WYŻSZA SZKOŁA EKONOMII I INNOWACJI W LUBLINIE

WYDZIAŁ TRANSPORTU I INFORMATYKI

KIERUNEK: INFORMATYKA

SPECJALNOŚĆ: INŻYNIERIA OPROGRAMOWANIA

ALEKSANDER WÓJCIK

NR ALBUMU: 22786

Projekt Lombok jako sposób na uelastycznienie kodu Javy
The Project Lombok as a tool to make Java code more elastic

Praca inżynierska napisana
na Wydziale Transportu i Informatyki
pod kierunkiem
dra Andrzeja Bobyka

Lublin 2015

Spis treści

Wstęp

Językiem Java zajmuję się od czterech lat, natomiast w środowisku korporacyjnym od półtora roku. Język Java jest razem z językiem C najpopularniejszym językiem programowania od około 10. lat i jednocześnie najczęstszym wybieranym do nowych ogromnych biznesowych projektów. Mając styczność z podstawami innych języków programowania (C#, Groovy, Scala, JavaScript) zacząłem widzieć wiele obszarów, w których język Java ustępuje nowszym językom i nie rozwija się w równie szybkim tempie i nie prezentuje równie eleganckich rozwiązań podstawowych programów koncepcyjnych. Uznałem, że warto poruszyć ten ciekawy temat w pracy. Wybrałem Projekt Lombok jako przewodni temat mojej pracy z uwagi na znajomość narzędzia, a także możliwość zaprezentowania na stosunkowo prostych przykładach pewne problemy języka Java, jakie Projekt Lombok rozwiązuje.

Celem pracy inżynierskiej jest przedstawienie założeń języka Java, problemów języka z elastycznością w kontekście składni języka i zasad jego działania, krótki opis kierunków zmierzających do udoskonalenia języka, a także dokładne opisanie projektu Lombok jako jednego z tych kierunków.

Podczas pisania pracy korzystałem z materiałów przygotowanych przeze mnie i wygłoszonych w ramach konferencji uczelnianej **KNITS 2015** (Koła Naukowe Informatyki i Transportu — Sympozjum 2015) , zwanej dalej konferencją, dnia 19 października 2015 roku na terenie Wyższej Szkoły Ekonomii i Innowacji.

Praca inżynierska jest autoreferatem składającym się z następujących plików będących integralnymi częściami pracy:

1. `Załącznik nr 1` — referat przygotowany na konferencję w formie publikacji naukowej
2. `Załącznik nr 2` — prezentacja przygotowana na konferencję
3. `program` — katalog ze źródłami do programu ilustrującego działanie projektu Lombok — załączony tylko na płycie

W załącznikach zachowałem oryginalne formatowanie.

Streszczenie

W rozdziale pierwszym przedstawiono w sposób szczegółowy założenia języka Java – wprowadzono wiele terminów związanych z programowaniem obiektowym, które są niezbędne do prawidłowego zrozumienia języka Java oraz zasad działania Projektu Lombok.

W rozdziale drugim przedstawiono kierunki współczesnej informatyki zmierzające do wykorzystania i zachowania zalet języka Java i platformy z nią związanej, przy jednoczesnym udoskonaleniu i unowocześnieniu języka. Opisano dokładnie projekt Lombok jako jeden z tych kierunków i wyjaśniono jego zasady działania odwołując się do procesu kompilacji programu napisanego w języku Java oraz mechanizmu adnotacji.

W rozdziale trzecim przedstawiono właściwe użycie projektu Lombok przy współpracy z programem Intellij IDEA, a także przy użyciu zwykłej linii poleceń. Rozdział składa się z opisu konfiguracji Projektu Lombok pod środowisko Intellij IDEA, szczegółowego opisu wraz z przykładami kilku wybranych spośród 17 adnotacji w wersji stabilnej, jakie Projekt Lombok oferuje oraz instrukcji uruchomienia kodu źródłowego dołączonego do pracy, w którym zilustrowano użycie praktyczne adnotacji Projektu Lombok, a także jego interpretacji.

Abstract

In the first chapter of Thesis, the basics of Java programming language were introduced, including many terms regarding object-oriented language that are vital to properly understand Java and the capabilities of the Project Lombok.

In the second chapter the directions of modern computer science aiming to combine advantages of Java and its environment with enhancing the language itself were introduces. The Project Lombok was described in details as one of them and its way of working in the context of compilation process and annotation mechanism were explained.

In the third chapter the proper use of the Project Lombok with both Intellij IDEA and command line were put in practice. The step by step configuration process in Intellij IDEA, the detailed description of some of 17 annotations in stable version of the Project Lombok were included. Lastly, the chapter contains the instruction and interpretation of source code, in which practical use of those annotation was pictured.

Rozdział 1

Język Java

1.1 Podstawowe definicje[9]

program [7] — sekwencja symboli opisująca realizowanie obliczeń zgodnie z pewnymi regułami zwanymi językiem programowania

podprogram — inne nazwy: procedura, funkcja, metoda, operacja, operator, działanie, predykat - abstrakcja działań podejmowanych przez komputer, ciąg operacji maszynowych, który może zostać wywołany w trakcie trwania programu

klasa — jest abstrakcyjnym typem danych, definiowanym programowo przez podanie operacji dopuszczalnych na nim oraz przez jego reprezentacje i implementacje tych operacji. Definiuje również zestaw danych (pola) – tzn. dopuszczalne wartości, jakie instancje klasy mogą mieć podczas wykonania programu

klasa abstrakcyjna — oznaczona specyfikatorem `abstract`. Nie można tworzyć instancji klasy abstrakcyjnej, ale można ją rozszerzać i tworzyć instancje klas, które dziedziczą z klasy abstrakcyjnej

klasa finalna — oznaczona specyfikatorem `final`. Klasa finalna oznacza, że nie może mieć ona podklas

obiekt — jest instancją klasy, czyli inaczej mówiąc elementem danego typu

dziedziczenie — mechanizm pozwalający na budowanie jednych klas na bazie drugich tak, aby w sposób naturalny jeden typ danych był specjalnym przypadkiem innego typu danych

podklasa — inne nazwy: klasa pochodna, klasa potomna, klasa dziedziczona — klasa wywiedziona przez dziedziczenie z innej klasy

nadklasa — inne nazwy: klasa pierwotna, klasa bazowa — odwrotnie, jeśli A jest klasa pochodna względem B, to B jest klasa bazowa względem A

metoda instancji — każdy podprogram, który może być wywołany tylko na obiektach klasy, i który jest zdefiniowany w ramach tej klasy

metoda statyczna — każdy podprogram, który może być wywołany bez konieczności tworzenia obiektów danej klasy. Każda metoda statyczna jest przypisana do swojej klasy, ale może być wywoływana zarówno bez tworzenia obiektów danej klasy (wówczas metoda jest poprzedzana samą nazwą klasy), jak też możliwe jest wywołanie metody statycznej w taki sam sposób jak metody instancji

metoda główna — program, który może być wywoływany z wiersza poleceń. Metoda ma ściśle określoną sygnaturę i przyjmuje argumenty wiersza poleceń jako argument metody

pole — inne nazwy: własność obiektu, zmienna obiektu - to zmienna zamknięta wewnątrz obiektu. Wartości pól definiują jednoznacznie stan obiektu

polimorfizm — mechanizm pozwalający na dynamiczne — czyli w czasie pracy programu — wywoływanie metod w zależności od typu obiektu

hermetyzacja — inna nazwa: enkapsulacja — oznacza możliwość tworzenia abstrakcyjnych typów danych, czyli takich, w których mamy wyraźnie oddzielony jawny interfejs (czyli zestaw operacji publicznych możliwych do wykonania) od ukrytej implementacji (czyli wewnętrznej realizacji owych operacji)

programowanie zorientowane obiektowo [8] — inna nazwa: programowanie obiektowe (ang. *object-oriented programming*) —- paradygmat programowania, w którym programy definiuje się za pomocą obiektów — elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań

metody dostępu — w programowaniu zorientowanym obiektowo dla każdego atrybutu powinny być utworzone dwie metody - jedna do odczytu (ang. *getter*), druga do ustawienia wartości (ang. *setter*)

IDE (ang. *Integrated Development Environment*) — zintegrowane środowisko programistyczne. Jest to program, który pomaga programiście pisać kod. Dla języka Java są trzy podstawowe platformy IDE: Netbeans, Eclipse oraz (komercyjnie płatny) Intellij IDEA. Posiadają one zaawansowane narzędzia edycji oraz analizy kodu, podświetlania składni oraz błędów, umożliwiają wiele akcji (np. kompilacja, generowanie prostego kodu (np. metod dostępu), uruchomienie aplikacji, nawigacja po klasach i plikach, zmiana nazwy zmiennej lub klasy) za pomocą skrótów klawiaturowych

1.2 Podstawowe reguły języka Java

1.2.1 Typy klas

W Javie mamy dwa podstawowe typy klas. Klasy zagnieżdżone są klasami, których definicja znajduje się wewnątrz innej klasy (ang. *nested class*), wewnątrz metody (ang. *method local class*) lub klasy anonimowe - tzn. klasy, które nie mają jawnej nazwy, a których definicja jest znana kompilatorowi do wystąpienia średnika. Klasy zagnieżdżone mogą być klasami anonimowymi, statystycznymi oraz mogą mieć wszystkie poziomy widoczności. Klasy główne (ang. *top level class*) są klasami, które nie są zagnieżdżone. Mogą mieć tylko poziom widoczności pakietowy lub publiczny. Nie mogą być statyczne, ani anonimowe. Każda klasa niezależnie od przedstawionego typu może być abstrakcyjna lub finalna, ale nigdy nie może być jednocześnie abstrakcyjna i finalna.

1.2.2 Struktura klas

W jednym pliku może być zdefiniowana tylko jedna klasa główna publiczna (tzn. jej definicja jest poprzedzona słowem kluczowym `public`) . Klasa ta musi mieć nazwę odpowiadającą nazwie pliku, w którym jest zdefiniowana. W języku Java klasy są pogrupowane w pakiety w ten sposób, że nazwa pakietu odpowiada nazwie folderu, w którym są przechowywane pliki klas będącym w jednym pakiecie. Program napisany w języku Java[1] składa się z hierarchii klas pogrupowanych w pakiety i posiada co najmniej jedną klasę z metodą główną, tzn. z metodą, której sygnatura ma jedną z poniższych postaci:

Listing 1.1: Możliwe sygnatury metody głównej. Co ważne – w klasie nie mogą znajdować się obie poniższe metody na raz z przyczyn kompilacyjnych

```
1 public static void main(String[] args)
2 public static void main(String... args)
```

1.2.3 Obiektowość

Język Java[5] jest silnie zorientowany obiektowo. W kontekście własności języka o obiekcie można myśleć jako o składowej części programu, który może przyjmować określone stany i ma określone metody, które mogą zmieniać te stany bądź przesyłać dane do innych obiektów. Wyjątkiem od obiektowości są typy prymitywne opisane w 1.3.1

Inna definicja obiektu:

Obiekt[3] *jest elementem modelu pojęciowego obdarzonym odpowiedzialnością za pewien obszar świata rzeczywistego. Sposób realizacji tej odpowiedzialności zależy od samego obiektu.*

1.2.4 Dziedziczenie

W Javie wszystkie obiekty są pochodną obiektu nadrzędnego (klasy `java.lang.Object`), z którego dziedziczą podstawowe metody. Dzięki temu wszystkie klasy mają wspólny podzbiór podstawowych możliwości. W języku Java klasy mogą być wyprowadzone z innych klasy, w ten sposób dziedziną pola i metody tych klas. Z wyjątkiem klasy Object, która nie ma nadklasy, każda klasa ma jedną i tylko jedną bezpośrednią nadklasę (jednokrotne dziedziczenie). W przypadku braku wyraźnie określonej nadklasy, każda klasa jest bezwzględnie podklasą klasy Object. Podklasa dziedziczy wszystkie składowe (pola, metody oraz zagnieżdżone klasy) z jej nadklasy, o ile widoczność pozwala na ich dostęp. Konstruktory nie są składowymi, więc nie są dziedziczone przez podklasy, ale konstruktor może być wywołany z podklasy poprzez użycie słowa kluczowego `super()`. Dokładniej - każda implementacja konstruktora musi zaczynać się od wywołania konstruktora nadklasy `super(args...)` lub innego konstruktora tej samej klasy `this(args...)`. Jeżeli w kodzie takiego wywołania nie będzie, wówczas kompilator sam wywoła konstruktor domyślny nadklasy (`super()`) lub zgłosi błąd kompilacji, jeżeli takiego konstruktora nie znajdzie. Klasa `Object`, zdefiniowania w pakiecie `java.lang`, definiuje i implementuje zachowania wspólne dla wszystkich klas.

Podstawową różnicą między klasami i interfejsami jest fakt, że klasy mają zmienne pola, podczas gdy interfejsy ich nie mają — tzn. interfejs może mieć tylko pola publiczne, statyczne i finalne, czyli każde pole interfejsu jest stałą i nie może zostać zmienione w trakcie działania programu. Dodatkowo, instancją

klasy jest obiekt, podczas gdy nie można mieć instancji interfejsu. Wyjątkiem jest anonimowa implementacja interfejsu, która jest tak naprawdę obiektem. Wówczas anonimowa implementacja dziedziczy bezpośrednio z klasy Object oraz implementuje dany interfejs. Obiekt przechowuje swoje stany w polach, które są zdefiniowane w klasie.

1.2.5 Wielodziedziczenie

Wielodziedziczenie — oznacza sytuację, w której dana klasa dziedziczy z więcej niż jednej klasy bazowej. Język Java nie wspiera wielodziedziczenia w przeciwieństwie do niektórych starszych obiektowych języków np. C++.

Powodem, dla którego język Java nie pozwala na rozszerzanie więcej niż jednej klasy jest uniknięcie wielodziedziczenia stanu, które to może prowadzić do dziedziczenia pól z wielu klas. Utrudnia to czytelność programu - programista musi wykonać dodatkową pracę, by mieć świadomość, do pola której nadklasy obiekt w danym momencie się odwołuje. Drugim powodem uniknięcia wielodziedziczenia w języku Java jest możliwość przedstawienia hierarchii dziedziczenia w formie drzewa, a nie w formie grafu, co jest wizualnie dużo bardziej czytelne dla programisty. Powoduje też zwiększenie czytelności kodu.

W Javie SE 8 wprowadzono szczątkowe wielodziedziczenie implementacji metod. Interfejsy mogą zawierać implementację domyślną metod, dzięki czemu w klasie implementującą wiele interfejsów z różnymi implementacjami domyślnymi jednej metody, mamy możliwość odwołania się do każdej z tych implementacji z pomocą konstukcji:
`InterfaceName.super.method()`.

