

Debugowanie

Czego się dowiesz

dlaczego nadużywanie `System.out.println` to nie najlepszy pomysł,
co oznacza debugowanie aplikacji,
jak wykorzystać debugger wbudowany w IntelliJ IDEA.

Wstęp

Na początku swojej przygody z programowaniem większość osób do znajdowania błędów w kodzie wykorzystuje metodę `System.out.println()`. W przypadku prostych aplikacji nie jest to problemem i podejście takie zazwyczaj pozwala dosyć sprawnie dojść do źródła problemu. Z czasem, gdy aplikacja staje się coraz większa, pojawiają się jednak dodatkowe trudności, które trzeba brać pod uwagę.

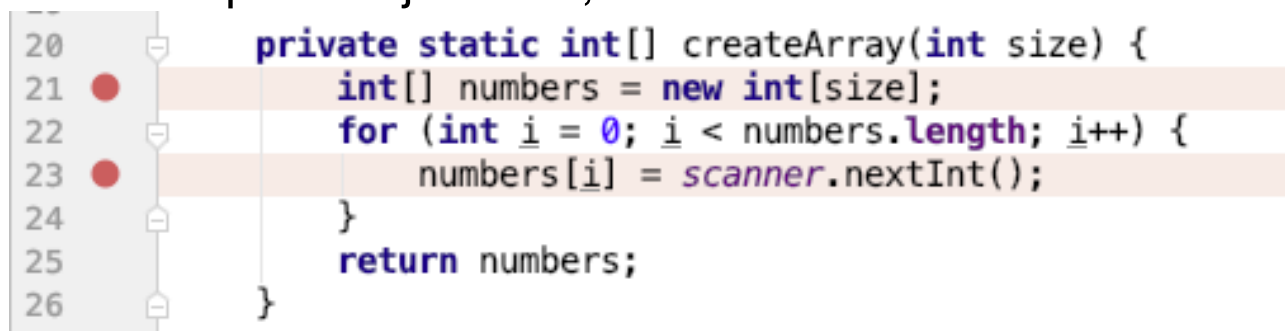
Duże aplikacje, które składają się z setek, czy tysięcy klas potrafią się uruchamiać kilkadziesiąt sekund, czasami nawet kilka minut. Szukanie błędów poprzez dodawanie w programie kolejnych wywołań metody `System.out.println()` staje się w takiej sytuacji bardzo mało wygodne i sprawia, że większość czasu marnujemy na oczekiwanie, aż aplikacja wystartuje, a nie na analizę problemu i poszukiwanie źródła błędu. Lepszym rozwiązaniem jest wykorzystanie debuggera, czyli specjalnego narzędzia, najczęściej wbudowanego w środowisko programistyczne (np. IntelliJ IDEA lub eclipse), które umożliwia wstrzymanie działania programu w dowolnym momencie jego wykonywania, a następnie pozwala śledzić wszystkie zmiany, które dzieją

się w programie wiersz po wierszu. Dzięki temu nie musisz co chwilę restartować aplikacji, aby zobaczyć, czy wyniki w konsoli już są poprawne. Debugowanie swoją nazwę wzięło od słowa "bug", co dosłownie oznacza "robak", ale w języku angielskim jest to potoczna nazwa na błąd w programie. Debugowanie jest więc procesem znajdowania i eliminowania błędów w kodzie Twojej aplikacji.

Debugger w IntelliJ IDEA

Każde popularne środowisko programistyczne takie jak IntelliJ IDEA, eclipse, czy netbeans, posiada wbudowany debugger. Działa on na takiej zasadzie, że środowisko wpina się do wirtualnej maszyny, na której uruchamiany jest program i pozwala wstrzymywać lub kontynuować działanie programu, korzystając z wygodnego interfejsu użytkownika. jednym z kluczowych elementów przy korzystaniu z debuggera są tzw. **breakpointy**, czyli punkty wstrzymania programu. Dzięki nim możemy wskazać miejsca w kodzie, w których chcielibyśmy wstrzymać wykonanie kodu, np. w celu sprawdzenia wartości poszczególnych zmiennych czy stanu jakiegoś obiektu.

