# Testy jednostkowe (Unit tests)

Konrad Stępniak

# Czym są testy jednostkowe?

- Test jednostkowy (ang. unit test) to metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów programu.
- Poszczególne jednostki kodu, takie jak funkcje i metody, są testowane w celu weryfikacji ich działania.
- Dzięki temu deweloperzy mogą wykrywać i naprawiać błędy już na wczesnym etapie procesu tworzenia oprogramowania, co pozwala zaoszczędzić czas i wysiłek w dłuższej perspektywie.

# Po co przeprowadzać testy jednostkowe?

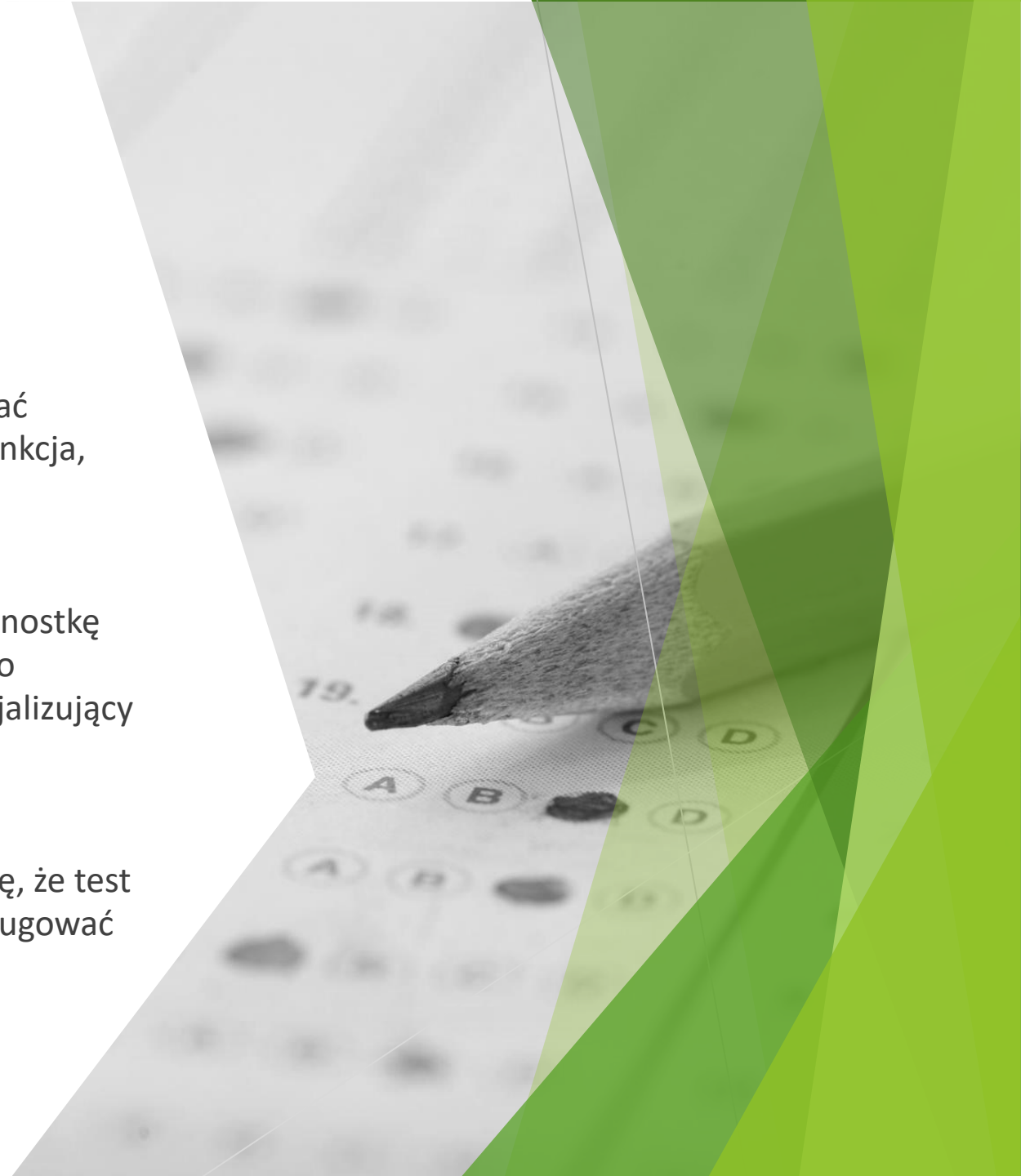
- Testowanie jednostkowe pomaga poprawić jakość kodu poprzez wykrywanie i naprawianie błędów na wczesnym etapie.
- Może również pomóc w debugowaniu, umożliwiając deweloperom łatwiejsze identyfikowanie źródła błędu.
- Testowanie jednostkowe może również ułatwić utrzymanie kodu, dając deweloperom pewność, że istniejące testy jednostkowe wykryją ewentualne błędy wprowadzone przez zmiany.

