
    .ifPresent(System.out::println); // wyświetli słowo "będzie"
```


OPERACJE TERMINALNE

- FINDANY

findAny, podobnie jak **findFirst** zwraca **pojedynczny element strumienia** (jako obiekt opakowany w Optional), jednak w tym przypadku **nie mamy pewności, który element strumienia zostanie zwrócony** w przypadku wielu dostępnych w nim elementów.

```
List.of(7, 21, 13, 4, 8).stream()  
    .filter(x -> x % 2 == 0)  
    .findAny()  
    .ifPresent(System.out::println);
```



OPERACJE TERMINALNE

- ALLMATCH, ANYMATCH

Metody pozwalają na ewaluację predykatu względem elementów tworzących strumień.

```
boolean anyMatch(Predicate <? super T> predicate)
boolean allMatch(Predicate <? super T> predicate)
boolean noneMatch(Predicate <? super T> predicate)
```

```
List<String> list = Arrays.asList("monkey", "2", "chimp");
Stream<String> infinite = Stream.generate(() -> "chimp");
Predicate<String> pred = x -> Character.isLetter(x.charAt(0));
System.out.println(list.stream().anyMatch(pred)); // true
System.out.println(list.stream().allMatch(pred)); // false
System.out.println(list.stream().noneMatch(pred)); // false
System.out.println(infinite.anyMatch(pred)); // true
```



OPERACJE TERMINALNE

- MIN, MAX

Metody **min** oraz **max** pozwalają wyznaczyć wartość minimalną oraz maksymalną elementów tworzących dany strumień (w oparciu o określony komparator).

```
Optional<T> min(<? super T> comparator)  
Optional<T> max(<? super T> comparator)
```

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");  
Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());  
min.ifPresent(System.out::println); // ape
```

```
Optional<?> minEmpty = Stream.empty().min((s1, s2) -> 0);  
System.out.println(minEmpty.isPresent()); // false
```



OPERACJE TERMINALNE

- COUNT

Metoda **count** umożliwia określenie liczby elementów tworzących dany strumień.

```
long count()
```

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");  
System.out.println(s.count());    // 3
```



OPERACJE TERMINALNE

- REDUCE

Operacja **reduce** umożliwia uzyskanie pojedynczego rezultatu ze wszystkich elementów strumienia. **Metoda reduce przyjmuje dwa argumenty:**

- **wartość początkową**
- **sposób transformacji**, tzn. informację, w jaki sposób zmienić aktualny wynik za pomocą kolejnego przetwarzanego elementu. Informację **tę zapisujemy wykorzystując interfejs funkcyjny BiFunction<T,U,R>.**

```
final Integer sum = List.of(2, 5, 9, 19, 14).stream()
    .reduce(0, (currentSum, streamElement) -> currentSum + streamElement); // lub Integer::sum
System.out.println(sum); // wynikiem jest suma - 49
```

OPERACJE TERMINALNE

- REDUCE

```
T reduce(T identity, BinaryOperator<T> accumulator)
Optional<T> reduce(BinaryOperator<T> accumulator)
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
BinaryOperator<U> combiner)
```

```
String[] array = new String[] { "w", "o", "l", "f" };
String result = "";
for (String s: array) result = result + s;
System.out.println(result);
```

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
String word = stream.reduce("", String::concat);
System.out.println(word);    // wolf
```

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> empty = Stream.empty();
Stream<Integer> oneElement = Stream.of(3);
Stream<Integer> threeElements = Stream.of(3, 5, 6);
```

```
empty.reduce(op).ifPresent(System.out::print); // no output
oneElement.reduce(op).ifPresent(System.out::print); // 3
threeElements.reduce(op).ifPresent(System.out::print); // 90
```

OPERACJE TERMINALNE

- PODSUMOWANIE

Method	What Happens for Infinite Streams	Return Value	Reduction
<code>allMatch()</code> <code>/anyMatch()</code> <code>/noneMatch()</code>	Sometimes terminates	boolean	No
<code>collect()</code>	Does not terminate	Varies	Yes
<code>count()</code>	Does not terminate	long	Yes
<code>findAny()</code> <code>/findFirst()</code>	Terminates	Optional<T>	No
<code>forEach()</code>	Does not terminate	void	No
<code>min()/max()</code>	Does not terminate	Optional<T>	Yes
<code>reduce()</code>	Does not terminate	Varies	Yes



STREAM – PORÓWNIANIE ROZWIĄZAŃ