W środowisku programistycznym breakpointy są najczęściej reprezentowane jako czerwone kropki stawiane po lewej stronie, zaraz obok kodu.



```
20 private static int[] createArray(int size) {
21     int[] numbers = new int[size];
22     for (int i = 0; i < numbers.length; i++) {
23         numbers[i] = scanner.nextInt();
24     }
25     return numbers;
26 }
```

The screenshot shows a code editor with a Java method `createArray`. Two red circular breakpoints are placed on the left margin, one next to line 21 (`int[] numbers = new int[size];`) and another next to line 23 (`numbers[i] = scanner.nextInt();`). The code is highlighted with alternating light orange and light blue background colors.

Dalszą część lekcji przejdźmy omawiając konkretny przykład. Utwórz w IntelliJ nowy projekt i skopiuj poniższą klasę.

DebugExample.java

```
import java.util.Scanner;

class DebugExample {
    private static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        int size = getArraySize();
        int[] numbers = createArray(size);
        int multiplier = getMultiplier();
        multiplyAllNumbersBy(numbers, multiplier);
        printArray(numbers);
    }

    private static int getArraySize() {
        System.out.println("Podaj ilość liczb do wczytania:");
        int result = scanner.nextInt();
        scanner.close();
        return result;
    }

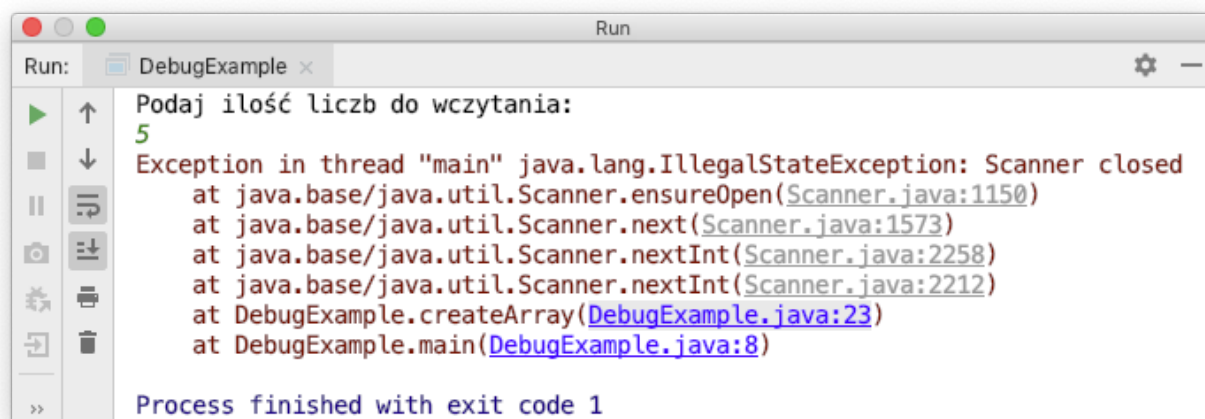
    private static int[] createArray(int size) {
        int[] numbers = new int[size];
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Podaj kolejną liczbę:");
            numbers[i] = scanner.nextInt();
        }
        return numbers;
    }

    private static int getMultiplier() {
        System.out.println("Podaj mnożnik:");
        return scanner.nextInt();
    }

    private static void multiplyAllNumbersBy(int[] array, int multiplier)
{
        for (int number : array) {
            number = number * multiplier;
        }
    }

    private static void printArray(int[] numbers) {
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

Powyższy program wczytuje od użytkownika dowolną liczbę całkowitą. Liczba ta jest rozmiarem tablicy, która tworzona jest w metodzie *createArray()*. Po utworzeniu tablicy przekazujemy ją do metody *multiplyAllNumbersBy()* wraz z dodatkowym argumentem wczytanym przez metodę *getMultiplier()* w celu przemnożenia każdego elementu o wskazaną wartość. Na końcu zawartość tablicy wyświetlana jest z wykorzystaniem metody *printArray()*. Po uruchomieniu programu w konsoli zobaczysz mniej więcej taki błąd:

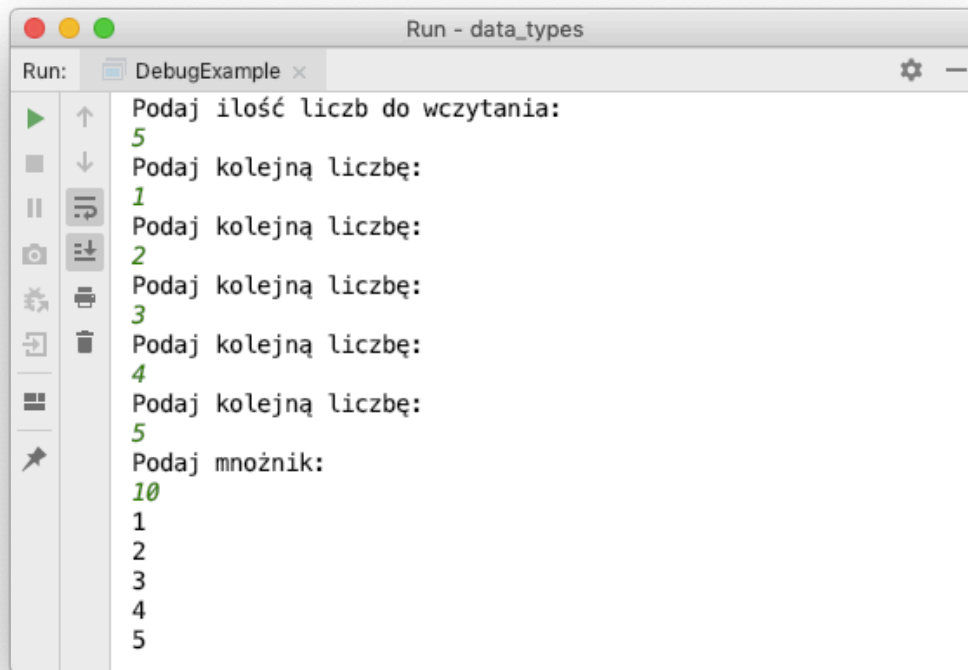