# Jak pisać testy jednostkowe

- Aby napisać test jednostkowy, należy najpierw zidentyfikować jednostkę kodu, którą chcemy przetestować. Może to być funkcja, metoda lub klasa, na przykład.

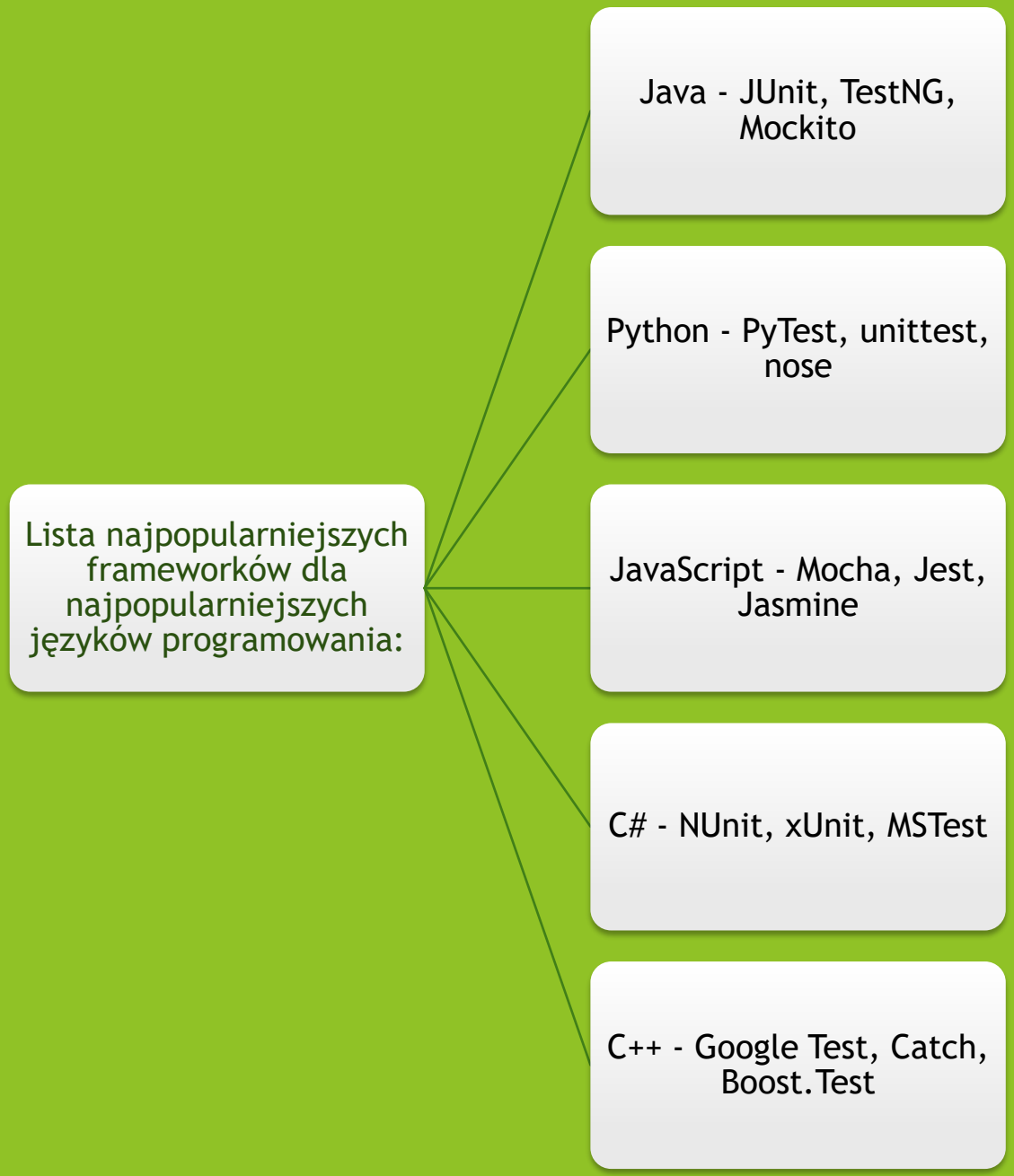
- Następnie należy napisać funkcję testową, która wywoła jednostkę kodu i sprawdzi wynik działania w stosunku do oczekiwanego wyniku. Funkcja testowa powinna również zawierać kod inicjalizujący i kończący test.