1.2.6 Klasa abstrakcyjna

Klasa abstrakcyjna jest zadeklarowana za pomocą słowa kluczowego `abstact`. Może, ale nie musi, zawierać abstrakcyjnych metod. Abstrakcyjna klasa nie może mieć instancji, ale może mieć podklasy. Abstrakcyjna metoda jest metodą zadeklarowaną bez implementacji (bez nawiasów klamrowych i zakończona średnikiem). Jeżeli klasa zawiera abstrakcyjne metody, wtedy musi być zadeklarowana jako abstrakcyjna. Każda metoda musi albo być abstrakcyjna i posiadać w sygnaturze słowo kluczowe `abstact`, albo musi mieć implementację. Kiedy abstrakcyjna klasa ma podklasę, to podklasa zazwyczaj zawiera implementacje wszystkich abstrakcyjnych metod klasy rodzica. W przeciwnym wypadku sama musi być zadeklarowana, jako abstrakcyjna. Metody w interfejsie muszą być bezwzględnie abstrakcyjne, więc modyfikator abstrakt nie musi być użyty wraz z interfejsami metod (może, ale nie jest to konieczne). Każda metoda zadeklarowana w interfejsie jest automatycznie publiczna i abstrakcyjna. Każde pole zadeklarowane w interfejsie jest automatycznie publiczne, statyczne i finalne, zatem jego wartość musi być podana w trakcie deklaracji.

W Javie SE 8 rozszerzono definicję interfejsu. Może on zawierać metody domyślne, które nie są abstrakcyjne, a posiadają implementację na poziomie interfejsu. Wówczas są one oznaczone słowem kluczowym `default`.

Metoda wirtualna — w Javie wszystkie metody są domyślnie wirtualne[2], tzn.:

- o możliwych różnych definicjach przy konkretyzacji

- o nieznanym (w chwili kompilowania) dokładnie sposobie działania
- niekoniecznie już istniejące – mogą być abstrakcyjne

Metodami niewirtualnymi[2] są:

- metody statyczne
- metody finalne jawnie — oznaczone specyfikatorem `final`
- metody finalne niejawnie — metody klasy finalnej
- metody prywatne

1.2.7 Polimorfizm

Słownikowa definicja polimorfizmu[2] odnosi się do zasady w której dany byt może mieć różne formy lub etapy. Zasada została zaczerpnięta z biologii i odnosi się do programowania zorientowanego obiektowo i języków takich jak Java. Mając zadeklarowaną dowolną metodę wirtualną kompilator nie wie w czasie kompilacji, której konkretnie implementacji metody będzie używał, ponieważ metoda może być nadpisana przez jej podklasę, a więc może się zmienić jej implementacja. Zatem decyzja o wyborze implementacji musi zostać odłożona do momentu wykonywania programu. Rodzaj takiego polimorfizmu nazywamy polimorfizmem czasu wykonania[4]. Polimorfizm dotyczy tylko metod wirtualnych, ponieważ w przypadku metod niewirtualnych wiadomo, która dokładnie metoda (tzn. podprogram, sekwencja instrukcji) się wykona już w trakcie kompilacji.

1.3 Typy danych w Javie

W Javie mamy dwa typy danych: typy pierwotne i typy obiektowe.

1.3.1 Typy pierwotne

Do typów pierwotnych (inaczej: prostych) należą:

- typ znakowy: `char`
- typ całkowitoliczbowy: `byte, short, int, long`
- typ zmiennoprzecinkowy: `float, double`
- typ logiczny: `boolean`
- specjalny typ: `void` informujący maszynę wirtualną Javy, że dana metoda nie zwraca żadnej wartości i żadnego obiektu, a więc nie musi zawierać w sobie słowa kluczonego `return`

Typy pierwotne nie podlegają dziedziczeniu. Język Java nie pozwala na definiowanie własnych typów pierwotnych.

1.3.2 Typy obiektowe

Do typów obiektowych należą wszystkie typy, które dziedziczą z klasy `java.lang.Object`. Możliwe jest tworzenie nowych typów obiektowych poprzez rozszerzanie już istniejących klas i interfejsów.

1.3.3 Typy obiektowe będące odpowiednikiem typów pierwotnych

W Języka Java mamy specjalne typy, które są odpowiednikami typów pierwotnych opisanych w 1.3.1. Należą do nich:

- typ znakowy: `Character`
- typ całkowitoliczbowy: `Byte, Short, Integer, Long`
- typ zmiennoprzecinkowy: `Float, Double`
- typ logiczny: `Boolean`
- specjalny typ: `Void`

Zalety wprowadzenia w/w typów:

- organizacja kodu — w klasach zdefiniowano stałe i metody statyczne pomagające w operowaniu na typach prymitywnych
- typ `Void` jest pomocny przy generycznej parametryzacji klasy lub metody (informuje wówczas, że dany parametr nie może mieć żadnej wartości ani instancji) oraz przy definiowaniu metadanych w adnotacjach oraz w refleksji. W Javie nie można utworzyć obiektu klasy `Void` bez podnoszenia widoczności konstruktora za pomocą refleksji.
- Autoboxing. Jest to specjalna właściwość kompilatora, która pozwala dynamicznie konwertować typ z pierwotnego do obiektowego odpowiednika i odwrotnie w zależności od tego, jaki statyczny typ do danej operacji jest potrzebny.

  Przykład — w języku Java operatory `+`, `-` są zdefiniowane tylko dla typów pierwotnych, natomiast nie są dozwolone dla typów obiektowych. Dzięki autoboxingowi powyższe wyrażenie będzie poprawne, gdyż kompilator dynamicznie zmieni typ z `Integer` (obiektowy) na `int` (prymitywny):

Listing 1.2: Dodanie typu prymitywnego do listy

```
1 Integer a=5, b=10;
2 a=a+b;
```

Przykład:
Kolekcje w Javie pozwalają na przechowywanie tylko obiektów, a nie typów prymitywnych. Dzięki autoboxingowi poniższy kod będzie poprawny, mimo, że literał `5` oznacza typ statyczny prymitywny `int`:

Listing 1.3: Dodanie typu prymitywnego do listy

```
1 List<Integer> list= new ArrayList<>();
2 list.add(5);
```

Problemy wynikające z wprowadzenia w/w typów: Głównym problemem jest to, że typy prymitywne zawsze mają wartość (wyjątek – typ `void`), natomiast dla typów obiektowych przewidziana jest specjalna wartość `null`, która oznacza — brak wartości. Jest to szczególnie uciążliwe dla typu `Boolean`, ponieważ z logiki dwuwartościowej (zmienna logiczna ma dwie wartości — prawda/ fałsz — `true/false`) przechodzimy do logiki trójwartościowej, (zmienna logiczna ma trzy wartości — prawda/ fałsz/ brak wartości — `true/false/null`). Przeszkadza to szczególnie przy rzutowaniu — poniższe kody zakończą się wyrzuceniem wyjątku `NullPointerException`:

Listing 1.4: Problemy przy wartości `null` dla typu logicznego

```
1 //1
2 Boolean b=null;
3 if(b){
4 ///...
5 }
6
7 //2
8 Boolean b=null;
9 boolean b2=b;
```

Wprowadzenie wartości logicznej `null` ma jednak uzasadnienie przy mapowaniu wartości tabel np. w systemie baz danych Oracle, gdzie w komórce tabeli typu logicznego może być wartość `true`, `false`, ale może także nie być wartości — wówczas naturalnym odpowiednikiem w języku Java staje się wartość `null`. Jednak problemy wynikające z wartości `null` wydają się większe niż ta jedna zaleta.

1.4 Hermetyzacja

Hermetyzacja — oznacza zastrzeżenie widoczności składowych (metod i pól) w celach poprawy jakości kodu.
Dokładniejsza definicja:
Hermetyzacja[6] *to sposób odizolowania od otoczenia wybranych danych i funkcji (operujących na tych danych) zgromadzonych w jednej strukturze. Widoczne są tylko niezbędne fragmenty programu, natomiast zmienne i funkcje pomocnicze są ukryte i niedostępne z zewnątrz. Dzięki takiemu połączeniu programista uwalnia się od pamiętania o wszystkich szczegółach implementacyjnych, co zapewnia zmniejszenie liczby błędów oraz prostszą strukturę programu końcowego.*

Znalety hermetyzacji:

- mniejsza podatność na błędy
- poprawienie przejrzystości kodu dzięki ograniczeniu widoczności klasy na zewnątrz pakietu tylko do metod publicznych i chronionych
- danie programiście możliwości większej ekspresji kodu poprzez wyrażenie do czego dany element ma być wykorzystywany. Dzięki temu informacje o widoczności są zawarte w kodzie i nie muszą być wyrażone w komentarzach

1.4.1 Specyfikatory widoczności

W języku Java mamy następujące specyfikatory widoczności. Możemy je używać do pól oraz metod klas, enumów (typów wyliczeniowych) oraz interfejsów, choć w przypadku interfejsów nie są one brane

pod uwagę przez kompilator, ponieważ są zastępowane domyślnymi spefikatorami. Możemy je też używać do oznaczania widoczności samych klas, enumów oraz interfejsów. Nie możemy natomiast precyzować widoczności: zmiennych lokalnych, klas o zasięgu metody.

- `public` — publiczny, widziany dla klas ze wszystkich pakietów
- `protected` — chroniony, widziany z klas z danego pakietu oraz klas dziedziczonych
- `package` (bez specyfikatora) — domyślny – pakietowy, widziany z klas z danego pakietu
- `private` — prywatny, widziany tylko dla danej klasy oraz klas w niej zagnieżdżonych

1.5 Hermetyzacja na przykładzie języka Java

Hermetyzacja w języku Java oznacza, że pola obiektu powinny mieć dostęp prywatny, natomiast dostęp do nich powinien być możliwy poprzez publiczne metody. Stałe natomiast powinny być wyrażone jako publiczne (ewentualnie prywatne), statyczne i finalne pola. Dzięki hermetyzacji możemy wyodrębnić, które pola będą modyfikowalne, a które uznajemy jako pomocnicze dla naszej klasy i nie powinny one posiadać metod dostępu. Ponadto możemy w metodach dostępowych zawrzeć dodatkową logikę np. zliczanie ile razy dane pole było odczytane.

Wadą takiego rozwiązania jest powstanie wielu metod publicznych (po dwie dla każdego pola – jedna do odczytu, druga do zmiany wartości pola) tylko do obsługi podstawowej funkcjonalności — zapisu i odczytu. Przez to inne metody, których funkcjonalność jest mniej banalna, nie są w żaden sposób przez język wyróżnione, a powinny.
Inne języki programowania jak np. C# bardziej elegancko podchodzą do zagadnienia odczytu i zmiany wartości pola.

Rozdział 2

Uelastycznienie kodu Java jako cel powstania Projektu Lombok

Język Java z powodu konieczności zachowania wstecznej kompatybilności rozwija się powoli. Dla przykładu — dodanie w wersji 1.4 słowa kluczowego `assert` powoduje, że metody i klasy o nazwie "assert" nie będą się poprawnie kompilowały przy kompilatorze w wersji większej lub równej 1.4. Jednak platforma uruchomieniowa JVM + JRE kodu napisanego w Javie jest atrakcyjna dla projektów z powodu znaczącej liczby gotowych bibliotek oraz zalet JVM. Z tego powodu powstały propozycje jak wykorzystać potencjał platformy związanej z językiem Java jednocześnie zyskując większą elastyczność na poziomie składni i paradygmatu niż język Java oferuje.

2.1 Języki JVM

Pierwszym podejściem jest wprowadzenie nowego języka, który jest rozszerzeniem języka Java (dzięki czemu możemy korzystać z bibliotek języka Java), oraz który kompiluje się do java bajtkodu. Takimi językami są Groovy, Scala, Gosu, AspectJ. Istotną wadą takiego rozwiązania jest konieczność wprowadzenia nowego narzędzia — kompilatora dla danego języka. Oraz jego przetestowanie — domyślny kompilator `javac` został już przez wiele lat przetestowany i jest na pewno lepiej poznany.

2.2 Xtend

Xtend jest drugą opcją oferującą rozszerzenie składni języka Java. Jest to nadzbiór języka. Kod napisany w języku Xtend jest następnie kompilowany do kodu Java. Oznacza to, że przed kompilacją do java bajtkodu możemy wprowadzić jakieś zmiany do kodu. Język jest bardzo prosty do nauczenia się, ale posiada słabe wsparcie IDE. Zaletą takiego rozwiązania jest brak konieczności wprowadzenia dodatkowego kompilatora. Oczywiście dodatkowy kompilator tutaj występuje (z kodu Xtend do kodu Java), jednak jest to znacznie mniej złożone narzędzie.

2.3 Projekt Lombok

Projekt Lombok jest trzecią propozycją. Jest to niewielka (1.3 MB) biblioteka napisana w "niskopoziomowej" Javie dostępna pod licencją MIT[12]. W skrócie polega on na generowaniu bajtkodu java za

pomocą instrukcji zapisanych w adnotacjach. Istotną zaletą jest brak konieczności nauki narzędzia — wystarczą do tego przykłady z tutoriali lub strony projektu[11]. **Nie jest to nowy język programowania**. Zaletą jest też to, że nie wprowadzamy nowego kompilatora języka, a jedynie dokonujemy zmian w działaniu kompilatora podstawowego `javac`.

Zalety:

- dobra dokumentacja, szybkość w konfiguracji i nauce
- poprawa czytelności kodu i redukcja zbędnego kodu
- dobra integracja ze wszystkimi IDE
- możliwość globalnej konfiguracji
- możliwość definiowania własnych adnotacji
- polecenie `delombok` powoduje usunięcie wszystkich adnotacji projektu lombok i zastąpienie ich właściwym kodem Javy
- mniejsze prawdopodobieństwo popełnienia trudnych do wykrycia błędów. Znany jest przypadek, gdzie po prowadzeniu Projektu Lombok kilka błędów zniknęło z systemu:
  And to add to this: switching to Lombok has actually allowed to solve some intricate bugs in the past due to mismatched equals / hashCode implementations, and cases in which I forgot to update now-generated methods when a field was added[13]

Wady:

- naruszenie kontraktu adnotacji — adnotacje przestają być tylko metadanymi, a zaczynają być instrukcjami dla procesora do generowania kodu
- w przypadku tworzenia dodatkowych adnotacji potrzebujemy dodatkowej pracy by zapewnić współpracę z niektórymi popularnymi IDE (Eclipse, Intellij IDEA)
- brak wysokopoziomowego API do rozszerzenia projektu – tzn. do pisania własnych adnotacji, które zmieniają wygenerowany java bajtkod
- generowany kod jest "niewidoczny" — tzn. istnieje dopiero po kompilacji. Istnieje ryzyko, że niektóre narzędzia będą czegoś nie rozpoznawały i praca z nimi będzie utrudniona. Ponadto trudne jest zmienianie wygenerowanego kodu poza funkcjonalnościami udostępnionymi w konfiguracji projektu

2.4 Zasady działania Projektu Lombok

Szczegółowy opis działania Projektu Lombok został opisany w moim referacie na konferencje KNITS 2015 (patrz Załącznik A).
W skrócie opisałem tam:

1. szczegółowy opis kompilacji kodu źródłowego programu napisanego w języka Java do bajkodu z podziałem na fazy kompilacji
2. zasady działania JVM

3. pojęcie adnotacji w języka Java
4. działanie procesora adnotacji
5. wprowadzenie pojęcia drzewa AST — Abstrakcyjne Drzewo Składniowe (ang. *Abstract Syntax Tree*, AST)
6. opis manipulowania drzewem AST będącym podstawowym narzędziem działania Projektu Lombok
7. krótkie przedstawienie API projektu

Rozdział 3

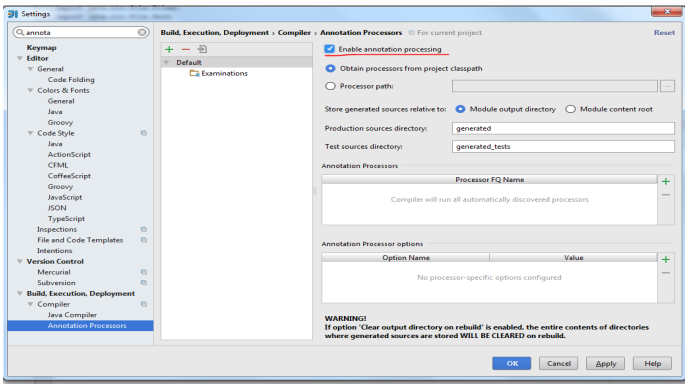
Użycie Projektu Lombok

Opiszę w kilku krokach jak zainstalować Projekt Lombok na środowisku Intellij IDEA. Środowisko jest dostępnie bezpłatnie w wersji podstawowej (Community), natomiast płatne w wersji rozszerzonej o wsparcie framework'ów do budowy aplikacji komercyjnych (Ultimate).