```
Run: DebugExample x
Podaj ilość liczb do wczytania:
5
Exception in thread "main" java.lang.IllegalStateException: Scanner closed
    at java.base/java.util.Scanner.ensureOpen(Scanner.java:1150)
    at java.base/java.util.Scanner.next(Scanner.java:1573)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at DebugExample.createArray(DebugExample.java:23)
    at DebugExample.main(DebugExample.java:8)

Process finished with exit code 1
```

Przyczynę błędu łatwo zlokalizować na podstawie komunikatu, który się pojawił, czyli "*Scanner closed*". Skoro problemem jest to, że Scanner jest zamknięty, to rozwiązanie jest proste i po prostu wystarczy usunąć wywołanie metody *scanner.close()* z metody *getArraySize()*.

No dobrze, ale do tego nie był nam potrzebny debugger. Jeśli uruchomisz program jeszcze raz, to na pierwszy rzut oka wszystko niby wygląda ok, nie ma żadnego błędu, ale wynik jest niepoprawny.



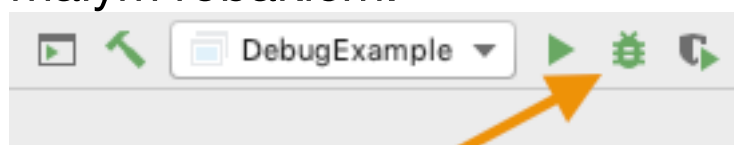
Wyświetlone zostały liczby 1, 2, 3, 4, 5, a skoro podaliśmy mnożnik 10, to oczekiwany wynik to 10, 20, 30, 40, 50. Na podstawie takiego wyniku możemy wyciągnąć wniosek, że liczby zostały poprawnie wczytane od użytkownika, ale albo mnożnik nie jest poprawnie wczytywany w metodzie *getMultiplier()* (np. z jakiegoś powodu zawsze wynosi 1), albo mnożenie kolejnych elementów tablicy w metodzie *multiplyAllNumbersBy()* nie działa. Wielu początkujących programistów w tym momencie zaczyna wszędzie gdzie to możliwe wrzucać instrukcje *System.out.println()*, np.:

```
public static void main(String[] args) {  
    int size = getArraySize();  
    int[] numbers = createArray(size);  
    int multiplier = getMultiplier();  
    System.out.println(multiplier);  
    multiplyAllNumbersBy(numbers, multiplier);  
    printArray(numbers);  
}
```

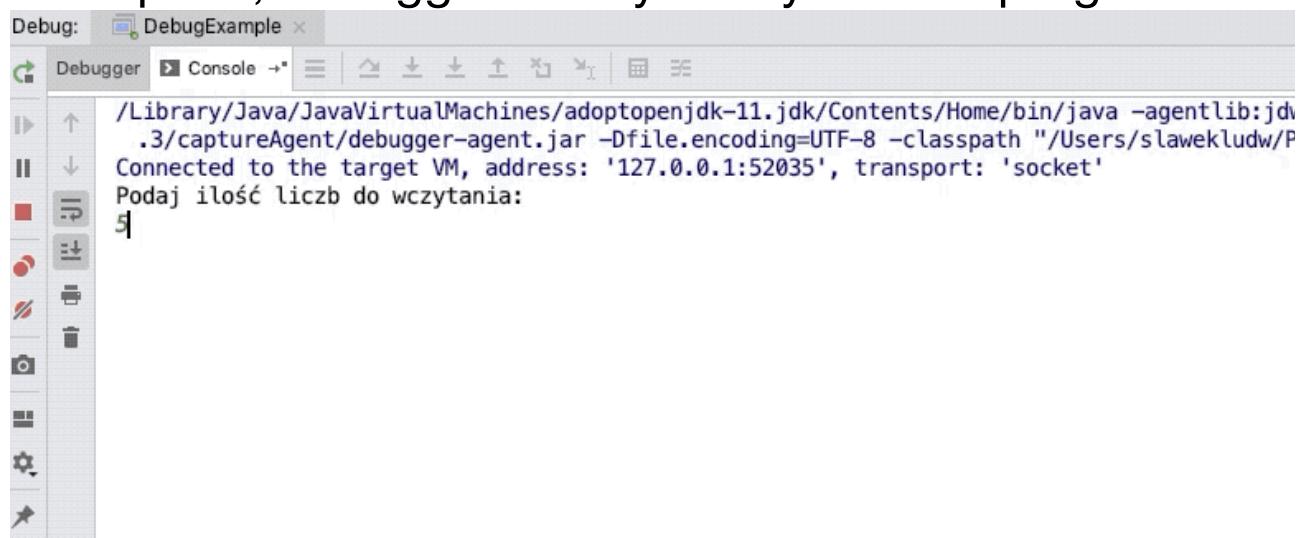
Na tej podstawie oczywiście można dojść do wniosku, że mnożnik jest poprawnie wczytany, ale zamiast korzystać z instrukcji `System.out.println()` lepiej obok wiersza z wczytywaniem mnożnika wstawić breakpoint:

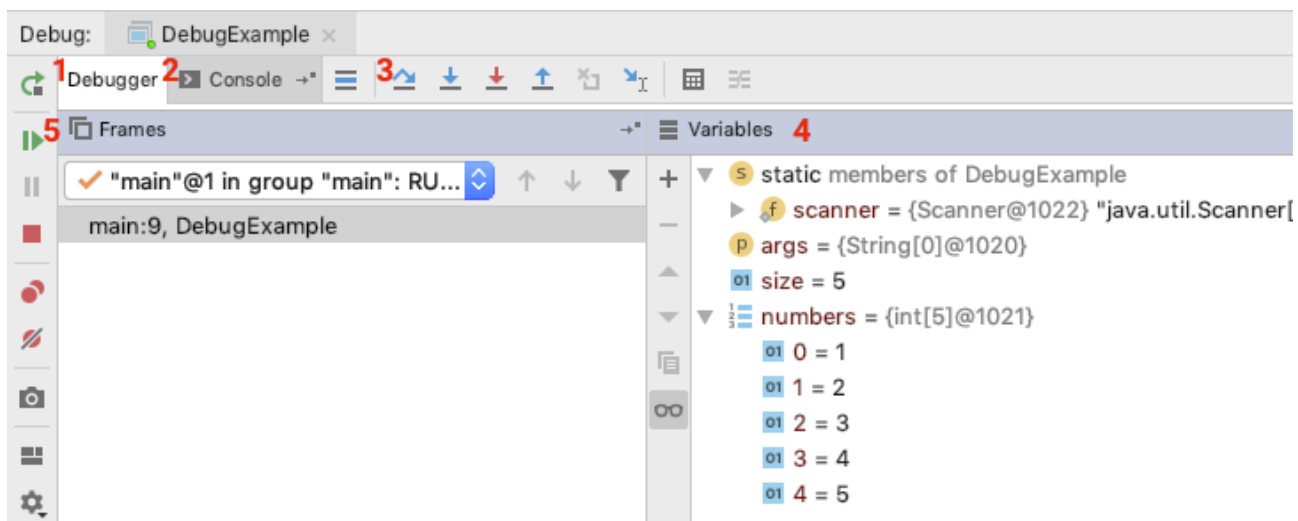
```
6  ▶ public static void main(String[] args) {  
7      int size = getArraySize();  
8      int[] numbers = createArray(size);  
9      int multiplier = getMultiplier();  
10     multiplyAllNumbersBy(numbers, multiplier);  
11     printArray(numbers);  
12 }
```

Jeśli uruchomisz program w trybie debugowania, czyli zamiast wciskając zieloną strzałkę wybierzesz przycisk z małym robakiem:



to zauważysz, że w dolnej części IntelliJ pojawi się konsola debuggera, czyli specjalna przestrzeń, w której wyświetlane są wartości zmiennych lokalnych i pól, do których mamy dostęp w metodzie. Po wpisaniu w konsoli wielkości tablicy oraz następnie kilku liczb, gdy program dojdzie do wiersza, w którym wstawiony był breakpoint, debugger wstrzyma wykonanie programu.