```
List<String> list = Arrays.asList("Toby", "Anna", "Leroy", "Alex");
List<String> filtered = new ArrayList<>();
for (String name: list) {
    if (name.length() == 4) filtered.add(name);
}
Collections.sort(filtered);
Iterator<String> iter = filtered.iterator();
if (iter.hasNext()) System.out.println(iter.next());
if (iter.hasNext()) System.out.println(iter.next());
```

```
List<String> list = Arrays.asList("Toby", "Anna", "Leroy", "Alex");
list.stream().filter(n -> n.length() == 4).sorted()
    .limit(2).forEach(System.out::println);
```

Before you say that it is harder to read, we can format it:

```
stream.filter(n -> n.length() == 4)
    .sorted()
    .limit(2)
    .forEach(System.out::println);
```



STREAM – PORÓWNIANIE ROZWIĄZAŃ

```
Stream.generate(() -> "Elsa")  
  .filter(n -> n.length() == 4)  
  .sorted()  
  .limit(2)  
  .forEach(System.out::println);
```

```
Stream.generate(() -> "Elsa")  
  .filter(n -> n.length() == 4)  
  .limit(2)  
  .sorted()  
  .forEach(System.out::println);
```

```
Stream.generate(() -> "Olaf Lazisson")  
  .filter(n -> n.length() == 4)  
  .limit(2)  
  .sorted()  
  .forEach(System.out::println);
```



STREAM – PORÓWNIANIE ROZWIĄZAŃ

```
Stream<Integer> infinite = Stream.iterate(1, x -> x + 1);  
infinite.limit(5)  
    .filter(x -> x % 2 == 1)  
    .forEach(System.out::print); // 135
```

```
Stream<Integer> infinite = Stream.iterate(1, x -> x + 1);  
infinite.limit(5)  
    .peek(System.out::print)  
    .filter(x -> x % 2 == 1)  
    .forEach(System.out::print);
```

```
Stream<Integer> infinite = Stream.iterate(1, x -> x + 1);  
infinite.filter(x -> x % 2 == 1)  
    .limit(5)  
    .forEach(System.out::print); // 13579
```



STREAM – WYPISYWANIE ELEMENTÓW

Option	Works for Infinite Streams?	Destructive to Stream?
<code>s.forEach(System.out::println);</code>	No	Yes
<code>System.out.println(s.collect(Collectors.toList()));</code>	No	Yes
<code>s.peek(System.out::println).count();</code>	No	No
<code>s.limit(5).forEach(System.out::println);</code>	Yes	Yes



STRUMIENIE - TYPY PRYMITYWNE

API Javy udostępnia dedykowane interfejsy do obsługi strumieni operujących na elementach typu prostego (prymitywach), tj.: int, long, double. Są to:

IntStream, LongStream, DoubleStream.

- `IntStream`: Used for the primitive types int, short, byte, and char
- `LongStream`: Used for the primitive type long
- `DoubleStream`: Used for the primitive types double and float

```
DoubleStream oneValue = DoubleStream.of(3.14);
DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
oneValue.forEach(System.out::println);
System.out.println();
varargs.forEach(System.out::println);
```

```
DoubleStream random = DoubleStream.generate(Math::random);
DoubleStream fractions = DoubleStream.iterate(.5, d -> d / 2);
random.limit(3).forEach(System.out::println);
System.out.println();
fractions.limit(3).forEach(System.out::println);
```

```
IntStream count = IntStream.iterate(1, n -> n+1).limit(5);
count.forEach(System.out::println);
```

```
IntStream range = IntStream.range(1, 6);
range.forEach(System.out::println);
```