3.1 Konfiguracja

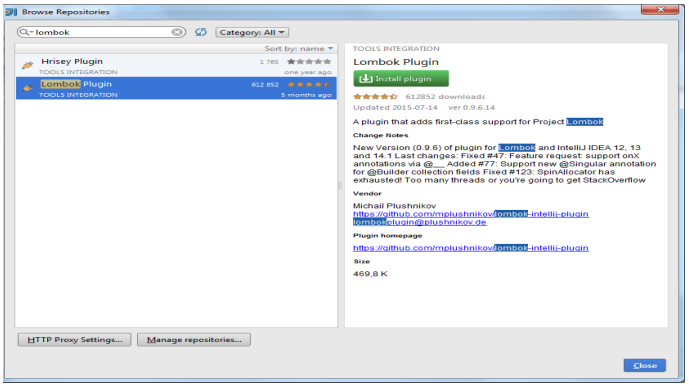
1. Pobieramy i dodajemy do classpath'u plik lombok.jar ze strony:
   https://projectlombok.org/download.html

2. Umożliwiamy przetwarzanie procesora adnotacji:



Rysunek 3.1: Zaznaczamy opcję "Enable annotation processing" podkreśloną czerwoną linią

3. Instalujemy wtyczkę Lombok Plugin ze strony lub poprzez IDE:
   https://plugins.jetbrains.com/plugin/6317



Rysunek 3.2: Klikamy na przycisk "Install" aby zainstalować wtyczkę

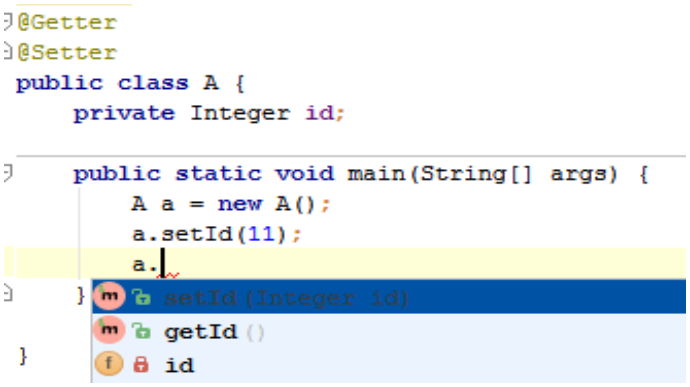
Sprawdzenie poprawności konfiguracji

Po skonfigurowaniu IDE powinno rozpoznawać, że adnotacje projektu Lombok generują nam kod. Poniżej zdjęcia – jak nam środowisko podpowiada przed i po konfiguracji:

```
@Getter
@Setter
public class A {
    private Integer id;

    public static void main(String[] args) {
        A a = new A();
        a.setId(11);
        a.
    }
}
```

id
clone()
equals(Object)

Rysunek 3.3: Przed konfiguracją. Zauważmy, że środowisko nie rozpoznaje, że adnotacja `@Getter` wygenerowała nam metodę `getId()`



Rysunek 3.4: Po konfiguracji. Środowisko rozpoznało, że adnotacja `@Getter` wygenerowała nam metodę `getId()`

Uruchomienie programu Po konfiguracji program należy uruchomić jak standardowy program Java

3.2 Polecenie delombok[10]

Jeżeli chcielibyśmy z jakiegoś powodu zrezygnować z adnotacji projektu Lombok, z pomocą przychodzi nam polecenie delombok. Powoduje ono usunięcie adnotacji projektu Lombok i zastąpienie ich kodem Java równoważnym funkcjonalnością adnotacjom. Należy wówczas wydać polecenie:
`java -jar lombok.jar delombok src -d src-delomboked`

3.3 Wzmianka o IDE

Ponieważ IDE korzystają ze "swoich" drzew AST, natomiast projekt Lombok modyfikuje w trakcie uruchomienia drzewo AST wygenerowane przez kompilator `javac`, IDE radzą sobie z problemem jak poniżej:

1. Netbeans — korzysta ze standardowych drzew generowanych przez kompilator `javac`. Dlatego nie wymaga zewnętrznych narzędzi, a każda własna adnotacja oparta o projekt Lombok jest automatycznie rozpoznawana przez Netbeans

2. Eclipse — korzysta ze swojego drzewa AST. Dlatego każda własna adnotacja oparta o projekt Lombok musi zostać uzupełniona o wsparcia dla środowiska Eclipse. Na szczęście wszystkie adnotacje projektu Lombok są dostosowane do tego środowiska

3. Intellij IDEA — tak jak Eclipse korzysta ze swojego drzewa AST. Aby korzystać z Projektu Lombok należy zainstalować wtyczkę Lombok Plugin

3.4 Możliwości Projektu Lombok

Przykłady użyć adnotacji Projektu Lombok można znaleźć na stronie projektu[11]. W pracy opiszę kilka z nich. Każdy przykład będzie zawierał: opis, kod z użyciem adnotacji Lombok oraz kod w czystym

języku Java, który jest równoważny kodowi z użyciem adnotacji.

Adnotacje Projektu Lombok można w pewnym stopniu konfigurować — np. wykluczać pewne elementy z generowanych metod, zmieniać widoczność wygenerowanych metod oraz regulować co ma być logowane jako ostrzeżenie w specyficznych sytuacjach. Można również zapisać domyślną konfigurację wszystkich adnotacji dla danego projektu w pliku `lombok.config`.

Nie jest jednolite zachowanie Projektu Lombok w sytuacji, kiedy Projekt Lombok chce wygenerować składową (metodę lub konstruktor), podczas gdy składowa o takiej samej sygnaturze już istnieje — została wcześniej jawnie napisana przez programistę. Podam dwa przykłady:

- W przypadku próby wygenerowania metody dostępu o takiej samej sygnaturze jak ta napisana przez programistę, **Projekt Lombok zrezygnuje z generowania danej metody** i zostawi tę napisaną przez programistę
- W przypadku próby wygenerowania konstuktora o samej sygnaturze jak ta napisana przez programistę, **Projekt Lombok zgłosi błąd kompilacji** i tym samym przerwie kompilację

W większości przypadków spodziewane jest pierwsze zachowanie.

Działanie programów napisanych z użyciem adnotacji Projektu Lombok może być różne dla różnych wersji Projektu Lombok oraz dla różnych wersji języka Java.

3.4.1 @Getter @Setter

Generowanie metod dostępu (ang. *getter, setter*) jest najprostszym użyciem adnotacji projektu Lombok. Przynosi też najwięcej korzyści — klasy zamiast posiadać wiele metod dostępu + metod biznesowych, posiadają tylko metody biznesowe. Zatem użycie adnotacji powoduje zmniejszenie liczby metod widocznych w kodzie klasy o

$$2 * n$$

gdzie n —liczba pól z dostępem przez publiczne metody. Często zdarza się, że wszystkie pola mają publiczne metody dostępu.

Warto zaznaczyć, że:

- Dla pól typu `boolean` adnotacja generuje metody `isXxx()/setXxx(...)` zamiast metod `getXxx()/setXxx(...)` . Niestety standard Java nie wykłada jednoznacznie, która konwencja nazewnictwa metod dostępowych jest prawidłowa, a nawet w niektórych przypadkach proponuje niestandardowe konwencje np:
  ```
  boolean hasLicense();
  boolean canEvaluate();
  ```

  Co ciekawe środowisko Intellij IDEA przy generowaniu metod odczytu dla typu `boolean` zachowuje konwencje `isXxx()`, natomiast dla typu `Boolean` konwencje `getXxx()`. Wynika to prawdopodobnie stąd, że typ `Boolean` może przyjmować brak wartości (`null`).
- Domyślnie generowane są metody dostępu z widocznością publiczną, jednak możliwe jest wygenerowanie metod dostępu z każdą inną widocznością za pomocą jednego z poniższych parametrów

adnotacji:
`AccessLevel.PUBLIC, AccessLevel.PROTECTED, AccessLevel.PACKAGE, AccessLevel.PRIVATE`

- Użycie adnotacji na polu powoduje wygenerowanie odpowiednich metod dostępu dla danego pola. Użycie adnotacji na klasie powoduje wygenerowanie odpowiednich metod dostępu dla wszystkich niestatycznych pól klasy.

Przykład użycia — z adnotacjami

Listing 3.1: Poniższe adnotacje wygenerują metody dostępu dla wszystkich niestatycznych pól klasy

```
1  @Getter @Setter
2  public class Point {
3      private int x;
4      private int y;
5
6      public void move(int vectorX, int vectorY) {
7          this.x+=vectorX;
8          this.y+=vectorY;
9      }
10 }
```

Listing 3.2: Możliwe jest też wybranie, dla których pól metody dostępu mają zostać wygenerowane

```
1  public class Point {
2      @Getter @Setter
3      private int x;
4      @Getter @Setter
5      private int y;
6
7      public void move(int vectorX, int vectorY) {
8          this.x+=vectorX;
9          this.y+=vectorY;
10     }
11 }
```

Przykład użycia — bez adnotacji

Listing 3.3: Kod Java równoważny kodowi z poprzedniego listingu. Zwróćmy uwagę, że sygnatura metody `move(...)` nie jest w żaden sposób wyróżniona, a jako jedyna nie jest zwykłą metodą dostępu. To jest przypadek małej czytelności kodu, o którym wcześniej wspominałem

```
1  public class Point {
2      private int x;
3      private int y;
4
5      public void move(int vectorX, int vectorY) {
6          this.x+=vectorX;
```

```
7           this.y+=vectorY;
8       }
9       public int getX() {
10          return x;
11      }
12      public void setX(int x) {
13          this.x = x;
14      }
15      public int getY() {
16          return y;
17      }
18      public void setY(int y) {
19          this.y = y;
20      }
21  }
```

3.4.2 @EqualsAndHashCode

Metody `equals(...)` oraz `hashCode()` są metodami klasy`java.lang.Object`, które każda klasa powinna nadpisywać, ale nie jest to jej obowiązkiem. Powinno się albo nadpisywać obie metody albo żadną. Metody mają poniższą sygnaturę:

Listing 3.4: Sygnatura metod: `equals(...)` oraz `hashCode()`

```
1  boolean equals(Object o)
2  int hashCode()
```

Metoda `equals(...)` jest używana intensywnie przy testowaniu za pomocą metody `Assert.equals(...)` odpowiedniej dla wybranego framerowka (JUnit, TestNG).
Metody `equals(...)` oraz `hashCode()` są intensywnie używane w kolekcjach dla implementacji podstawowych interfejsów — `java.util.Set` oraz `java.util.Map`. Wyjątkiem od tych reguł jest klasa `java.util.WeakHashMap`, która elementy porównuje za pomocą porównania ich referencji (`==`), a nie metody `equals(...)`.

Zarówno zbiory (implementacje interfejsu `java.util.Set`) jak i mapy (implementacje interfejsu `java.util.Map`) **nie pozwalają na duplikacje elementów** (tzn. w zbiorach wszystkie elementy są unikalne, natomiast w mapach zbiór kluczy ma unikalne elementy). Nie pozwalają na duplikacje — w praktyce wygląda następująco: nie istnieją w zbiorze żadne dwa elementy, takie, że poniższy algorytm stwierdzi, że te dwa elementy są równe.

Algorytm określania czy dane elementy są równe wygląda następująco:

1. Najpierw są porównywane hashCode'y obu elementów. Jeżeli są różne — wówczas algorytm rozstrzyga, że elementy są różne. Jeżeli są równe — wówczas przechodzi do punktu drugiego
2. Następnie porównuje elementy metodą equals(...), która ostatecznie rozstrzyga, czy elementy są równe, czy różne

Kontrakt pomiędzy tymi metodami jest następujący: jeżeli dwa obiekty `o1, o1` są równe (poprzez porównanie metodą `equals(...)`), to ich hashcode'y muszą być równe. Logicznie warunek wygląda następująco:

Listing 3.5: Kontrakt pomiędzy metodami `equals(...)` oraz `hashCode()`

```
1 o1.equals(o2) == true => o1.hashCode() == o2.hashCode()
```

Problem jest tutaj prosty: funkcja haszująca `hashCode()` liczy skrót stanu obiektu na podstawie jego pól. Załóżmy, że w chwili obecnej klasa A ma n pól: $x_1, x_2, ...x_n$ i funckja `hashCode()` jest postaci:

$$f(x_1, x_2, \dots x_n)$$

Załóżmy, że dodajemy do klasy kolejne pole — x_{n+1}. Mamy teraz trzy potencjalne zachowania programisty:

1. Programista zaktualizuje zarówno metodę `equals(...)` jak i `hashCode()`
2. Programista nie zaktualizuje żadnej z metod: `equals(...)`, `hashCode()`
3. Programista zaktualizuje tylko jedną z metod: `equals(...)`, `hashCode()`

Wówczas sytuacja nr 3 będzie dla pewnych obiektów **łamać kontrakt** metod `equals(...)` oraz `hashCode()`. Może powodować to szereg **trudnych do wykrycia, a więc kosztownych czasowo** błędów (czyli w warunkach korporacyjnych — drogich) związanych z obsługą kolekcji jak np. znikanie elementów ze zbioru, pojawianie się elementów w mapie, których nie powinno być lub też nadmiernie niewydajne operacje na kolekcjach.

Przykładowo — załóżmy, że mamy klasę abstrakcyjną `Xxx` z jednym polem np. `id` oraz n różnych klas dziedziczących po `Xxx`, przy czym metody `equals(...)` oraz `hashCode()` nadpisane są tylko w klasie `Xxx`. Wówczas, jeżeli mamy $k > n$ dowolnych instancji dowolnych klas dziedziczonych po `Xxx` mających pole `id` ustawione na jednakową wartość (np. 6 lub brak wartości — `null`), oraz jeżeli do pustego zbioru dodamy te k obiektów, to liczba elementów w zbiorze będzie równa 1. Jest to prawidłowe działania programu oraz subtelny i potencjalnie trudny do wykrycia błąd, którego przyczyną jest specyficzne nadpisywanie w/w metod.

