Java 17 update

Spis treści

Java 17 update . . . 1
Sealed Classes . . . 1
Pattern Matching for switch (preview) . . . 3
Podsumowanie . . . 4

Java 17 update

Java 17 została wydana we wrześniu 2021 i jest wersją **LTS**. W końcu mamy wydanie, które jest **LTS**. Wróćmy do kwestii ideologicznych, które były poruszane wcześniej. Niektóre organizacje mogą uznać, że stosowanie wersji **LTS** jest bezpieczniejsze ze względu na wsparcie i poprawki, które są oferowane razem z tymi wersjami. Można powiedzieć, że podejście do wersji **non-LTS** i **LTS** jest kwestią indywidualną. Przecież programiści biorą odpowiedzialność za aplikację, którą wytworzyli i która jest faktycznie używana przez klientów na produkcji. Każda organizacja ma inne podejście do potencjalnego ryzyka i to z tego wynika polityka stosowania wersji **non-LTS** i **LTS**.

Dlaczego do tego wracam? Jeżeli dana organizacja ma taką politykę, że stosuje tylko wersje **LTS**, to będzie to oznaczało, że wersja Javy w przykładowym projekcie w takiej organizacji będzie zwiększana bezpośrednio z 11 do 17. Co za tym idzie, to to, że programiści będą mieli dostępne funkcje omówione wcześniej, dopiero gdy dany projekt zostanie przepięty na wersję 17. Java jest **backwards compatible**, czyli funkcjonalności wprowadzone w Java 12, 13, 14, 15 i 16 (oczywiście te, które nie zostały usunięte albo zmienione, bo były np. **preview features**) będą mogły być swobodnie używane w wersji 17, bo dalej są tam dostępne.

Pamiętasz, jak wcześniej zostało wspomniane, że we wrześniu 2021 zaproponowane zostało skrócenie okresu wydań **major** z 3 do 2 lat. Przypominam o tym, bo jesteśmy akurat przy wydaniu Javy, w okolicach którego podjęta została ta decyzja. Oznacza to, że kolejnym wydaniem **LTS** nie będzie Java 23 tylko Java 21.

Przejdźmy do konkretów. Poniżej omówimy niektóre funkcjonalności udostępnione w tym wydaniu. Przy aktualizacji wersji Javy często poprawianych jest o wiele więcej funkcjonalności i dodawanych o wiele więcej klas lub metod niż te, które wymieniamy tutaj. W obrębie tych materiałów poruszamy tylko te kwestie, które są adekwatne do naszego poziomu zaawansowania jako Java developerów.

Sealed Classes

Java 17 wprowadziła koncepcję **Sealed Classes** jako **standard feature**. Koncepcja ta polega na dodaniu możliwości ograniczenia, które klasy mogą dziedziczyć z klasy bazowej.

Wiemy już na tym etapie, na czym polega dziedziczenie i implementacja interfejsów. Patrząc z perspektywy klasy dziedziczącej (subklasy), możemy odczytać informację, z jakiej klasy bazowej (superklasy) dana klasa dziedziczy. Patrząc natomiast na tylko klasę bazową, nie mamy podanej żadnej informacji, które klasy dziedziczą z danej klasy bazowej albo które implementują podany interfejs.

Jednocześnie nie mamy możliwości narzucenia ograniczenia (przed Java 17) które klasy mogą implementować interfejs albo które klasy mogą dziedziczyć z podanej klasy bazowej. Jedyne ograniczenie jakie znamy to narzucenie, że klasa jest `final`. Wtedy żadna klasa nie może z niej dziedziczyć. Czyli nie mamy możliwości rozwiązania pośredniego - określenia, które klasy mogą implementować dany interfejs.

Takie pośrednie rozwiązanie zostało wprowadzone w Java 17 i nazywa się właśnie **sealed classes** (*klasy zapieczętowane*). Przykładowo:

```
public sealed class Animal permits Cat, Dog, Monkey { ①
    private String name;
}
```

① Zostały w tej linijce wprowadzone dwa nowe słówka kluczowe: `sealed` i `permits`. Pierwsze oznacza, że dana klasa ma być traktowana jak klasa zapieczętowana. Drugie pozwala na zdefiniowanie listy dozwolonych typów. Klasa `Animal` określa w tym momencie, które klasy mogą z niej dziedziczyć.

Przy tym zapisie należy zwrócić uwagę na pewną kwestię. Jeżeli stworzysz klasy `Cat`, `Dog` i `Monkey` i nie dopiszesz w definicji tych klas jednej z możliwości: `final`, `sealed` albo `non-sealed` to dostaniesz błąd kompilacji mówiący: *All sealed class subclasses must either be final, sealed or non-sealed*. Oznacza to, że wszystkie klasy, które pojawiły się na liście `permits`, muszą spełniać jeden z wymienionych warunków:

- być oznaczone jako `final`,
- być oznaczone jako `sealed` - będzie to oznaczało, że taka klasa pochodna również jest zapieczętowana, przez co będziemy musieli wskazać kolejne możliwe typy pochodne,
- być oznaczone jako `non-sealed` - w ten sposób określamy, że z tej klasy dalej można dziedziczyć normalnie, bez ograniczeń.