```
IntStream rangeClosed = IntStream.rangeClosed(1, 5);
rangeClosed.forEach(System.out::println);
```

STRUMIENIE

- TYPY PRYMITYWNE

Source Stream Class	To Create Stream	To Create DoubleStream	To Create IntStream	To Create LongStream
Stream	map	mapToDouble	mapToInt	mapToLong
DoubleStream	mapToObj	map	mapToInt	mapToLong
IntStream	mapToObj	mapToDouble	map	mapToLong
LongStream	mapToObj	mapToDouble	mapToInt	map

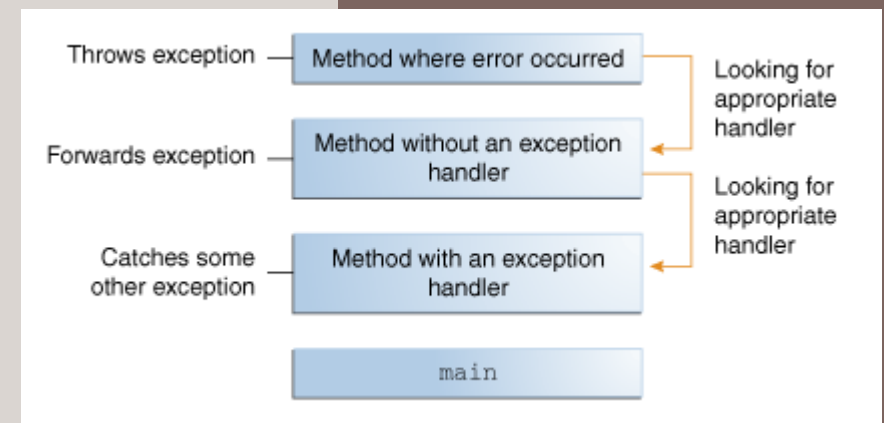
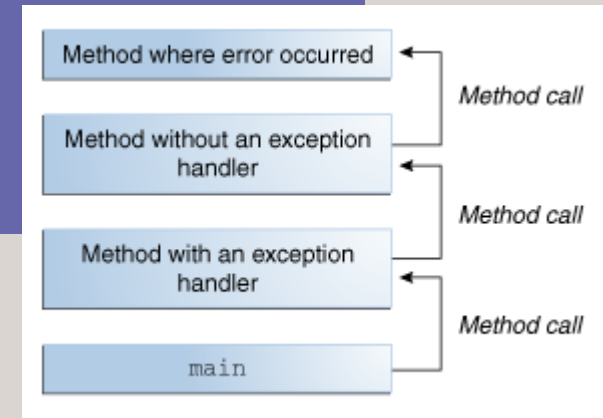
Source Stream Class	To Create Stream	To Create DoubleStream	To Create IntStream	To Create LongStream
Stream	Function	ToDoubleFunction	ToIntFunction	ToLongFunction
DoubleStream	Double Function	DoubleUnary Operator	DoubleToInt Function	DoubleToLong Function
IntStream	IntFunction	IntToDouble Function	IntUnary Operator	IntToLong Function
LongStream	Long Function	LongToDouble Function	LongToInt Function	LongUnary Operator

WYJĄTKI I ASERCJE

Wyjątek jest zdarzeniem, które następuje podczas wykonywania programu i **łamie normalny proces jego działania**. Po tym, jak wyjątek zostanie wyrzucony, system wykonawczy próbuje znaleźć fragment kodu, który sobie z tym poradzi. Zestawem możliwych elementów do obsługi wyjątku jest uporządkowana lista metod, które zostały wywołane, aby dostać się do metody, w której wystąpił błąd. **Lista metod jest znana jako stos wywołań** (ang. stacktrace).

WYJĄTKI

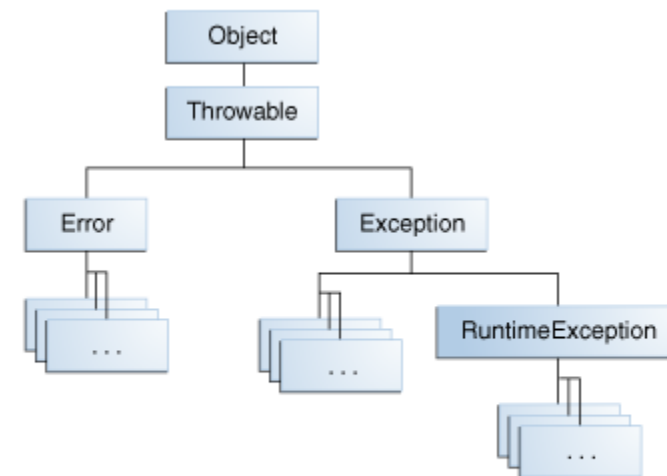
System w trakcie wykonywania **przeszukuje stos wywołań** w celu znalezienia metody, która obsłuży wyjątek. Wyszukiwanie rozpoczyna się od metody, w której wyrzucony został wyjątek, a kończy się na metodzie, która jest odpowiedzialna za obsługę zdarzenia.



TYPY WYJĄTKÓW

W Javie wszystkie wyjątki możemy podzielić na dwie główne grupy. Grupy te zawierają wyjątki, które:

- dziedziczą po klasie **RuntimeException**, tzw. **wyjątki unchecked**
- dziedziczą po klasie **Exception**, tzw. **wyjątki checked**



UWAGA: Każdy wyjątek dziedziczy po klasie **Throwable**.

TYPY WYJĄTKÓW

Type	How to recognize	Recommended for program to catch?	Is program required to catch or declare?
Runtime exception	<code>RuntimeException</code> or its subclasses	Yes	No
Checked exception	Exception or its subclasses but not <code>RuntimeException</code> or its subclasses	Yes	Yes
Error	<code>Error</code> or its subclasses	No	No

PODSTAWOWE WYJĄTKI

- **ArithmeticException** – dzielenie przez zero
- **ArrayIndexOutOfBoundsException** – nieprawidłowy indeks tablicy
- **ClassCastException** – nieprawidłowe rzutowanie klasy
- **IllegalArgumentException** – do metody przekazano nieprawidłowy argument
- **NullPointerException** – otrzymano wartość nieokreśloną **null** podczas, gdy oczekiwano obiektu
- **NumberFormatException** – błąd konwersji łańcucha znaków do liczby (String -> numeric type)



CHECKED EXCEPTIONS

Exception	Used when	Checked or unchecked?
<code>java.text.ParseException</code>	Converting a String to a number.	Checked
<code>java.io.IOException</code> <code>java.io.FileNotFoundException</code> <code>Exception</code> <code>java.io.NotSerializable</code> <code>Exception</code>	Dealing with IO and NIO.2 issues. <code>IOException</code> is the parent class. There are a number of subclasses. You can assume any <code>java.io</code> exception is checked.	Checked
<code>java.sql.SQLException</code>	Dealing with database issues. <code>SQLException</code> is the parent class. Again, you can assume any <code>java.sql</code> exception is checked.	Checked



RUNTIME EXCEPTIONS

Exception	Used when	Checked or unchecked?
<code>java.lang.ArrayStoreException</code>	Trying to store the wrong data type in an array.	Unchecked
<code>java.time.DateTimeException</code>	Receiving an invalid format string for a date.	Unchecked
<code>java.util.MissingResourceException</code>	Trying to access a key or resource bundle that does not exist.	Unchecked
<code>java.lang.IllegalStateException</code> <code>java.lang.UnsupportedOperationException</code>	Attempting to run an invalid operation in collections and concurrency.	Unchecked



PRZECHWYTYWANIE I OBSŁUGA WYJĄTKÓW

Jeżeli wywołanie pewnej metody wyrzuca wyjątek **typu checked**, możemy taki wyjątek:

- nie obsługiwać i przekazać „wyżej” (tzn. **spropagować wyjątek w górę**, zgodnie ze stosem wywołań)
- obsłużyć w obrębie metody.

Aby przekazać konieczność obsługi wyjątków, musimy w sygnaturze metody dodać **słowo kluczowe throws**, a po nim listę typów wyjątków, jakie ta metoda może wyrzucić.