Pomocą w tej sytuacji jest adnotacja `@EqualsAndHashCode`. Dzięki niej powyższe metody są **generowane automatycznie w trakcie kompilacji programu na podstawie pól niestatycznych danej klasy**. Dzięki temu kłopotliwy przypadek nr 3. z powyższego programu nie ma prawa się nigdy wydarzyć, co oszczędza ryzyko kosztownych błędów jak i zwalnia programistę z pamiętania o trywialnej rzeczy, o której zapomnienie może generować błędy.

Przykład użycia — z adnotacjami

```
1  @EqualsAndHashCode
2  public class Point {
3      private int x;
4      private int y;
5
6      public void move(int vectorX, int vectorY) {
```

```
7           this.x+=vectorX;
8           this.y+=vectorY;
9       }
10  }
```

Przykład użycia — bez adnotacji

```
1   public class Point {
2       private int x;
3       private int y;
4
5       public void move(int vectorX, int vectorY) {
6           this.x+=vectorX;
7           this.y+=vectorY;
8       }
9
10      @Override
11      public boolean equals(Object o) {
12          if (this == o) return true;
13          if (!(o instanceof Point)) return false;
14
15          Point point = (Point) o;
16
17          if (x != point.x) return false;
18          return y == point.y;
19
20      }
21
22      @Override
23      public int hashCode() {
24          int result = x;
25          result = 31 * result + y;
26          return result;
27      }
28  }
```

3.4.3 @Data

Adnotacja `@Data` powoduje wygenerowanie następujących składowych klasy:

- metody — `String toString()`
- odpowiednich metod dostępu dla niestatycznych pól
- metod `equals(...)` oraz `hashCode()`
- konstruktora, który inicjuje wszystkie finalne pole klasy, których wartość nie została ustawiona

Dobrą praktyką jest, aby wszystkie klasy w projekcie nadpisywały w/w metody. Zatem — gdy mamy do naszego projektu podpiętą bibliotekę Lombok, wszystkie klasy powinny posiadać adnotację `@Data`, natomiast gdy jej nie mamy, wtedy w/w metody powinny być wygenerowane przez IDE. Oczywiście może się zdarzyć sytuacja, gdy szczególne zalecenia architektoniczne są inne.

W skrócie — bez metod `equals(...)` oraz `hashCode()` kolekcje nie będą działały prawidłowo oraz przy testowaniu nie będziemy mogli prawidłowo korzystać z metody `Assert.equals(...)`, natomiast bez nadpisanej metody `toString()` logowanie jest mało czytelne, a debugowanie utrudnione.

Przykład użycia — z adnotacjami

```
@Data
public class Point {
    private int x;
    private int y;

    public void move(int vectorX, int vectorY) {
        this.x+=vectorX;
        this.y+=vectorY;
    }
}
```

Przykład użycia — bez adnotacji

```
public class Point {
    private int x;
    private int y;

    public void move(int vectorX, int vectorY) {
        this.x+=vectorX;
        this.y+=vectorY;
    }
    public Point() {
    }

    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
```

```
23      }
24
25      @Override
26      public boolean equals(Object o) {
27          if (this == o) return true;
28          if (!(o instanceof Point)) return false;
29
30          Point point = (Point) o;
31
32          if (x != point.x) return false;
33          return y == point.y;
34
35      }
36
37      @Override
38      public int hashCode() {
39          int result = x;
40          result = 31 * result + y;
41          return result;
42      }
43
44      @Override
45      public String toString() {
46          return "Point{x=" + getX() + ", y=" + getY() + '}';
47      }
48
49  }
```

3.4.4 Inne adnotacje

Projekt Lombok oferuje również inne, bardziej zaawansowane adnotacje: w sumie 17 adnotacji w wersji stabilnej i 7 adnotacji w wersji testowej. W pracy inżynierskiej skupiłem się na omówieniu najistotniejszych i najprostszych do wytłumaczenia i opisania na przykładzie. Do wytłumaczenia niektórych pozostałych adnotacji niezbędne jest znacznie obszerniejsze wprowadzenie w język Java jak i we wzorce projektowe (np. adnotacja `@Builder`). Wymienię niektóre z pozostałych adnotacji w wersji stabilnej:

- `@Cleanup` — automatycznie zamyka strumienie. Przydatna w wersji Javy 1.6, natomiast w wersji 1.7 problem został rozwiązany dzięki konstrukcji `try-with-resources`
- `@NonNull` — automatycznie wyrzuca wyjątek gdy jako argument funkcja dostanie brak wartości – tzn. literał `null`
- `@ToString` — generuje metodę `String toString()`
- `@SneakyThrows` — umożliwia rzucanie tzn. `checked exceptions`(tzn. pochodnych klas `java.lang.Exception`, ale nie będącymi pochodnymi klas `java.lang.RuntimeException`) bez obowiązku przechwytywania wyjątku. Wzorowane na języku C#, gdzie nie ma `checked exceptions`
- `@Value` — tworzenie niezmiennych klas za pomocą adnotacji. Niezmienne klasy oznaczają takie, że ich stan jest określany w chwili tworzenia i nie jest później możliwa zmiana wartości pól danej

klasy. Przykładami takich klas są obiektowe odpowiedniki typów prymitywnych (1.3.3), klasy: `java.land.String`, `java.io.File`, `java.util.Locale` lub klasy reprezentujące czas w wersji Javy 1.8

- `@Log` — tworzy statyczną instancję logger'a do logowania
- `@Builder` — deleguje proces tworzenia obiektu klasy `Xxx` do innej klasy – `XxxBulder` – zgodnie ze wzorcem budowniczy
- `@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor` — generuje odpowiednie konstruktory
- `val` — jako jedyny element projektu nie jest to adnotacja. Działa tak samo jak słowo kluczowe `val` znane z języków: Scala czy C#

3.5 Przykład programu

Założenia:

- wersja Javy: 1.7.0_79
- wersja projektu lombok: 1.16.6

Projekt powinien dać się uruchomić dla innych wersji i wynik działania konsoli powinien być identyczny, jednakże nie mogę tego gwarantować.

3.5.1 Opis plików na płycie

Na płycie znajdują się następujące katalogi w katalogu `program`:

- program/lib — zawiera bibliotekę lombok w wersji 1.16.6
- program/src — zawiera klasy źródłowe
- program/project — zawiera gotowy do zaimportowania projekt programu Intellij IDEA
- program/run — zawiera spakowany plik uruchomieniowy `aleksander.jar` oraz skrypt `run.bat` uruchamiający go pod systemem Windows 7

3.5.2 Wprowadzenie

Niezbędna do zrozumienia programu będzie dodatkowa teoria:

- domyślna (tzn. z klasy `java.lang.Object`) implementacja metody `String toString()` ma postać:

Listing 3.6: implementacja metody `String toString()`

```
1  public String toString() {
2      return this.getClass().getName() + "@" + Integer.toHexString(this.hashCode());
3  }
```

Oznacza to tyle, że jeżeli w naszej klasie nie nadpiszemy domyślnej implementacji metody `toString()` z klasy `java.lang.Object`, to wywołana zostanie powyższa implementacja przy wywołaniu metody `toString()` na instancji naszej klasy.
Domyślna implementacja zwraca napis będący połączeniem pełnej nazwy klasy naszego obiektu (pełnej — tzn. wraz z nazwą pakietu), separatora `@`, oraz heksadecymalnej reprezentacji hashcode'u naszego obiektu.

- domyślnie zmiennych lokalnych nie jest żadna wartość inicjalizowana, natomiast dla pól klasy inicjowane są następujące wartości:

  1. dla typów obiektowych — zawsze inicjowana jest wartość `null` (ściśle: brak wartości, referencja pusta)
  2. dla typów prymitywnych:
     (a) `byte, short, int` — `0`
     (b) `long` — `0L`
     (c) `float` — `0.0f`
     (d) `double` — `0.0`
     (e) `char` — `'\u0000'`
     (f) `boolean` — `false`

- pole statyczne `out` klasy `java.lang.System` jest klasy `java.io.PrintStream`, która to klasa zawiera metody:
  `public void print(String s)`\\drukuje zmienną s
  `public void println(String x)`\\drukuje zmienną x oraz łamie bieżącą linię
  Ponadto pole to służy jako standardowy strumień wyjścia. W przypadku naszej małej aplikacji będzie to zwykłe wyjście na konsolę

- w przypadku metod formatujących, a taką jest niewątpliwie wspomniana wyżej metoda `printf(...)`, bezpiecznie jest używać specjalnego znacznika formatowania `"%n"`, który to wstawia znak łamiący linię prawidłowy dla systemu operacyjnego, pod którym aplikacja jest odpalana (czyli zazwyczaj — dla systemu Windows: '\r\n', dla systemu Unix/Linux: '\n'). Można też skorzystać z metody: `String java.lang.System.lineSeparator()`

3.5.3 Struktura programu

Program składa się z:

1. biblioteki Lombok — pliku `lombok.jar`
2. pakietu `aleksander.wojcik.wsei`, a w nim plików, których zawartość zostanie przedstawiona na kolejnych stronach:
   - `Point.java`
   - `PointData.java`
   - `Program.java`

Point.java

Listing 3.7: Jest to pełna klasa z poprzednich przykładów. Zarówno metody dostępu jak i metoda `move(...)` są nadmiarowe — w sensie, że nie będę z nich korzystał przy uruchomieniu programu. Warto pamiętać, że w przypadku utworzenia instacji klasy `Point` pola `x` oraz `y` będą automatycznie inicjowane wartością 0

```
1  package aleksander.wojcik.wsei;
2
3  /**
4   * Created by Aleksander Wojcik WSEI
5   */
6  public class Point {
7      private int x;
8      private int y;
9
10     public void move(int vectorX, int vectorY) {
11         this.x += vectorX;
12         this.y += vectorY;
13     }
14     public int getX() {
15         return x;
16     }
17     public void setX(int x) {
18         this.x = x;
19     }
20     public int getY() {
21         return y;
22     }
23     public void setY(int y) {
24         this.y = y;
25     }
26 }
```

PointData.java

Listing 3.8: Klasa **różni się** od poprzedniej **tylko** adnotacją `@Data` (oraz importem do niej). Najważniejszym jest pamiętać, że ta adnotacja wygeneruje (tzn. nadpisze) nam metody: `toString()`, `equals(...)` oraz `hashCode()`

```
1  package aleksander.wojcik.wsei;
2
3  import lombok.Data;
4
5  /**
6   * Created by Aleksander Wojcik WSEI
7   */
8  @Data
9  public class PointData {
10     private int x;
11     private int y;
12
13     public void move(int vectorX, int vectorY) {
14         this.x += vectorX;
15         this.y += vectorY;
16     }
17     public int getX() {
18         return x;
19     }
20     public void setX(int x) {
21         this.x = x;
22     }
23     public int getY() {
24         return y;
25     }
26     public void setY(int y) {
27         this.y = y;
28     }
29 }
```

Program.java

Listing 3.9: Klasa z metodą główną. To w niej będzie generowane wyjście konsoli

```
1  package aleksander.wojcik.wsei;
2
3  import java.util.HashSet;
4  import java.util.Set;
5
6  /**
7   * Created by Aleksander Wojcik WSEI
8   */
9  public class Program {
10     public static void main(String[] args) {
11         printForPoint();
12         newlines();
13         printForPointData();
14     }
15     private static void newlines() {
16         System.out.printf("%n%n%n%n");
17     }
18     private static void printForPoint() {
19         System.out.println("Program.printForPoint");
20         Set set = new HashSet();
21         set.add(new Point());
22         set.add(new Point());
23         set.add(new Point());
24         System.out.println("inserted three Points");
25         System.out.println("set.size() = " + set.size());
26         System.out.println("set = " + set);
27         System.out.println("new Point().toString() = " + new Point().toString());
28     }
29     private static void printForPointData() {
30         System.out.println("Program.printForPointData");
31         Set set = new HashSet();
32         set.add(new PointData());
33         set.add(new PointData());
34         set.add(new PointData());
35         System.out.println("inserted three PointDatas");
36         System.out.println("set.size() = " + set.size());
37         System.out.println("set = " + set);
38         System.out.println("new PointData().toString() = " + new PointData().toString());
39     }
40
41 }
```

Metoda główna najpierw uruchamia procedurę wypisywania tekstu na konsolę dla klasy `aleksander.wojcik.wsei.Point`, wypisuje puste linie, a następnie powtarza procedurę dla klasy `aleksander.wojcik.wsei.PointData`.

Działanie procedury (tzn. metod: `printForPoint()` oraz `printForPointData()`) jest następujące:

1. Wypisuje nazwę metody wywołanej
2. Tworzy pusty zbiór
3. Dodaje do zbioru trzy nowo utworzone punkty
4. Wypisuje na konsolę rozmiar i zawartość zbioru
5. Wypisuje na konsolę wynik metody `toString()` wywołanej na nowo utowrzonym punkcie

3.5.4 Uruchomienie programu

Uruchomić program można każdym z poniższych sposobów:

- dla systemu Windows 7: dwukrotne kliknięcie na plik `program/run/run.bat`
- wydanie w katalogu `program/run` polecenia:
  `java -jar aleksander.jar`
- zaimportowanie projektu z katalogu `program/project` oraz uruchomienie metody głównej
  `aleksander.wojcik.wsei.Program#main`

Wynik działania programu

Listing 3.10: Wynik działania programu. Powinien być on identyczny dla każdego wywołania programu z dokładnością do wartości hashCode'ów poszczególnych obiektów

```
1  Program.printForPoint
2  inserted three Points
3  set.size() = 3
4  set = [aleksander.wojcik.wsei.Point@6cd8f317, aleksander.wojcik.wsei.Point@52db59df,
5  aleksander.wojcik.wsei.Point@173fa2d5]
6  new Point().toString() = aleksander.wojcik.wsei.Point@4e2c390c
7
8
9
10 Program.printForPointData
11 inserted three PointDatas
12 set.size() = 1
13 set = [PointData(x=0, y=0)]
14 new PointData().toString() = PointData(x=0, y=0)
15
16 Process finished with exit code 0
```

Wynik działania programu jest zgodny z teorią działania adnotacji `@Data` projektu Lombok jak i opisaną krótko przeze mnie implementacją zbiorów w języku Java.

Podsumowanie

Projekt Lombok jest małą, konfigurowalną, łatwą w użyciu, instalacji i nauce biblioteką dającą Javie dużo możliwości, rozwiązującą pewne specyficzne problemy i poprawiającą jakość kodu. Jest bezpieczny w użyciu i istnieje zawsze możliwość szybkiego wycofania się z niego poleceniem `delombok`. Projekt Lombok daje nadzieje na szybszy rozwój możliwości języka Java niż wynikałby on z oficjalnych wydań kolejnych wersji języka. Brak mu jednak wysokopoziomowego API, aby łatwe było programowanie własnych adnotacji na potrzeby np. danego projektu. Potencjalne problemy to:

- różnice w działaniu przy różnych wersjach Javy, JVM, biblioteki Lombok
- korzystanie z narzędzi, które nasłuchują kod źródłowy programu, a nie kod skompilowany oraz nie mają wiedzy o specyfice działania biblioteki Lombok

Podsumowując — jeżeli jest możliwość przenieść się na nowsze języki typu Groovy lub Scala, warto to zrobić, bo ich możliwości są z założenia dużo większe. Jeżeli chcemy zostać przy języku Java, warto doinstalować do projektu bibliotekę Lombok z powodów omówionych w niniejszej pracy.