Widok debuggera składa się z dwóch zakładek:

1. okno debuggera,
2. konsola z wydrukami aplikacji, gdzie możemy też wpisywać dane wymagane np. przez Scannera.

Obok zakładek znajdują się przyciski (3), które pozwalają sterować wykonaniem programu. Omówimy je za chwilę.

Po prawej stronie widoczna jest lista pól klasy i zmiennych lokalnych metody, do których mamy dostęp w danym miejscu programu (4). Jeżeli jakiś element jest obiektem, lub tablicą, to można go rozwinąć i zajrzeć jakie wartości przechowuje.

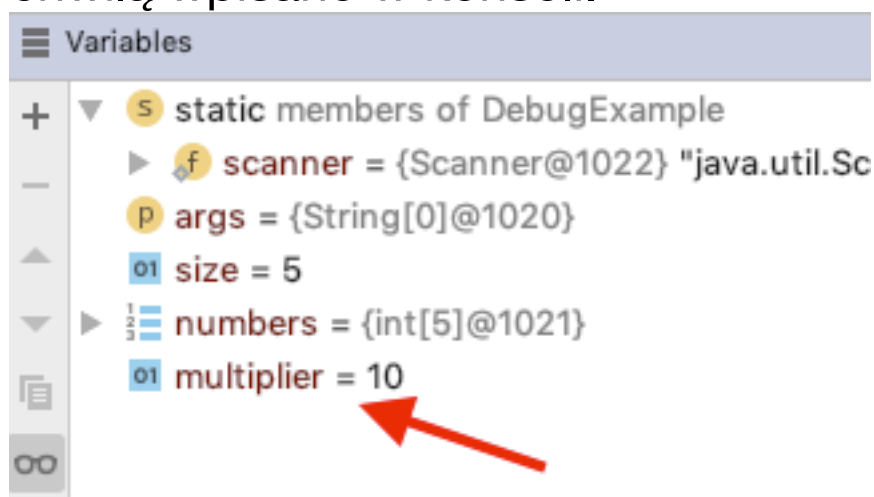
W naszym przypadku widać, że na razie wartość zmiennej `size` to 5, a tablica `numbers` jest wypełniona liczbami wpisanymi przez użytkownika, więc wszystko wygląda ok. Masz teraz do wyboru kilka opcji żeby wznowić działanie programu. Jeśli wciśniesz przycisk zielonej strzałki po lewej stronie (5), to program wykona się dalej, aż nie natrafi na kolejny breakpoint. W naszym przypadku będzie to mało użyteczne, bo w kodzie nie mamy więcej breakpointów, zamiast tego chcielibyśmy sprawdzić, czy metoda `getMultiplier()` zwraca poprawną wartość.