```
public void sleepAndOpenFile(final String filePath) throws InterruptedException, FileNotFoundException {  
    Thread.sleep(10);  
    new FileReader(filePath);  
}
```

PRZECHWYTYWANIE I OBSŁUGA WYJĄTKÓW

W **celu obsługi wyjątku**, tzn. zrobienia czegoś z faktem jego wystąpienia, musimy wykorzystać pewne słowa kluczowe:

- **try** wraz z **catch** i opcjonalnie **finally**
- **try** wraz z przekazaniem zasobów
(**try-with-resources**)



TRY | CATCH

Pierwszym krokiem w kierunku obsługi wyjątków jest zdeklarowanie bloku **try** wraz z blokiem **catch**. W bloku **try** definiujemy **kod, który może wyrzucić wyjątki pewnego typu**, następnie w bloku **catch**, który następuje zaraz po bloku try, definiujemy **kod, który powinien zostać wykonany w przypadku wystąpienia wyjątku**.

Typ wyjątku i jego instancję podajemy wewnątrz nawiasów, zaraz po słowie kluczowym catch, np.:

```
try {  
    Thread.sleep(100L);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

A finally block can only appear as part of a try statement.

```
try{  
    //protected code  
}catch(exceptiontype identifier){  
    //exception handler  
}finally{  
    //finally block  
}
```

The finally keyword

The finally block always executes, whether or not an exception occurs in the try block.