Cele pracy zostały osiągnięte i obszernie wyjaśnione na przykładach, w załącznikach oraz w dostarczonym na płycie, a opisanym w 3.5 programie.

Bibliografia

[1] *Java Tutorial* Oracle Corporation, http://docs.oracle.com/javase/tutorial/

[2] *PJWSTK* http://edu.pjwstk.edu.pl/wyklady/poj/scb/Polimorf/Polimorf.html

[3] *Ważniak* http://wazniak.mimuw.edu.pl/images/9/9d/Zpo-1-wyk.pdf

[4] *Wikipedia* http://pl.wikipedia.org/wiki/Polimorfizm_(informatyka)

[5] *Wikipedia* http://pl.wikipedia.org/wiki/Java

[6] *Blog programistyczny* http://www.programowanieobiektowe.pl/java_hermetyzacja.php

[7] *Wikipedia* https://pl.wikipedia.org/wiki/Program_komputerowy

[8] *Wikipedia* https://pl.wikipedia.org/wiki/Programowanie_obiektowe

[9] *Przeglad jezyków i paradygmatów programowania* Jarosław Bylina Beata Bylina Instytut Informatyki UMCS Lublin 2011 ISBN: 978-83-62773-00-8

[10] *Delombok* https://projectlombok.org/features/delombok.html

[11] *Strona projektu lombok* https://projectlombok.org/features/index.html

[12] *Licencja MIT* http://opensource.org/licenses/mit-license.php

[13] *Wpis z forum* http://stackoverflow.com/questions/3852091/is-it-safe-to-use-project-lombok/12807937#12807937

Zalacznik A

Referat na konferencję KNITS 2015

Słowa kluczowe:
język Java, projekt Lombok, procesor adnotacji, Abstract Syntax Tree

Aleksander WÓJCIK*

PROJEKT LOMBOK JAKO SPOSÓB NA UELASTYCZNIENIE KODU JAVY

STRESZCZENIE

Pierwsza część referatu jest wprowadzeniem do podstawowych mechanizmów języka Java. Omówione zostaną założenia języka, proces kompilacji, rola maszyny wirtualnej oraz zostanie wyjaśnione czym są i po co zostały wprowadzone adnotacje.

W części drugiej zostanie poruszone zagadnienie Abstrakcyjnego Drzewa Składniowego na przykładzie oraz zostaną opisane motywacje, sposób działania i możliwości projektu Lombok.

PROJECT LOMBOK AS A TOOL TO MAKE JAVA CODE MORE ELASTIC

ABSTRACT

The intention of the first part of this essay is to introduce the basics of Java programming language. The assumptions, compilation process, the role of virtual machine and annotation mechanism will be explained.

The second part is to focus on Abstract Syntax Tree and project Lombok - its motivation, way of working and capabilities.

* Wyższa Szkoła Ekonomii i Innowacji w Lublinie, ul. Projektowa 4, 20-209 Lublin.

1. WPROWADZENIE

1.1. OGÓLNE INFORMACJE O JĘZYKU JAVA [1]

Podstawowe koncepcje języka java zostały przejęte z języka Smalltalk (maszyna wirtualna, zarządzanie pamięcią), natomiast składnia i wiele słów kluczowych z języka obiektowo-proceduralnego C++.

Język Java jest silnie zorientowany obiektowo. W kontekście własności języka o obiekcie można myśleć jako o składowej części programu, która może przyjmować określone stany i ma określone metody, które mogą zmieniać te stany bądź przesyłać dane do innych obiektów. Wyjątkiem od obiektowości są typy prymitywne: byte, short, int, long, double, float oraz specjalny typ void.

W Javie wszystkie obiekty są pochodną obiektu nadrzędnego (klasy java.lang.Object), z które-go dziedziczą podstawowe metody. Dzięki temu wszystkie klasy mają wspólny podzbiór podstawowych możliwości. W języku Java klasy mogą być wyprowadzone z innych klasy, w ten sposób dziedziną pola i metody tych klas. Z wyjątkiem klasy Object, która nie ma nadklasy, każda klasa ma jedną i tylko jedną bezpośrednią nadklasę (jednokrotne dziedziczenie). W przypadku braku wyraźnie określonej nadklasy, każda klasa jest bezwzględnie podklasą klasy Object. Podklasa dziedziczy wszystkie składowe (pola, metody oraz zagnieżdżone klasy) z jej nadklasy. Konstruktory nie są składowymi, więc nie są dziedziczone przez podklasy, ale konstruktor może być wywołany z podklasy poprzez użycie słowa kluczowego super(). Klasa Object, zdefiniowania w pakiecie java.lang, definiuje i implementuje zachowania wspólne dla wszystkich klas.

Każdy kod źródłowy programu składa się z zestawu klas pogrupowanych w pakiety.

Po napisaniu kodu źródłowego program kompilowany jest do bytecodu. Nie jest to jeszcze kod zrozumiały dla procesora w sposób bezpośredni, który pozwalałby nam na jego uruchomienie. Jest to jednak kod zapisany w określonym formacie, który może zostać poprawnie zinterpretowany przez Maszynę Wirtualną Java JVM (z ang. Java Virtual Machine), przetłumaczony na kod wykonywalny i uruchomiony.

Ważne jest to, że w Javie nie możemy bezpośrednio skompilować program do kodu maszynowego. Program zawsze jest kompilowany do kodu pośredniego, który jest następnie interpretowany przez JVM. JVM rozpoznaje system operacyjny oraz sprzęt, na jakim jest program odpalany i dostosowuje do niego wynikowy kod maszynowy. W ten sposób realizowana jest koncepcja przenośności programów napisanych w Javie. JVM zarządza również pamięcią.

1.2. KOMPILACJA I URUCHOMIENIE PROGRAMU KORZYSTAJĄC Z JVM

Pliki źródłowe, które mogą być kompilowane mają rozszerzenie *.java. Kompilacja polega na wydaniu polecenia javac z argumentem pliku źródłowego. Powstaje wówczas plik *.class. W procesie kompilacji kompilator Java (program javac) tłumaczy kod programu na Byte Code JVM – pliki z rozszerzeniem .class. W czasie działania aplikacji Byte Bode jest tłumaczony, przez JVM, na kod maszynowy i wykonywany bezpośrednio na procesorze. Kod maszynowy wytworzony przez JVM nie jest nigdzie zapisywany, jest on generowany “w locie”.

Przykładowo, aby uruchomić plik MyProgram.java z metodą główną wykonujemy kolejno:

```
javac MyProgram.java
java MyProgram
```

Rys 1. Na rysunku zilustrowany został proces kompilacji programu MyProgram.java do kodu pośredniego (pliku MyProgram.class) za pomocą kompilatora javac oraz symbolicznie ukazana interpretacja kodu pośredniego do kodu maszynowego (zera i jedynki) przez JVM

1.3. ADNOTACJE

Nowością w stosunku do starszych języków programowania jest mechanizm adnotacji zapożyczony z jezyka C#. Adnotacje to dokładnie – metadane. Mogą być stosowane wobec: klasy, metody, interfejsu, pola klasy, argumentu metody, konstruktora, pakietu, zmiennej lokalnej, innej adnotacji.

Adnotacje są podstawowym narzędziem programowania aspektowego, zarządzania bean-ami w kontenerach Dependency Injection, konfiguracji systemów opartych na frameworkach klasy enterprise (Spring, EJB). Natomiast w projekcie Lombok stanowią pewnego rodzaju „polecenia generacji kodu”, o czym poniżej. Warto zwrócić uwagę, że w kontekście projektu Lombok adnotacje przestają być już tylko metadanymi, ale zyskują nowe znaczenie - stanowią pewnego rodzaju instrukcje dla procesora jaki kod należy wygenerować przed ostateczną kompilacją programu do kodu pośredniego.

Przykładowymi adnotacjami w JDK są:

- `@Override`
  Adnotacją tą oznaczane są metody, których przeznaczeniem jest nadpisanie

metody z klasy nadrzędnej. Jeżeli kompilator nie znajdzie w klasie nadrzędnej metody o tej samej (z dokładnością do kowariantnych typów danych) sygnaturze – wówczas program się nie skompiluje

- `@SuppressWarnings("unchecked")`
  W skrócie powoduje, że ostrzeżenia kompilatora typu „unchecked” nie będą wyświetlane na konsoli.

1.4. DZIAŁANIE PROCESORA ADNOTACJI [4]

Procesor adnotacji działa w trakcie kompilacji programu i jest możliwe zaprogramowanie odpowiednich procedur w kontekście użytych adnotacji. Deklarowany przez nas procesor adnotacji dziedziczy bezpośrednio z klasy `javax.annotation.processing.AbstractProcessor`. Musimy zadbać o nadpisanie jednej metody abstrakcyjnej:

```
public abstract boolean process(Set<? extends
TypeElement> annotations,RoundEnvironment roundEnv);
```

Metoda zwraca typ logiczny - mówiący o tym, czy dane adnotacje zostały przez ten konkretny procesor obsłużone. Jeśli metoda zwróci `true`, wtedy kolejne procesory nie będą proszone o obsłużenie danej adnotacji. W kontekście referatu ważniejsze są argumenty metody niż to, co ona zwróci.

Pierwszym argumentem jest zbiór typów, które są żądane do obsłużenia. W szczególności procesor musi umieć zareagować na sytuację, gdy zbiór jest pusty.

Drugim argumentem jest „środowisko” zawierające w sobie informacje o kontekście użycia adnotacji. Jest to dość rozbudowana klasa.

Najważniejszą metodą środowiska jest :

```
Set<? extends Element> getElementsAnnotatedWith(Class<?
extends Annotation> a)
```

Która zwraca wszystkie elementy oznaczone daną adnotacją, dzięki czemu możliwe jest iterowanie po nich i wymuszenie spójnego działanie adnotacji. To właśnie w tej "iteracji" zachodzi właściwe działanie instrukcji, wymuszonych użyciem adnotacji.

Warto na tym etapie zaznaczyć, że procesor adnotacji nie umożliwia edytowania już istniejących i załadowanych klas. W przypadku adnotacji @Override procesor zgłasza wyjątek i przerywa kompilacje, gdy warunek, że metoda oznaczana adnotacją nadpisuje metodę nadklasy, nie jest spełniony. Adnotacja @SuppressWarnings blokuje pisanie niektórych komunikatów na wyjście. W obu przypadkach nie ma modyfikacji działania danej klasy.

Warto zaznaczyć, że jest możliwe definiowanie nowych klas w ramach działania procesora adnotacji.

2. PROJEKT LOMBOK

2.1 UELASTYCZNIENIE KODU JAVA JAKO CEL POWSTANIA PROJEKTU

Zwyczajową nazwą publicznej metody, której jedynym celem jest zwrócenie wartości jej pola jest nazwa „getter”. Zwyczajową nazwą publicznej metody, której jedynym celem jest zmienienie wartości jej pola jest nazwa „setter”.

Jedną z poważniejszych wad języka Java jest generowanie dużej ilości metod zbędnych – np. getterów i setterów - mających za zadanie kontrolę nad dostępem do pól klasy zgodnie z podstawowym założeniem obiektowego paradygmatu programowania – hermetyzacji. Inne języki radzą sobie z tym lepiej. Poniżej porównanie tej samej funkcjonalności – klasy Square z polem side – w językach C# oraz Java:

```
//Java
public class Square{
      private int size;
      public void setSide (int size){
            this.size = size
      }
      public int getSide (){
            return this.size;
      }
}

//C#
public class Square {
      private int Side{get;set;};
}
```

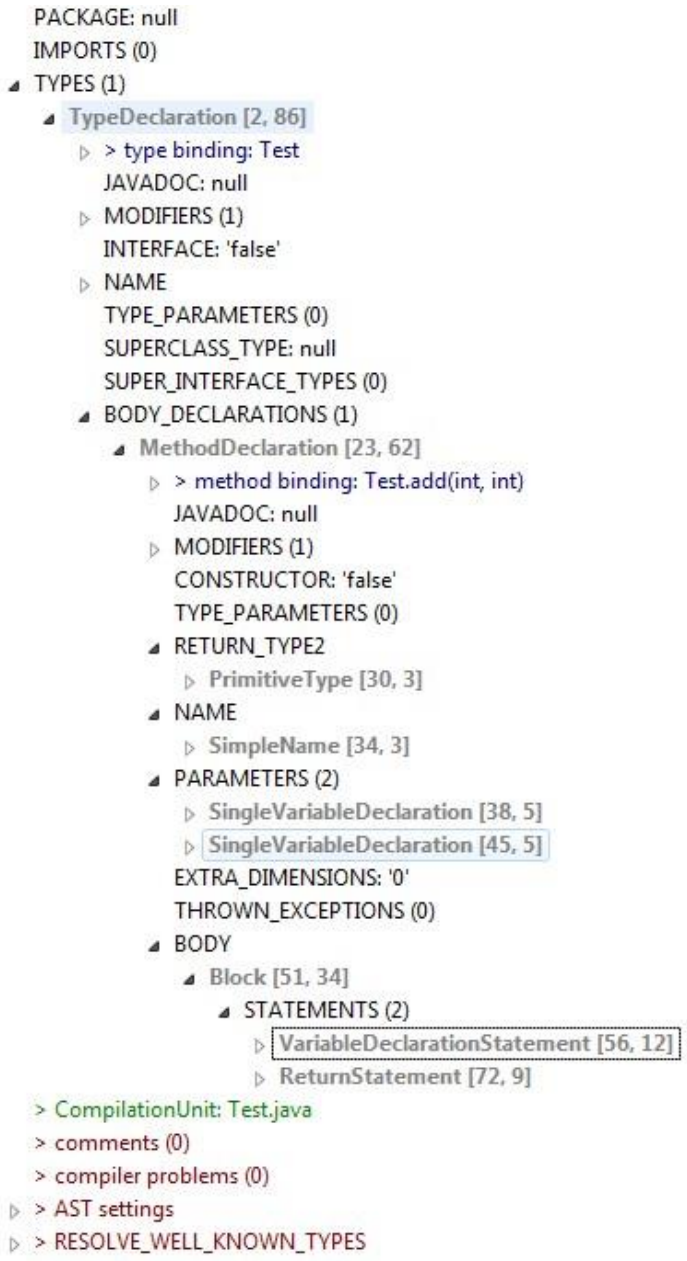
Zazwyczaj programista nie pisze getterów i setterów bez użycia IDE. W każdym podstawowym IDE (Intellij IDEA, Eclipse, Netbeans) jest możliwość automatycznej generacji w/w metod. Jednak za każdym razem, gdy programista zmienia dane pole powinien usunąć stare gettery i settery oraz wygenerować nowe, co jest zadaniem niepotrzebnie uciążliwym. Inne metody, które są często generowane przez IDE: equals(), toString(), hashCode().