Jeśli nie interesuje nas to co dzieje się wewnątrz metody, to możemy wybrać opcję Step Over, czyli pierwszą z opcji w sekcji (3).



Zamiast klikać na przyciski z IntelliJ, możesz także wcisnąć klawisz F8. W tym momencie musisz przenieść się do zakładki Console (2) i wpisać dowolną wartość mnożnika.

Jeśli wrócisz teraz do zakładki Debugger, to w sekcji ze zmiennymi zobaczysz, że pojawiła się tam zmienna *multiplier* z przypisaną wartością, którą przed chwilą wpisano w konsoli.



Jest to potwierdzenie tego, że wartość od użytkownika wczytana jest poprawnie, więc problem raczej leży gdzieś dalej.

Zauważ, że program jeszcze nie zakończył swojego działania. tylko przeszedł do kolejnego wiersza. W tym momencie masz do wyboru kilka opcji:

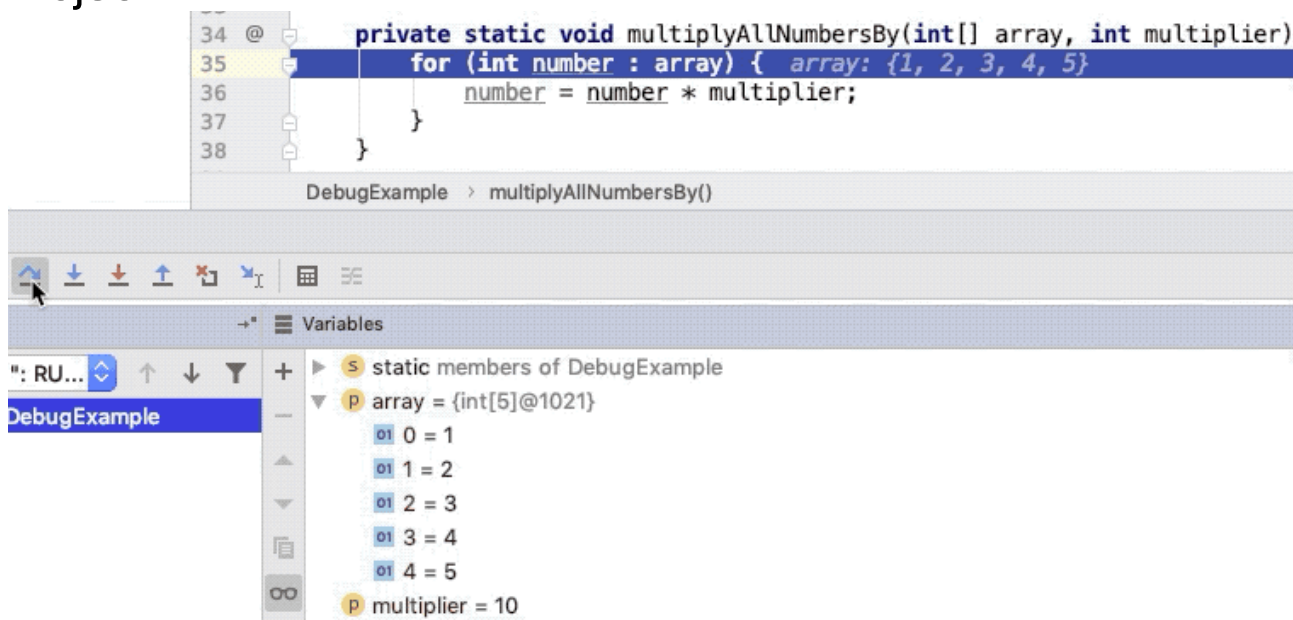
- wcisnąć strzałkę po lewej stronie okna debuggera (5) i pozwolić programowi się wykonać do końca (bo nie mamy więcej breakpointów),
- zatrzymać program wciskając czerwony kwadrat,

wybrać opcję Step Over, aby pozwolić się wykonać metodzie *multiplyAllNumbersBy()* i przejść do kolejnego wiersza w metodzie *main()*, wejść do metody *multiplyAllNumbersBy()* i śledzić jej wykonanie w poszukiwaniu błędów.

Skorzystajmy z ostatniej opcji. W tym celu klikamy drugą ze strzałek w sekcji (3), czyli Step Into lub wciskamy klawisz F7.