TRY | CATCH

Do pojedynczego bloku try, możemy dodać **wiele bloków catch**. Każdy z bloków **catch może obsługiwać inny typ wyjątku**. Tworząc bloki catch musimy pamiętać o kilku faktach:

- możemy złapać **dowolny wyjątek typu unchecked**
- możemy złapać tylko te **wyjątki typu checked, które mogą zostać wyrzucone w obrębie bloku try**
- możemy złapać typ będący podklasą typu wyjątku
- możemy łapać wyjątki, które po sobie dziedziczą, ale **bloki catch z bardziej specyficznym wyjątkiem muszą pojawić się jako pierwsze**

```
try {  
    Thread.sleep(100L);  
    new RestTemplate().getForEntity("http://localhost:8080", Object.class);  
} catch (RestClientException e) {  
    System.err.println("Error occurred from RestTemplate object");  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

```
public class CatchHierarchy {  
    public static void main(String[] args) {  
        try {  
            final KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");  
            final Cipher cipher = Cipher.getInstance("AES");  
        } catch (NoSuchAlgorithmException e) {  
            System.err.println("Incorrect algorithm name");  
        } catch (NoSuchPaddingException e) {  
            System.err.println("Oops");  
        } catch (Exception e) {  
            System.err.println("Generic exception occurred");  
        }  
    }  
}
```


TRY | CATCH

Co jeżeli chcemy wiele typów wyjątków **obsłużyć w dokładnie taki sam sposób**? Nie musimy w tym celu definiować wielu bloków catch. Wystarczy jeden, a **listę wyjątków które chcemy obsłużyć za pomocą tego samego kodu, oddzielamy za pomocą znaku |**

UWAGA: Wykorzystując znak | w bloku catch, **nie możemy podać dwóch klas z jednej hierarchii** (tzn. jedna z nich jest pochodną drugiej). Skończy się to błędem kompilacji, np.: `catch (Exception | RuntimeException e)`.

```
public class MultiExceptionsSingleCatch {  
    public static void main(String[] args) {  
        try {  
            final KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");  
            final Cipher cipher = Cipher.getInstance("AES");  
        } catch (NoSuchPaddingException | NoSuchAlgorithmException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

TRY | CATCH

```
try {  
    // do some work  
} catch(RuntimeException e) {  
    e = new RuntimeException();  
}
```

```
try {  
    throw new IOException();  
} catch(IOException | RuntimeException e) {  
    e = new RuntimeException();           // DOES NOT COMPILE  
}
```

```
public static void main(String[] args) {  
    try {  
        Path path = Paths.get("dolphinsBorn.txt");  
        String text = new String(Files.readAllBytes(path));  
        LocalDate date = LocalDate.parse(text);  
        System.out.println(date);  
    } catch (DateTimeParseException | IOException e) {  
        e.printStackTrace();  
        throw new RuntimeException(e);  
    } }
```

TRY | CATCH

Dlaczego poniższy kod nie kompiluje się?

```
11: public void doesNotCompile() { // METHOD DOES NOT COMPILE
12:     try {
13:         mightThrow();
14:     } catch (FileNotFoundException | IllegalStateException e) {
15:     } catch (InputMismatchException e | MissingResourceException e) {
16:     } catch (SQLException | ArrayIndexOutOfBoundsException e) {
17:     } catch (FileNotFoundException | IllegalArgumentException e) {
18:     } catch (Exception e) {
19:     } catch (IOException e) {
20:     }
21: }
22: private void mightThrow() throws DateTimeParseException, IOException { }
```