2.2 ABSTRAKCYJNE DRZEWO SKŁADNIOWE (AST) [2]

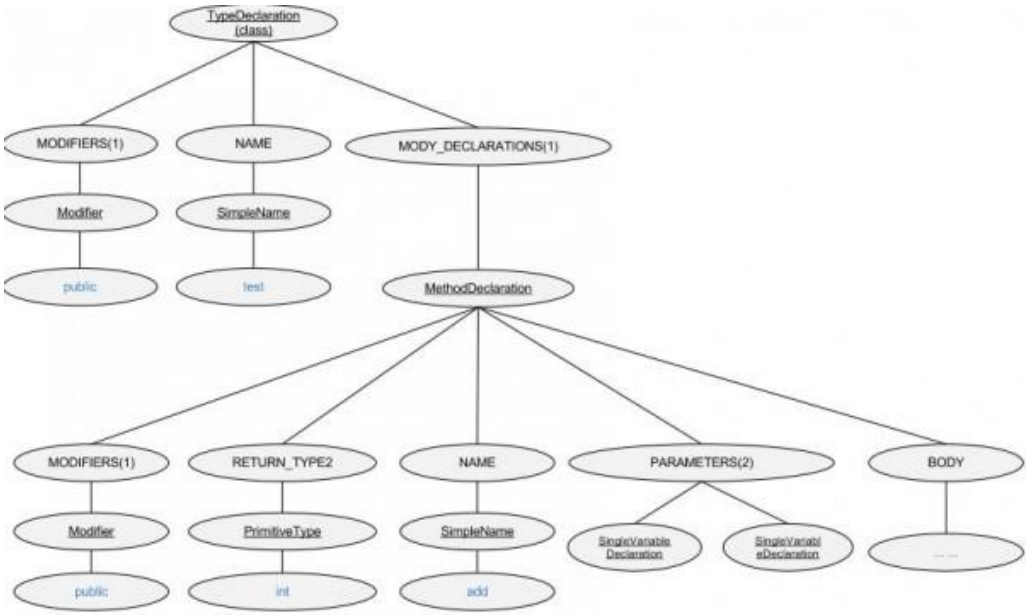
Abstrakcyjne Drzewo Składniowe – z ang. Abstract Syntax Tree (AST)) - jest reprezentacją programu źródłowego napisanym w języku programowania np. Java w postaci drzewa. Powstaje ono w jednym z etapów kompilacji kodu źródłowego. Jest ono dużo mniej czytelne dla programisty niż kod źródłowy, jednak jest niezbędne dla procesu kompilacji. Poniżej porównanie tej samej funkcjonalności w postaci kodu Java oraz w postaci drzewa.

```
public class Test {
        public int add(int a, int b){
                int c = a+b;
                return c;
        }
}
```

Rys 2. klasa Test z metodą add



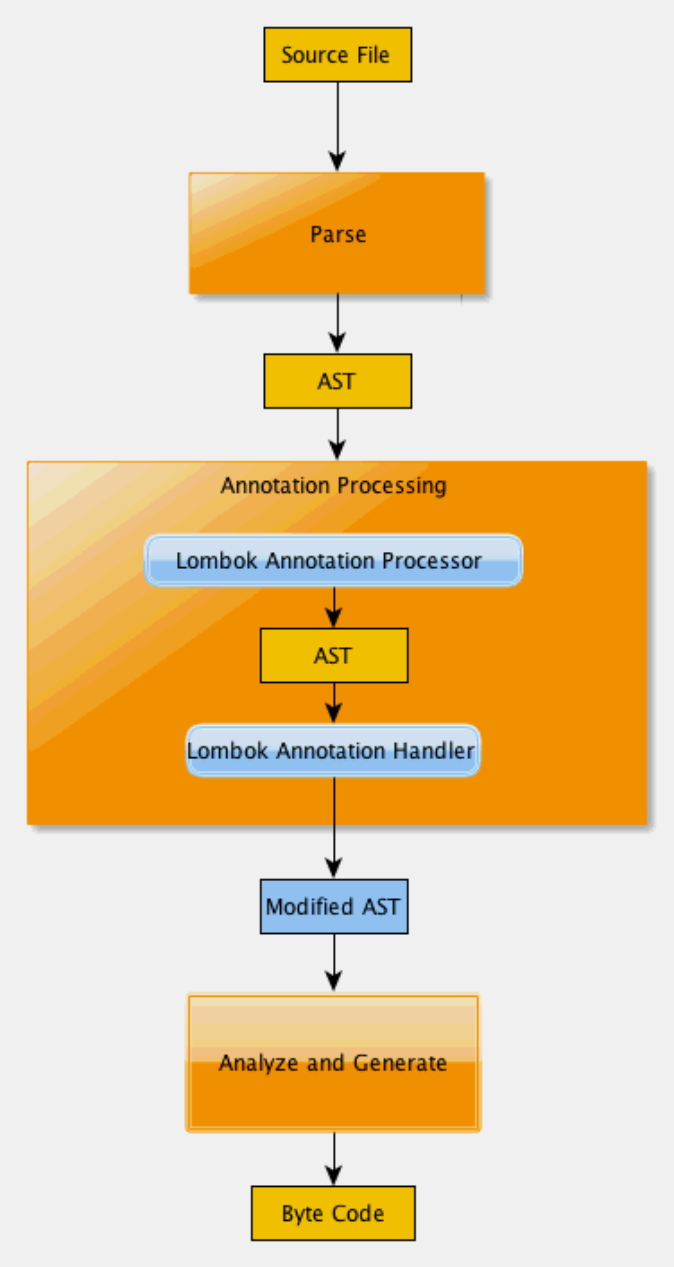
Rys 3. klasa test z metodą add. W AST jest bardzo dużo elementów, których nie ma w kodzie np. informacje, że klasa nie jest interfejsem: INTERFACE: false



Rys 4. Reprezentacja klasy test w postaci drzewa AST bez ciała metody – w węźle pod BODY wstawiony został trzykropek

2.3 MANIPULACJA AST PODSTAWOWYM NARZĘDZIEM DZIAŁANIA PROJEKTU LOMBOK

Projekt Lombok w założeniu wychodzi poza ograniczenia działania procesora adnotacji, ponieważ z założenia działa inaczej w fazie kompilacji:



Rys 5. Proces kompilacji pliku źródłowego (Source File) do kodu pośredniego ([Java} Byte Code) z wykorzystaniem modyfikacji drzewa AST

Projekt Lombok korzysta z narzędzi dostarczonych przez firmę Sun (podpakietów pakietu com.sun.tools.javac) do manipulacji na AST oraz swoich metod pomocniczych tworzących zupełnie nowe API. Mimo, że całe działanie jest wykonywane w języku Java, API do manipulacji AST jest skomplikowane.

Każda adnotacja jest obsługiwana przez klasę dziedziczącą z abstrakcyjnej klasy generycznej `lombok.javac.JavacAnnotationHandler<T extends Annotation>` i nadpisującą jej metodę abstrakcyjną:

```
public abstract void handle(AnnotationValues<T> annotation, JCAnnotation ast, JavacNode annotationNode);
```

Warto zwrócić uwagę, że w projekcie Lombok sygnatura metody jest inna niż w przypadku korzystania ze zwykłego procesora adnotacji (`AbstractProcessor`). W parametrze występują następujące argumenty:

- obiekt typu AnnotationValues – właściwa adnotacja, która aktualnie jest obsługiwana
- obiekt typu JCAnnotation – węzeł AST reprezentujący adnotację
- obiekt typu JavacNode – specjalny obiekt projektu Lombok opakowujący węzeł AST – węzeł ten rozszerza możliwości domyślnego węzła AST z JDK Javy o możliwość przechodzenia w górę i w dół po drzewie AST. Jest to funkcjonalność zaimplementowana właśnie w projekcie Lombok. Z tego węzła drzewa AST możliwe jest też czytanie i zmienianie innych węzłów AST

Poniżej dla przykładu nagłówek odpowiedzialny za obsłużenie adnotacji @Getter [3]:

```
@Override
public void handle(AnnotationValues<Getter> annotation,
  JCAnnotation ast, JavacNode annotationNode){
      ///…
}
```

2.4 PRZYKŁADY MOŻLIWOŚCI, JAKIE PROJEKT LOMBOK OFERUJE [3]

1. Generowanie getterór i setterów – poniżej – zwięzła wersja klasy Square z polem side:

```
public class Square{
     @Getter
     @Setter
     private int size;
}
```

Z kolei poniższa równoważna deklaracja klasy spodowuje wygenerowanie getterów i setterów dla wszystkich prywatnych niefinalnych pól klasy, jeśli byłoby ich więcej:

```
@Getter
@Setter
public class Square{
      private int size;
}
```

2. Generowanie getterór i setterów, metod toString(), equals(), hashCode() oraz konstruktora z parametrami analogicznymi do pól klasy. Wszystko powyższe jest zawarte w zwięzłej adnotacji @Data:

```
@Data
public class Square{
      private int size;
}
```

3. Metody oznaczane adnotacją @SneakyThrows mogą wyrzucać *checked exceptions*, (tj. wyjątki, które w normalnym działaniu programu muszą być obsłużone lub umieszczone w sygnaturze metody) bez konieczności deklarowania ich w sygnaturze metody.

4. Tworzenie niezmiennych klas bez dodatkowych zabiegów programistycznych – adnotacja @Value na klasie

3. PODSUMOWANIE

Programy napisane w języku Java nie są bezpośrednio kompilowane do kodu maszynowego rozumianego przez procesor, ale do kodu pośredniego interpretowanego przez JVM. Sam proces kompilacji też jest wieloetapowy, co pozwala na uruchamianie odpowiednich programów (narzędzi) pomiędzy poszczególnymi etapami kompilacji – np. projektu Lombok.

Adnotacje w języku Java są metadanymi dla pól, metod, klas, argumentów, a także dla innych składowych języka Java. W projekcie Lombok stanowią natomiast instrukcje dla procesora – powodują generowanie dodatkowego kodu w trakcie kompilacji. Używanie adnotacji projektu Lombok powoduje zwiększenie czytelności kodu, zmniejszenie objętości kodu, a także daje nowe możliwości (np. adnotacja @SneakyTrows) niedostępne w inny sposób. Istnieje możliwość rozszerzenia projektu Lombok i zaprogramowania własnych adnotacji. Wykracza to poza zakres tego referatu.

LITERATURA

[1] Java Tutorial, Oracle Corporation, data odwiedzin:14.07.2015r.
http://docs.oracle.com/javase/tutorial/
[2] Wpis na blogu programistycznym, data odwiedzin:14.07.2015r.
http://www.programcreek.com/2012/04/represent-a-java-file-as-an-astabstract-syntax-tree/
[3] Strona główna projektu Lombok, data odwiedzin:14.07.2015r.
https://projectlombok.org/features/
[4] Fragment dokumentacji JDK Javy , Oracle Corporation, data odwiedzin:14.07.2015r.
http://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/AbstractProcessor.html

Zalacznik B

Prezentacja na konferencję KNITS 2015

Projekt Lombok jako sposób na uelastycznienie języka Java

Aleksander Wójcik
Informatyka IV stacjonarne
spec. inżynieria oprogramowania

Plan prezentacji

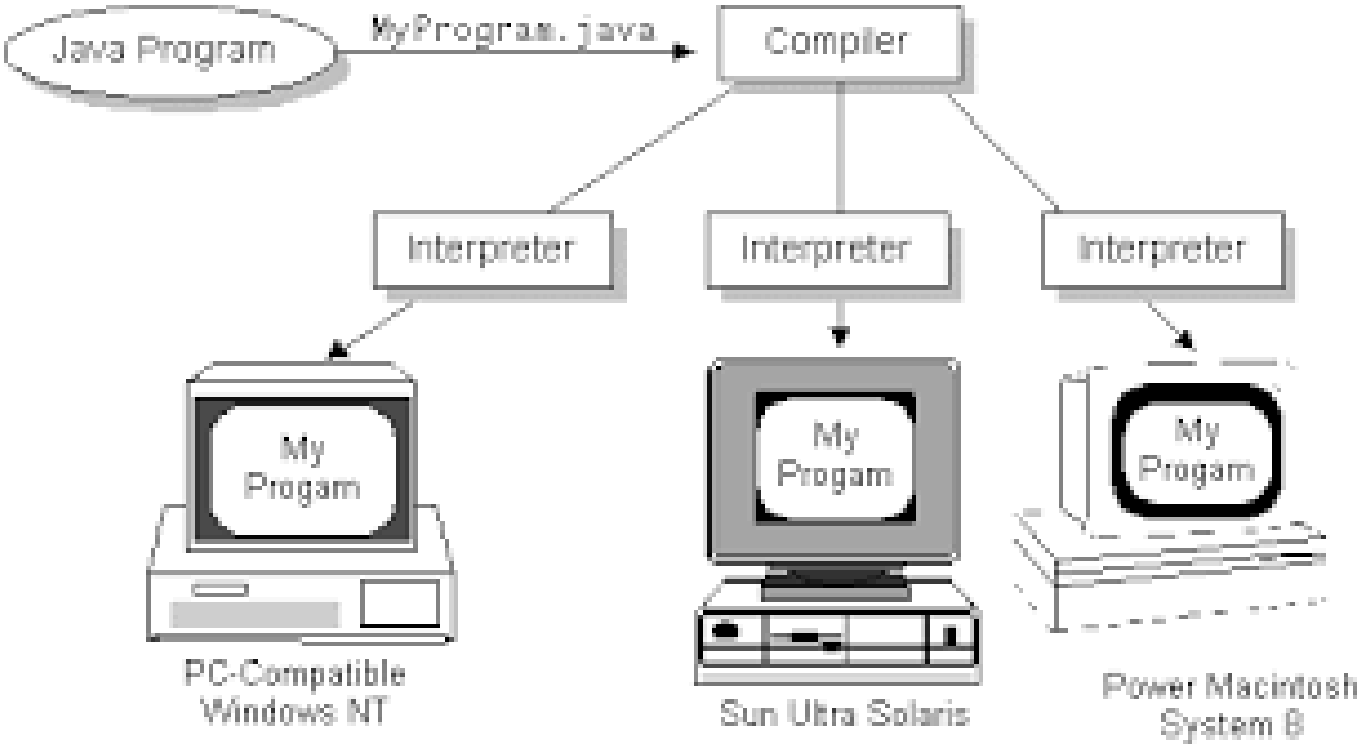
- Powstanie języka Java
- Opis języka i maszyny wirtualnej
- Zalety i problemy języka
- Próby rozwoju
- Projekt Lombok