Biorąc pod uwagę powyższe, spójrz na klasy poniżej:

Klasa Cat

```
public final class Cat extends Animal {
}
```

Klasa Dog

```
public sealed class Dog extends Animal permits Husky, Poodle {
}
```

Klasa Husky

```
public final class Husky extends Dog {}
```

Klasa Poodle

```
public final class Poodle extends Dog {}
```

Klasa Monkey

```
public non-sealed class Monkey extends Animal {}
```

Klasa Giraffe

```
public class Giraffe extends Animal {} ①
```

① Ze względu na mechanizm **sealed classes** dostaniemy w tym przypadku błąd kompilacji: *Giraffe is not allowed in the sealed hierarchy*.

No dobrze, ale co nam to wszystko daje? Trzy rzeczy:

- Otrzymujemy dodatkowy mechanizm zabezpieczenia przed dziedziczeniem z danej klasy, który nie jest zero-jedynkowy - tak jak jest `final`. Stosując `final` mogliśmy określić tylko: *można dziedziczyć* albo *nie można dziedziczyć*.
- Przykładowo, twórcy bibliotek albo dla sami twórcy Javy otrzymują dodatkowy mechanizm bezpieczeństwa. Jeżeli ktoś napisze klasę, która jest analogiczna do `String` (w kontekście tego, że `String` jest `final` - nie można z niego dziedziczyć), ale jednak taki twórca będzie miał potrzebę, żeby z takiej klasy dziedziczyły tylko dwie inne klasy, to dzięki **sealed classes** otrzymuje taki mechanizm. Inaczej mówiąc, jeżeli dane API ma nie być rozszerzanie przez nikogo innego niż twórcę danego API - stosując ten mechanizm można dodać takie zabezpieczenie.

Pattern Matching for switch (preview)

Ta funkcjonalność jest trochę analogiczna do `Pattern Matching instanceof`. Została ona wprowadzona jako **preview feature**. Wyobraź sobie, że masz napisany kod jak poniżej:

```
public class Runner {

    private static void who(Animal animal) {
        if (animal instanceof Cat cat) {
            System.out.println("Here is the Cat: " + cat);
        } else if (animal instanceof Dog dog) {
            System.out.println("Here is the Dog: " + dog);
        } else if (animal instanceof Monkey monkey) {
            System.out.println("Here is the Dog: " + monkey);
        } else {
            System.out.println("No idea!");
        }
    }
}
```

Zastosowanie `Pattern Matching for switch` pomogłoby nam skrócić ten zapis do takiej postaci:

```
public class Runner {

    private static void who(Animal animal) {
        switch (animal) {
            case Cat cat -> System.out.println("Here is the Cat: " + cat);
            case Dog Dog -> System.out.println("Here is the Dog: " + Dog);
```

```
            case Monkey monkey -> System.out.println("Here is the Monkey: " + monkey);
            default -> System.out.println("No idea!");
        }
    }
}
```

Stosowanie tego zapisu mogłoby być jednocześnie wykorzystywane z `Sealed Classes`.

Podsumowanie

Przypomnę, że przy aktualizacji wersji Javy często poprawianych jest o wiele więcej funkcjonalności i dodawanych o wiele więcej klas lub metod niż te, które wymieniamy tutaj. Z kolejnymi wersjami wprowadzane są również rozmaite poprawki lub usprawnienia w samym działaniu JVM albo przykładowo Garbage Collectora (w tym przypadku mogą to być, chociażby różne algorytmy, o których działanie oparty jest GC). Zmianom mogą ulegać również kwestie dotyczące zarządzania pamięcią. Oprócz tego kolejne wersje Javy mogą również wprowadzać dodatkowe narzędzia, które programista może wykorzystywać w swojej pracy. Do tego poprawkom mogą podlegać istniejące implementacje metod. W obrębie tych materiałów poruszamy tylko te kwestie, które są adekwatne do naszego poziomu zaawansowania jako Java developerów. Nie poruszamy też zagadnień, co do których twórcy Zajavki uznali, że z naszego punktu widzenia zmiany te nie są aż tak istotne i lepiej poświęcić ten sam czas na skupienie się na dalszych zagadnieniach.

Jeżeli natomiast interesuje Cię, jakie jeszcze zmiany są wprowadzane z każdą wersją — wystarczy, że wpiszesz w Google np. "Java 17 features" i znajdziesz dużo artykułów opisujących wprowadzone zmiany. Możesz również zerknąć na tę stronę JDK 17. Zaznaczam jednak, że wiele funkcjonalności będzie niezrozumiałych. ☺