FINALLY

Aplikacje operują skończoną ilością zasobów (np. ilość pamięci, maksymalna ilość plików, czy połączeń do bazy danych). Jeżeli w trakcie wykonywania kodu pewne **zasoby zostały otwarte i następnie został wyrzucony wyjątek, mimo to powinniśmy je zawsze zamknąć.**

Mechanizm ten zapewnia blok **finally**, który wywołuje się **zawsze, gdy istnieje blok try**, niezależnie od tego, czy został wyrzucony wyjątek czy nie. **Block finally znajduje się zawsze na końcu bloków try/catch.**

- Blok **try** wymaga bloku **catch** i/lub bloku **finally**.
- Blok **finally** wykona się **nawet, jeżeli w bloku try wyjdziemy z metody za pomocą return.**

```
public class FinallyExample {  
    public static void main(String[] args) throws IOException {  
        BufferedReader bufferedReader = null;  
        try {  
            bufferedReader = new BufferedReader(new FileReader("/tmp/java_course.txt"));  
        } catch (FileNotFoundException e) {  
            System.exit(1);  
        } finally {  
            if (bufferedReader != null) {  
                bufferedReader.close();  
            }  
        }  
    }  
}
```

TRY Z ZASOBEM

Istnieje również możliwość przekazania do bloku try **listy zasobów**. Wykorzystanie bloku try w taki sposób **zwalnia nas z ręcznego zamykania zasobów w bloku finally**.

Zasobem takim może być dowolny obiekt, który implementuje **java.lang.AutoCloseable**, bądź realizuje **java.io.Closeable**.

```
public String readFirstLineFromFile(String path) {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    } catch (IOException e) {  
        return "FAILED";  
    }  
}
```

```
public class Turkey {  
    public static void main(String[] args) {  
        try (Turkey t = new Turkey()) { // DOES NOT COMPILE  
            System.out.println(t);  
        }  
    }  
}
```

TRY Z ZASOBEM

```
try (BufferedReader r = Files.newBufferedReader(path1);  
    BufferedWriter w = Files.newBufferedWriter(path2)) {  
    //protected code  
  
} catch (IOException e) {  
    // exception handler  
  
} finally {  
    // finally block  
  
}
```

Optional clauses; resources still closed automatically



TRY-CATCH-FINALLY - KONFIGURACJE

traditional try statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Not legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

try-with-resources statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

RZUCANIE WYJĄTKÓW

W kodzie źródłowym, oprócz przechwytywania zgłoszonych już wyjątków, **istnieje możliwość rzucania wyjątków**. Wyjątek taki musi być **po pochodną klasy `Throwable`**. Do rzucania wyjątków wykorzystujemy klauzulę **`throw`**

```
public void validateAddress(String address) {  
    if (address == null || address.isEmpty()){  
        throw new IllegalArgumentException("illegal address");  
    }  
    // do some operations  
}
```


THROW VS THROWS

```
public String getDataFromDatabase() throws SQLException {  
    throw new UnsupportedOperationException();  
}
```



STŁUMIONE WYJĄTKI (SUPPRESSED EXCEPTIONS)

```
public class JammedTurkeyCage implements AutoCloseable {  
    public void close() throws IllegalStateException {  
        throw new IllegalStateException("Cage door does not close");  
    }  
    public static void main(String[] args) {  
        try (JammedTurkeyCage t = new JammedTurkeyCage()) {  
            System.out.println("put turkeys in");  
        } catch (IllegalStateException e) {  
            System.out.println("caught: " + e.getMessage());  
        }  
    }  
}
```

```
try (JammedTurkeyCage t = new JammedTurkeyCage()) {  
    throw new IllegalStateException("turkeys ran off");  
} catch (IllegalStateException e) {  
    System.out.println("caught: " + e.getMessage());  
    for (Throwable t: e.getSuppressed())  
        System.out.println(t.getMessage());  
}
```



TWORZENIE WŁASNYCH WYJĄTKÓW

Java zapewnia wiele klas wyjątków, z których można skorzystać. **Można też tworzyć własne wyjątki.** Jeśli któreś z poniższych kryteriów zostanie spełnione, warto jest rozważyć przygotowanie własnego typu:

- istnieje potrzeba posiadania wyjątku, który **nie jest reprezentowany przez żaden dostępny typ wyjątku**
- wyjątek wykorzystywany w kodzie powinien **różnić się od wyjątków innych dostawców.**

```
public class JavaCourseException extends RuntimeException {  
    public JavaCourseException(final String message, final Throwable cause) {  
        super(message, cause);  
    }  
}  
  
public class IllegalAddressException extends IllegalArgumentException {  
    public IllegalAddressException(final String address) {  
        super(String.format("Provided address %s is not valid!", address));  
    }  
}
```

ERROR

Instancje klasy **Error** są wyjątkami, których **nie powinniśmy obsługiwać przy pomocy bloku try i catch**. Nie musimy też deklarować informacji o takim błędzie przy pomocy throws. Oznaczają one błędy, z którymi **nie poradzimy sobie z poziomu aplikacji i najczęściej ich efektem jest zakończenie działania danej aplikacji**. Przykładem takiego typu błędu jest wyjątek **OutOfMemoryError**.



ASERCJE

W najbardziej ścisłym ujęciu można powiedzieć, że asercja jest **predykatem**, czyli **stwierdzeniem, które może być prawdziwe, lub fałszywe**. W języku JAVA istnieje **słowo kluczowe assert**, które pozwala nam stwierdzić, że pewne wyrażenie jest **prawdziwe, lub wygenerować błąd w przeciwnym wypadku**.

Głównym zastosowaniem asercji jest **wykonywanie rozmaitych testów**.



ASERCJE

Domyślnie sprawdzanie asercji **jest wyłączone** w języku JAVA i nie zaleca się używania ich poza etapem wytwarzania oprogramowania. W celu włączenia asercji możemy posłużyć się następującymi poleceniami:

```
java -enableassertions Rectangle
```

```
java -ea Rectangle
```

Użycie flagi **-ea** bez dodatkowych argumentów powoduje odblokowanie wszystkich asercji (za wyjątkiem klas systemowych javy).



ASERCJE

- włączenie asercji dla **pakietu com.wiley.demos** oraz odpowiednich **podpakietów**:

```
java -ea:com.wiley.demos... my.programs.Main
```

- włączenie asercji dla klasy com.wiley.demos.TestColors:

```
java -ea:com.wiley.demos.TestColors my.programs.Main
```

- włączenie asercji dla pakietu com.wiley.demos oraz wyłączenie asercji dla klasy TestColors:

```
java -ea:com.wiley.demos... -da:com.wiley.demos.TestColors my.programs.Main
```



POLECENIE ASSERT

```
assert boolean_expression;  
assert boolean_expression: error_message;
```

- **asercje wyłączone** – instrukcje **assert** są pomijane w trakcie wykonywania kodu
- **asercje włączone** – wyrażenie **boolean_expression** ewaluuje do **true** – aplikacja wykonuje się bez zmian
- **asercje włączone** - wyrażenie **boolean_expression** ewaluuje do **false** – aplikacja rzuca wyjątek

AssertionError


```
if (!boolean_expression) throw new AssertionError();
```

```
public class Assertions {  
    public static void main(String[] args) {  
        int numGuests = -5;  
        assert numGuests > 0;  
        System.out.println(numGuests);  
    }  
}
```



ASERCJE - UŻYCIE

```
public enum Seasons {  
    SPRING, SUMMER, FALL  
}
```

```
public class TestSeasons {  
    public static void test(Seasons s) {  
        switch (s) {  
            case SPRING:  
            case FALL:  
                System.out.println("Shorter hours");  
                break;  
            case SUMMER:  
                System.out.println("Longer hours");  
                break;  
            default:  
                assert false: "Invalid season";  
        }  
    }  
}
```



```
Exception in thread "main" java.lang.AssertionError: Invalid season  
    at TestSeason.main(Test.java:12)  
    at TestSeason.main(Test.java:18)
```



ASERCJE - UŻYCIE

- Asercje nie powinny modyfikować stanu zmiennych oraz obiektów

```
int x = 10;  
assert ++x > 10; // Not a good design!
```

- Asercje wykorzystujemy do debugowania
- Nie powinniśmy używać asercji do walidowania parametrów wejściowych

```
public Rectangle(int width, int height) {  
    if(width < 0 || height < 0) {  
        throw new IllegalArgumentException();  
    }  
    this.width = width;  
    this.height = height;  
}
```



DZIĘKUJĘ ZA UWAGĘ