Java - 1995

- **Niezależność od sprzętu**
- Projektowany do ogromnych projektów

- Garbage Collector, wyjątki, obiektowość

Niezależność od sprzętu



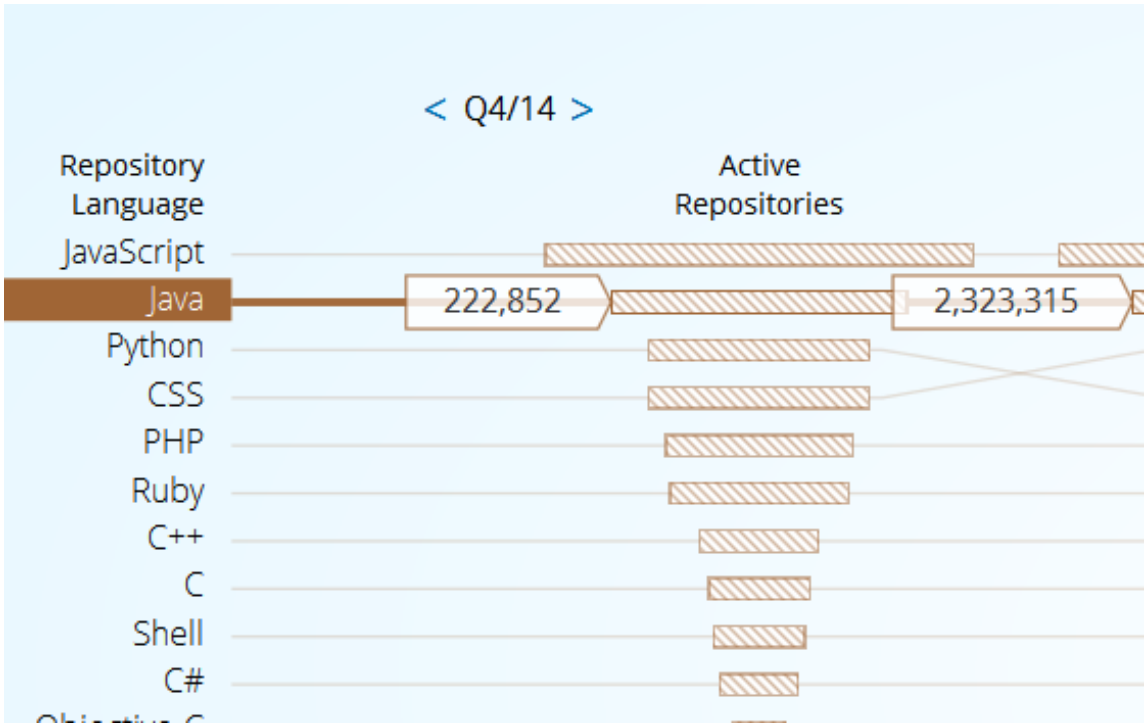
Java - 2015

- Wolna ewolucja języka – dla programistów
- Firma Oracle
- Pamięciożerność

- Wolna ewolucja języka – dla korporacji
- Ogromna ilość bibliotek i **dobrego softu**
- Ogromna architektura enterprise
- Kompatybilność wstecz

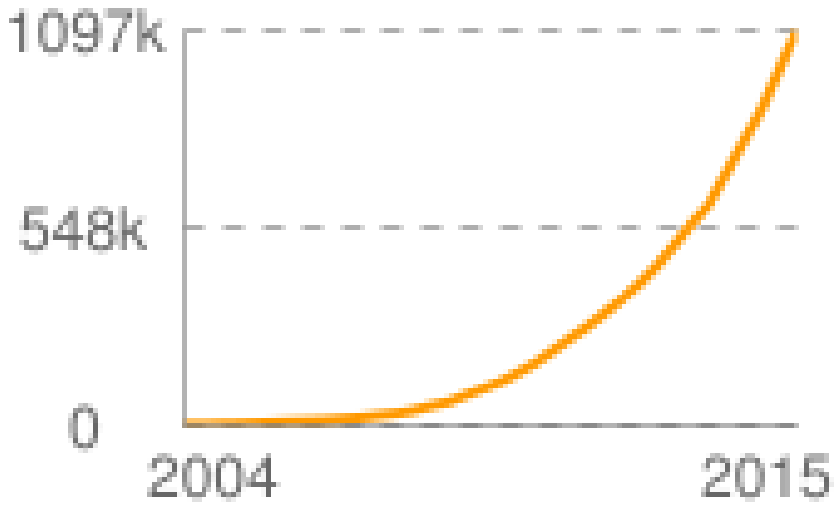
Biblioteki open source

- Github

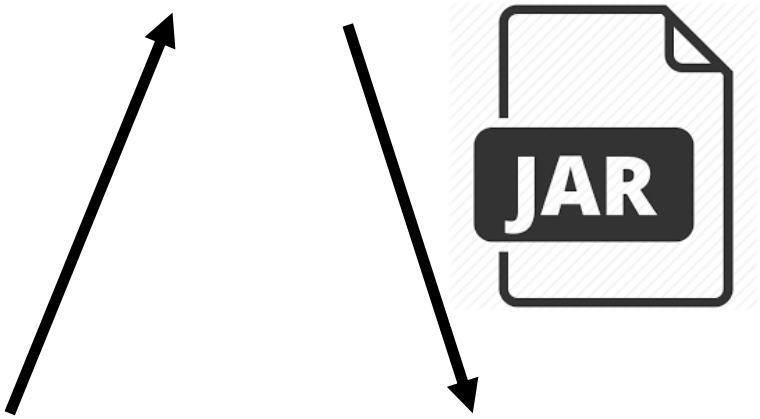


Biblioteki open source

- Maven

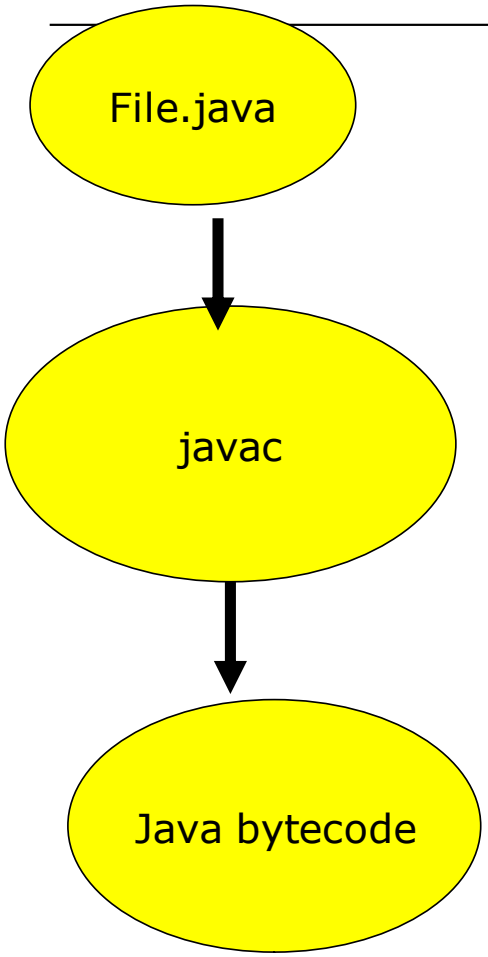


Maven™

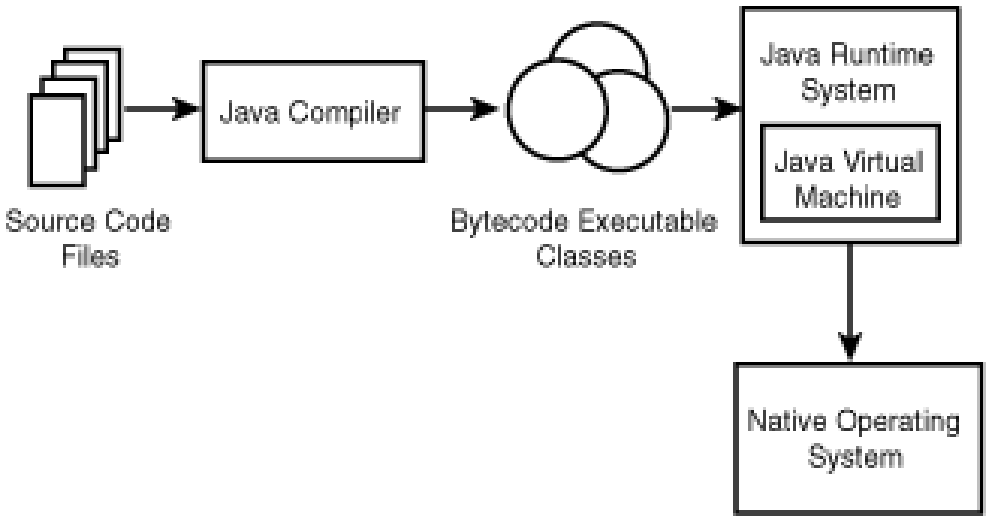


```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.36</version>
</dependency>
```

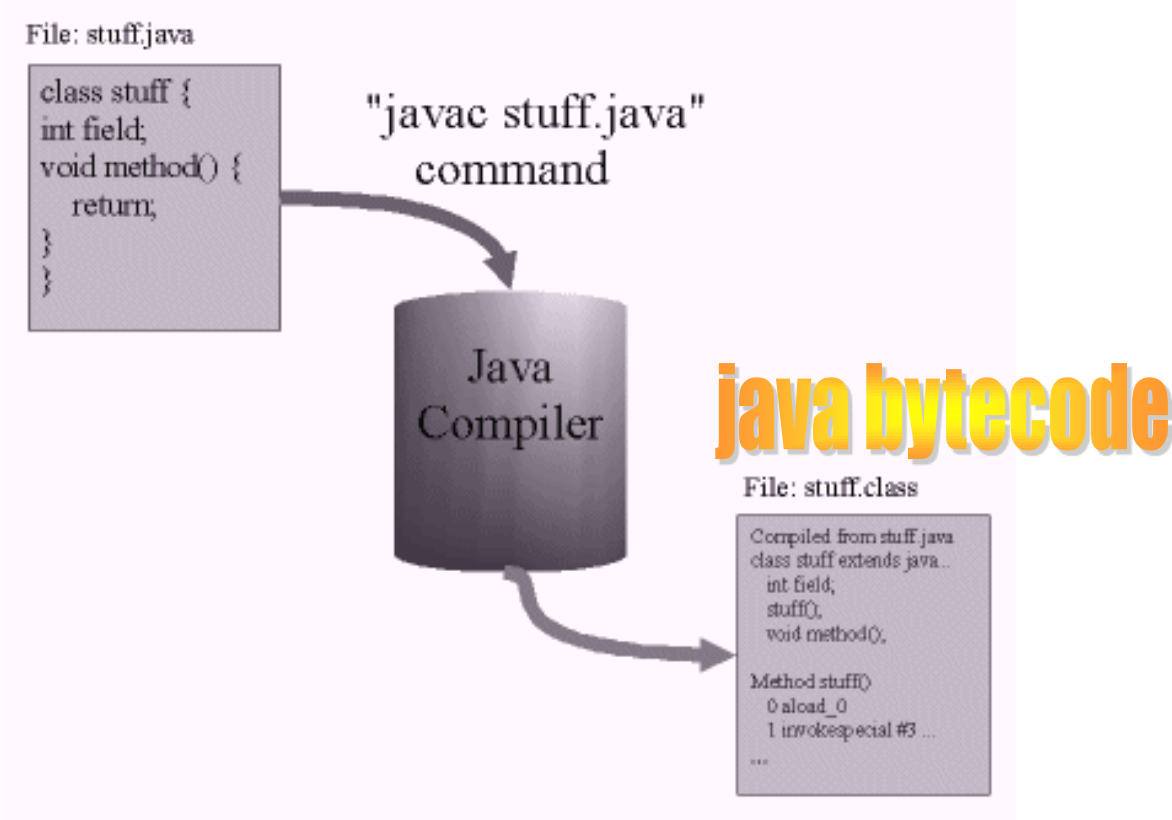
Kompilacja programu Java



Kompilacja programu



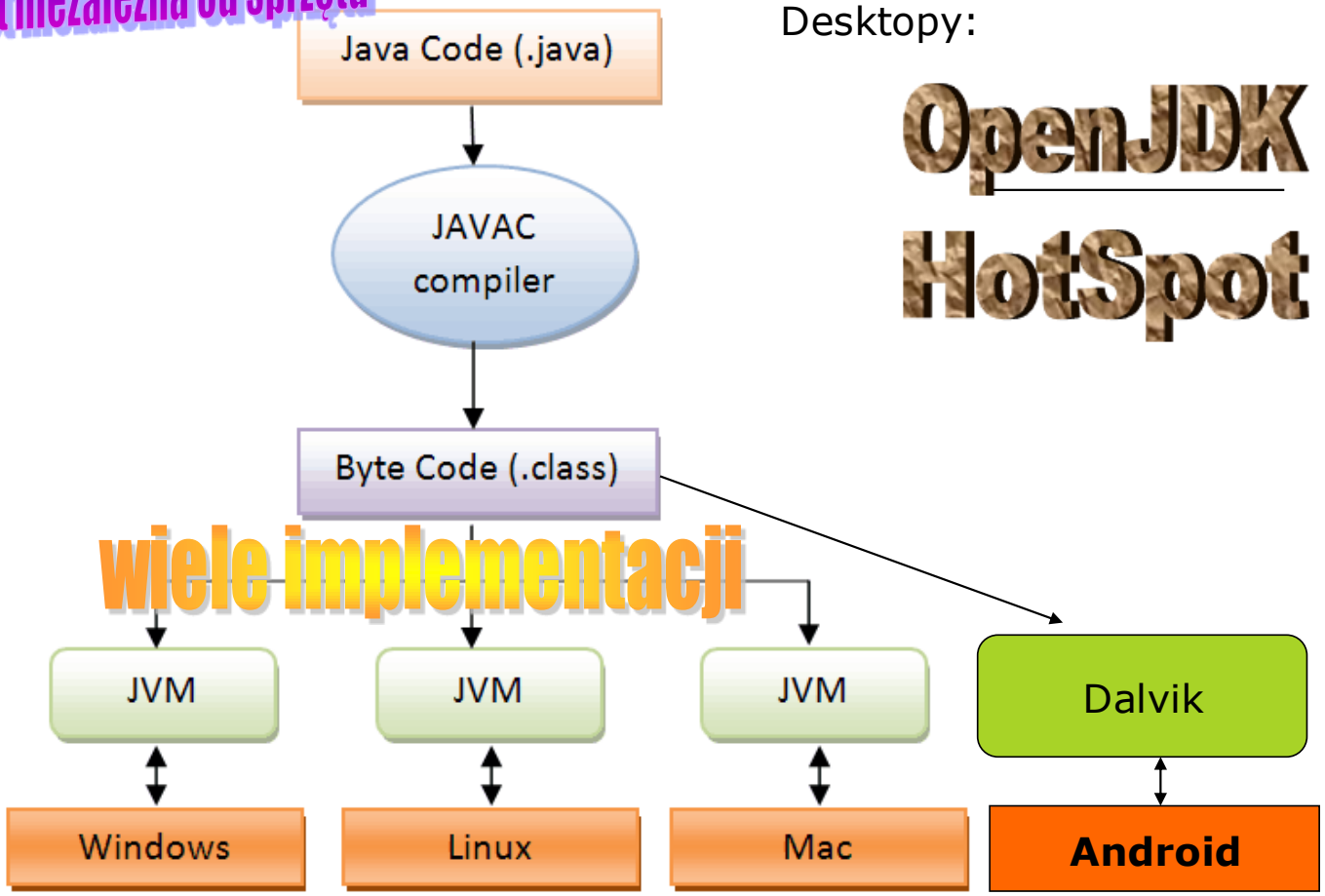
Kompilacja programu



Maszyna wirtualna oraz środowisko zdolne do wykonywania **kodu bajtowego** Javy

- 1. System operacyjny
- 2. Hardware – procesor itp.