Funkcja Step Into powoduje wejście do metody znajdującej się w danym wierszu kodu. W naszym przypadku do metody *multiplyAllNumbersBy()*. Później możemy śledzić wykonanie tej metody, ponownie korzystając z opcji Step Over lub Step Into, jeżeli znowu pojawiłyby się kolejne metody, do których chcemy wejść.

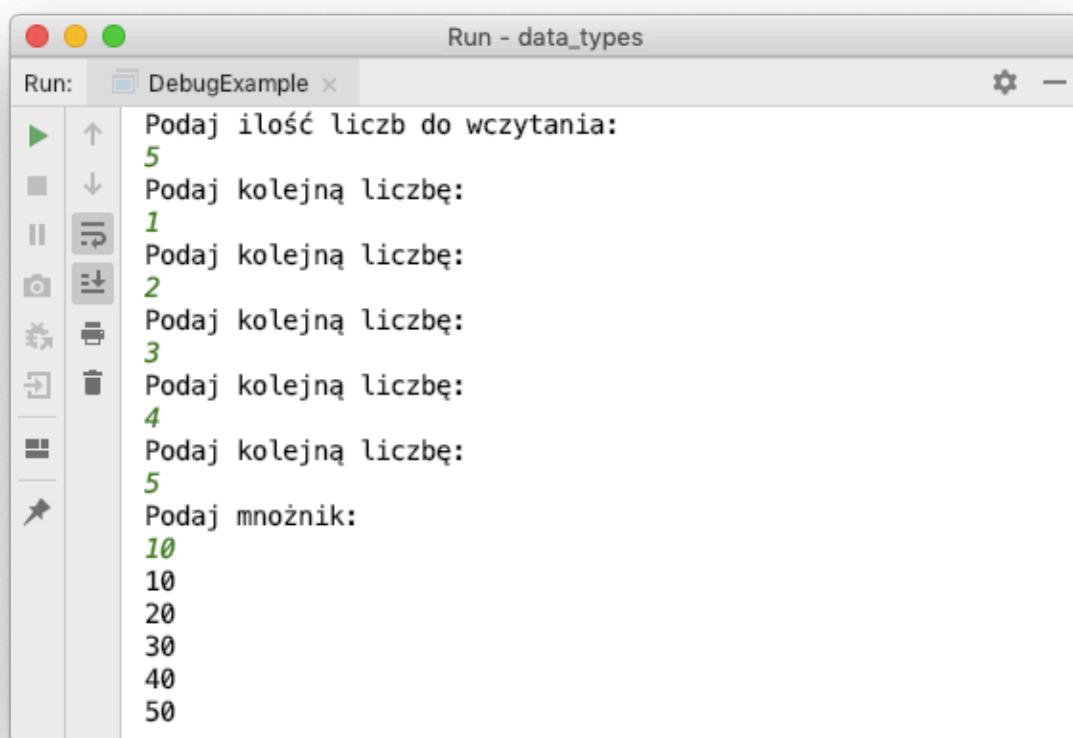


To co można wywnioskować na podstawie powyższego wyniku, to metoda *multiplyAllNumbersBy()* faktycznie się wykonuje i iteruje po kolejnych elementach tablicy, ale z jakiegoś powodu zawartość tablicy nie ulega zmianie.

Znaleźliśmy źródło błędu i jest nim pętla for each, która nadaje się świetnie do przeglądania zawartości tablicy, ale nie może być używana do wstawiania do niej nowych elementów. Rozwiązaniem jest zamiana pętli for each na tradycyjną pętlę for.

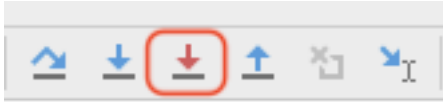
```
private static void multiplyAllNumbersBy(int[] array, int multiplier) {  
    for (int i = 0; i < array.length; i++) {  
        array[i] = array[i] * multiplier;  
    }  
}
```

Po ponownym uruchomieniu programu zobaczysz już poprawny wynik i liczby wprowadzone przez użytkownika będą przemnożone przez podaną przez niego wartość.