- Na końcu należy uruchomić funkcję testową, aby upewnić się, że test zostanie zaliczony. W przypadku niepowodzenia należy zdebugować kod i naprawić błąd.



# Frameworki do testów jednostkowych

Do praktycznie każdego języka programowania możemy znaleźć framework do unit testów



# Przykład testu jednostkowego z wykorzystaniem JUnit

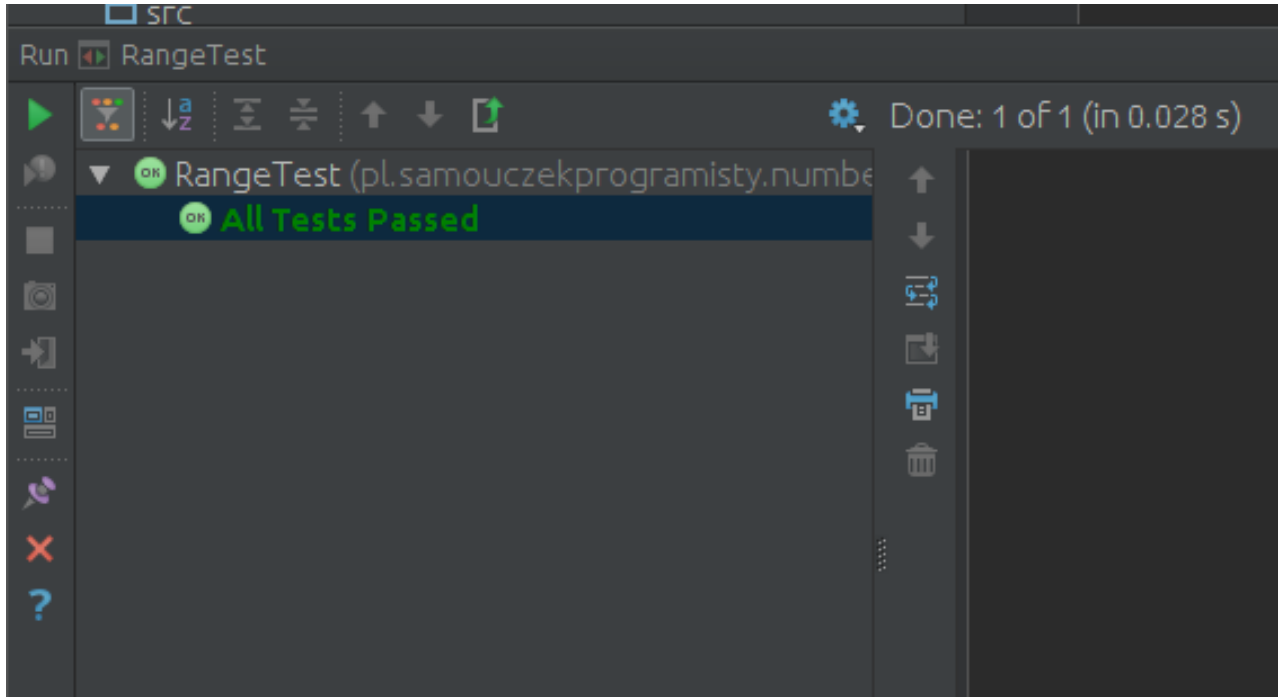
- Przykładowa klasa reprezentująca zakres liczb.
- Posiada jedną metodę, która sprawdza czy liczba przekazana jako argument należy do danego zakresu.

```
public class Range {
    private final long lowerBound;
    private final long upperBound;

    public Range(long lowerBound, long upperBound) {
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }

    public boolean isInRange(long number) {
        return number >= lowerBound && number <= upperBound;
    }
}
```

# Poniżej przykład prostego testu jednostkowego, który sprawdza czy, liczba 15 jest w zakresie liczb od 10 do 20.

```
@Test
public void shouldSayThat15rIsInRange() {
    Range range = new Range(10, 20);
    Assert.assertTrue(range.isInRange(15));
}
```