Pomimo uruchomienia programu tylko raz, byliśmy w stanie prześledzić działanie kolejnych metod. Rozpoczęliśmy od zbadania tropu z tym, że problemem

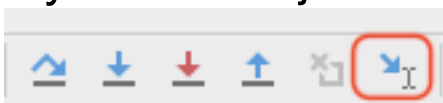
był błędnie wczytany mnożnik, a ostatecznie skończyliśmy na wykryciu błędu w innej metodzie. Oprócz opcji Step Over i Step Into na pasku narzędzi znajduje się jeszcze kilka opcji.



Force Step Into służy do wejścia do metod, które są pomijane przez opcję Step Into. Są to metody wynikające z ustawień IntelliJ Preferences > Build, Execution, Deployment > Debugger > Stepping. Domyślnie znajdują się tam głównie klasy wchodzące w skład pakietu JDK. Przykładowo jeśli postawisz breakpoint obok wywołania metody *nextInt()* klasy *Scanner*, to opcja Step Into nie spowoduje wejścia do tej metody, ale Force Step Into już tak. Dzięki temu możesz zobaczyć jak wygląda wczytanie liczby, czy napisu "pod spodem".



Step Out, jak nazwa sugeruje, powoduje wyjście z metody, w której znajduje się debugger, do miejsca wywołania tej metody.



Run to Cursor spowoduje przejście do wiersza, w którym aktualnie znajduje się kursor i wstrzyma w nim wykonywanie programu. Jest to wygodna opcja, która pozwala uniknąć wstawiania w kodzie zbyt wielu breakpointów.