- Test jednostkowy to metoda sprawdzająca naszą jednostkę, czyli metodę w innej klasie z dodaną adnotacją @Test.
- shouldSayThat15IsInRange jest testem, w którym tworzona jest instancja klasy zakresu i wywoływana jest metoda, która sprawdza czy 15 jest wewnątrz określonego zakresu.
- Wynik jest przekazywany do metody Assert.assertTrue(), która jest tzw. asercją. Jest to metoda dostarczana przez bibliotekę JUnit, które pomagają w testowaniu.
- W naszym przykładzie, jeśli metoda isInRange zwróci false, wówczas asercja assertTrue rzuci wyjątek, który zostanie zinterpretowany jako błąd przez IDE i pokaże błąd działania testowanego kodu.




# Przykład poprawnego testu w **IntelliJ Idea**

# Przygotowanie testów i cykle życia

Zdarza się, że kilka testów jednostkowych wymaga przygotowania. Na przykład trzeba utworzyć instancję, którą będziemy później testowali.

Junit wykorzystuje adnotacje takie jak @Before lub @After, które możemy dodać do metody w klasie z testami.

# Przykład klasy dla testu jednostkowego

Poniżej klasa testu z wykorzystaniem wspomnianych adnotacji:

```
public class RangeTest {
    private Range range;

    @Before
    public void setUp() {
        range = new Range(10, 20);
    }

    @Test
    public void shouldSayThat15rIsInRange() {
        assertTrue(range.isInRange(15));
    }

    @Test
    public void shouldSayThat5IsntInRange() {
        assertFalse(range.isInRange(5));
    }
}
```

# Najważniejsze adnotacje

- @Before - pozwala na wykonanie fragmentów kodu przed testem.
- @After – metoda z tą adnotacją uruchamiana po każdym teście jednostkowym, pozwala na „posprzątanie” po teście,
- @AfterClass – metoda statyczna z tą adnotacją uruchamiana jest raz po uruchomieniu wszystkich testów z danej klasy,
- @BeforeClass – metoda statyczna z tą adnotacją uruchamiana jest raz przed uruchomieniem pierwszego testu z danej klasy.

```java
public class TestLifecycle {
    @Before
    public void setUp() {
        System.out.println("set up");
        System.out.flush();
    }

    @After
    public void tearDown() {
        System.out.println("tear down");
        System.out.flush();
    }

    @BeforeClass
    public static void setUpClass() {
        System.out.println("set up class");
        System.out.flush();
    }

    @AfterClass
    public static void tearDownClass() {
        System.out.println("tear down class");
        System.out.flush();
    }

    @Test
    public void test1() {
        System.out.println("test 1");
        System.out.flush();
    }

    @Test
    public void test2() {
        System.out.println("test 2");
        System.out.flush();
    }
}
```

# Adnotacje w praktyce

Kolejność działania metod na przykładzie outputu konsoli:

- set up class
- set up
- test 1
- tear down
- set up
- test 2
- tear down
- tear down class

# Dobre praktyki przy pisaniu unit testów

- Staraj się pisać testy jednostkowe, które są małe i dotyczą małego wycinka funkcjonalności.
- Nadawaj metodom z testem nazwy, które pomagają zrozumieć co dany test powinien sprawdzić.
- Kolejność testów jednostkowych w klasie nie powinna mieć znaczenia. Nie możemy polegać na tym, że jako pierwszy musi się uruchomić test1 a po nim test2. Testy uruchomione w innej kolejności powinny mieć taki sam efekt.

# Dobre praktyki przy pisaniu unit testów c.d.

- Testuj warunki brzegowe i sytuacje wyjątkowe. Załóżmy, że masz metodę, która przyjmuje tablicę, która musi mieć maksymalnie trzy elementy. Napisz kilka testów:
  - przekazując null zamiast tablicy,
  - przekazując pustą tablicę,
  - przekazując tablicę z trzema elementami,
  - przekazując tablicę z czterema elementami.

- Testowany kod nie powinien być w tym samym miejscu, w którym są testy. Np. kod umieszczamy w katalogu np. src, testy natomiast w katalogu test. Oba katalogi mają odpowiednią strukturę odzwierciedlającą pakiety. Jest to ważne by później przy większych projektach testy nie mieszały się z kodem programu.