Java jest niezależna od sprzętu



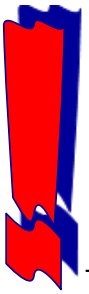
Desktopy:

OpenJDK

HotSpot

Desktopy

Smartfony



Problem – słaba ewolucja

- **Assert –** nowe słowa kluczowe

```
public void assert(){

}
```

- JDK <=1.3

- JDK >1.4

Nowe feature'y - Java vs C#

	Java	C#
val/var	Brak – tylko Lombok	2007
lambda	2014	2007

Cele rozszerzeń Javy

Zostaje!:

1. Biblioteki – JDK,
2. JVM – środowisko uruchomieniowe
3. Niezależność od sprzętu

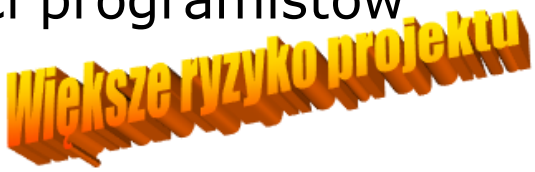
Nowe:

1. Składnia
2. Nowe elementy języka
3. Paradygmaty programowania (funkcyjne)
4. Możliwość typowania dynamicznego

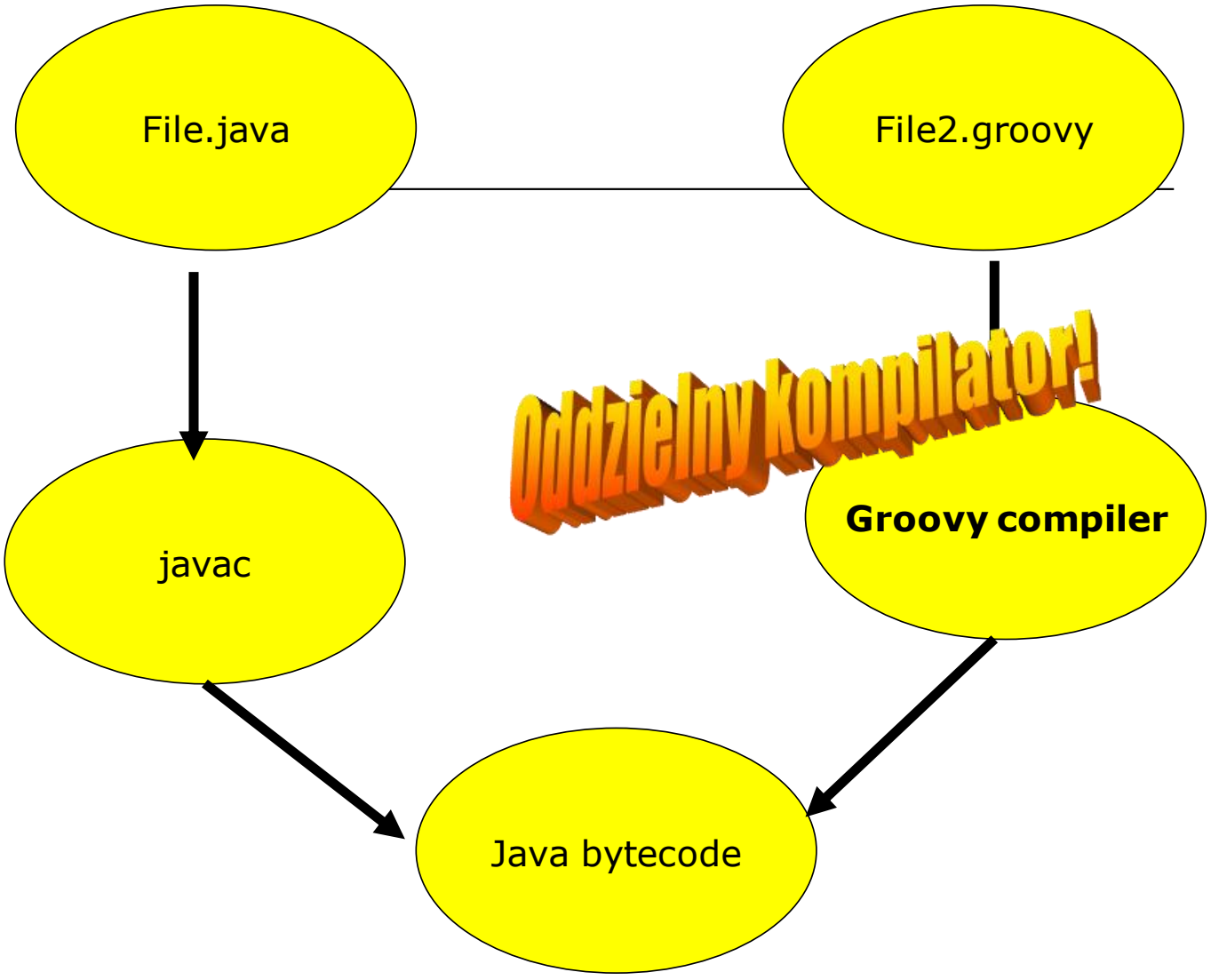
JVM Languages

Potrzeba osobnego kompilatora

Niski udział w rynku -> uzależnienie projektu od dostępności programistów



- Rozszerzona funckjonalność języka
- Korzystają ze wszystkich dobrodziejstw Javy:
  - Bibliotek - Maven, Github itp.
  - JDK

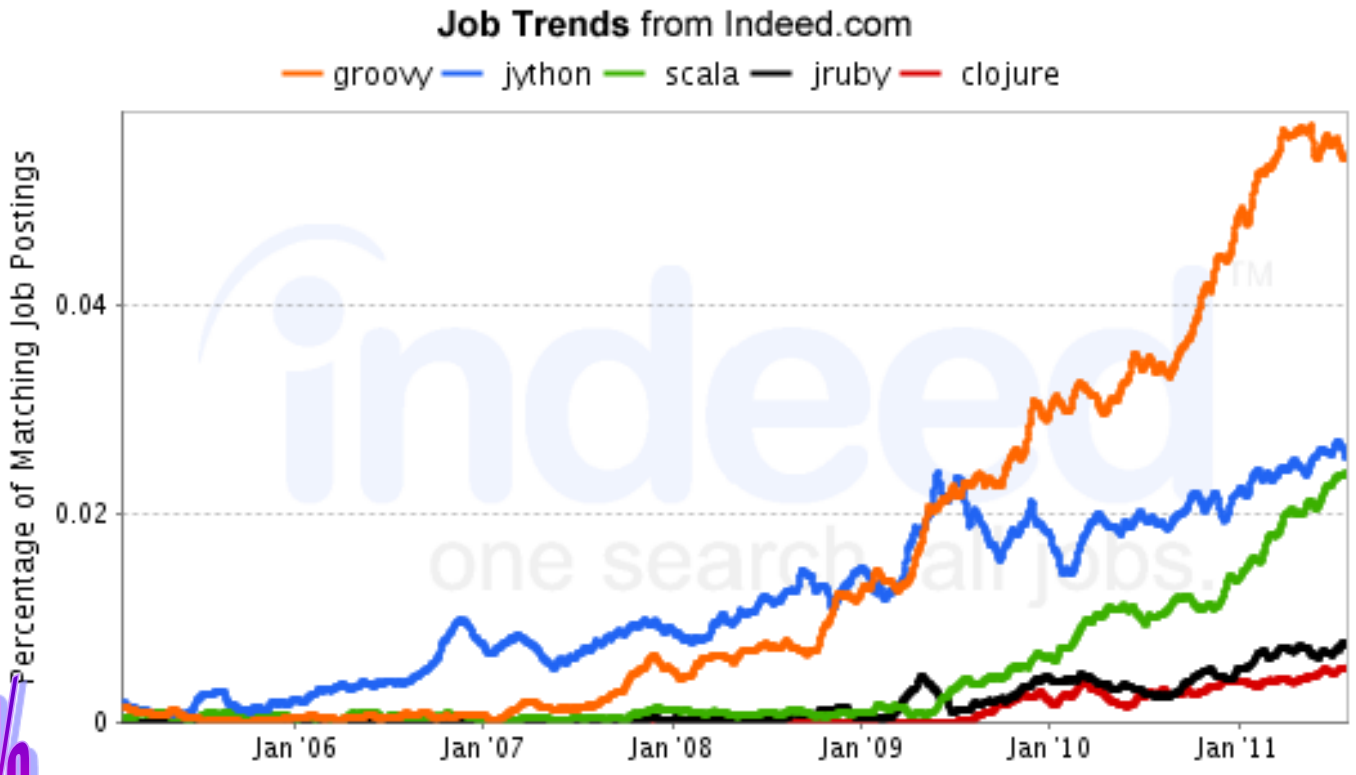

File.java
File2.groovy
Oddzielny kompilator!
javac
Groovy compiler
Java bytecode

JVM Languages

Lessons Learned:
Use of Modern JVM Languages Besides Java

MaibornWolff et al

Niskie zainteresowanie pracodawców

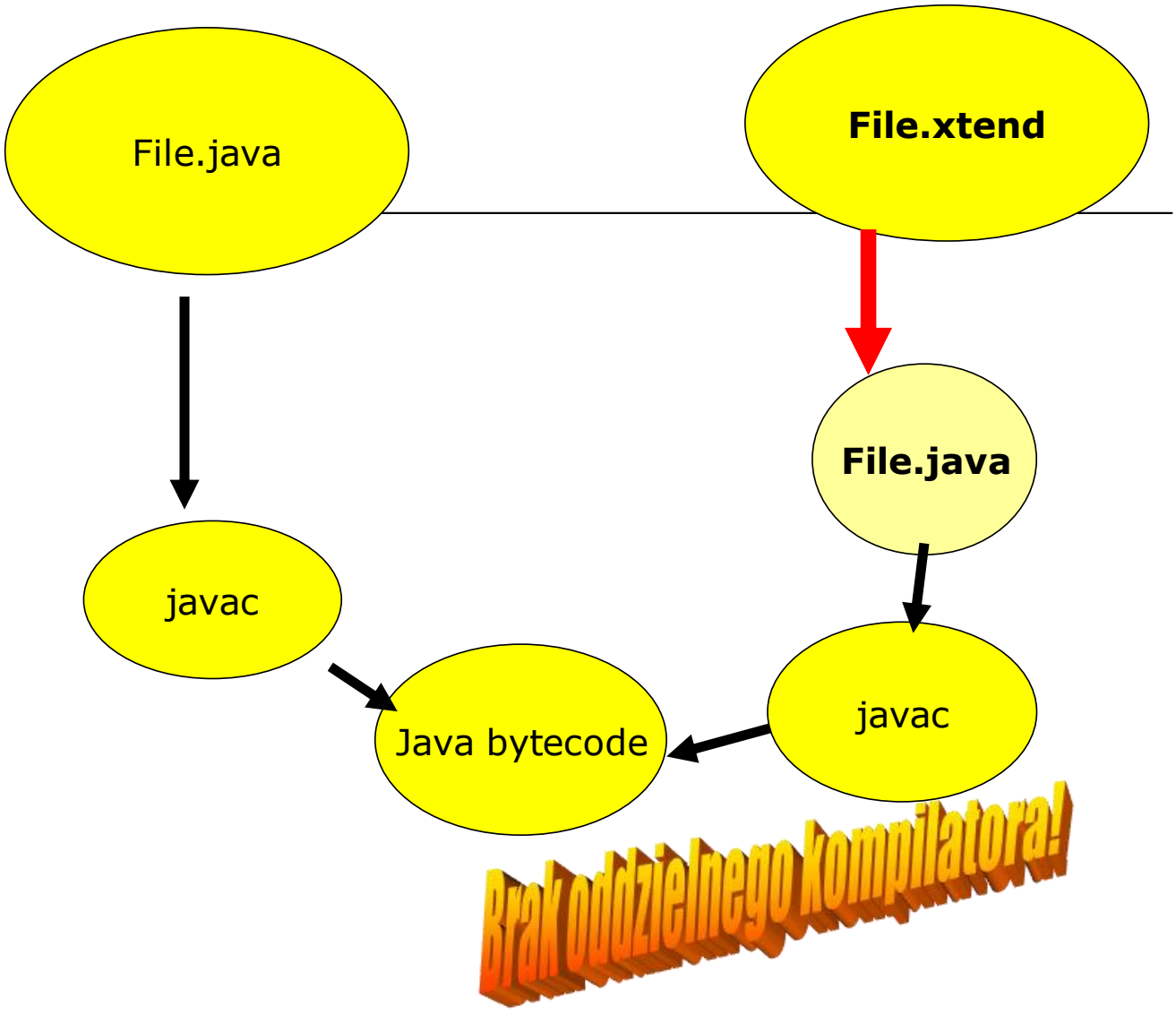


19%

Xtend

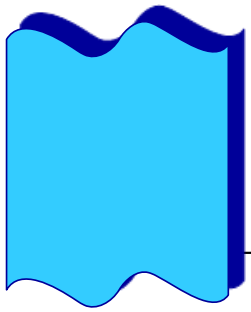
Słabe wsparcie IDE

- Nauka w 1-2 dni
- Brak konieczności osobnego kompilatora


File.java
File.xtend
File.java
javac
javac
Java bytecode
Brak oddzielnego kompilatora!

```
1. class HelloWorld {
2.   def static void main(String[] args) {
3.     println("Hello World")
4.   }
5. }
```

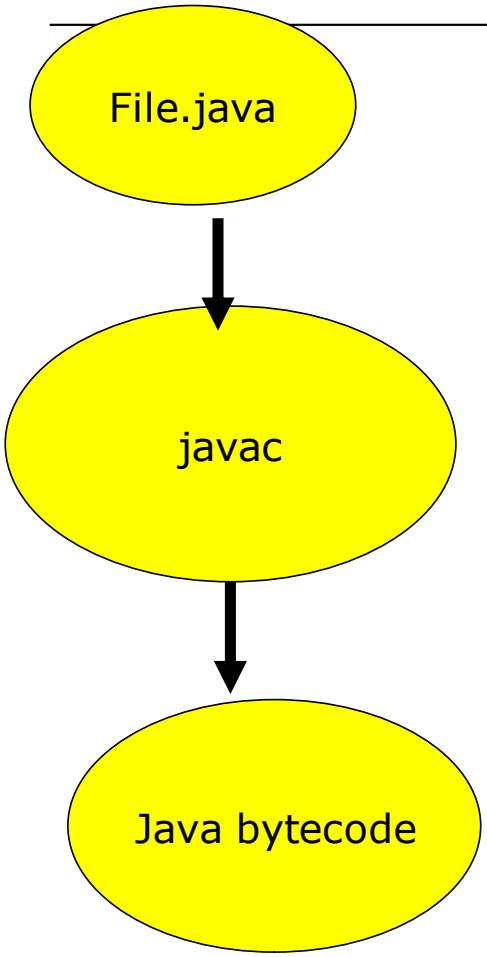
```
1. // Generated Java Source Code
2. import org.eclipse.xtext.xbase.lib.InputOutput;
3.
4. public class HelloWorld {
5.   public static void main(final String[] args) {
6.     InputOutput.<String>println("Hello World");
7.   }
8. }
```



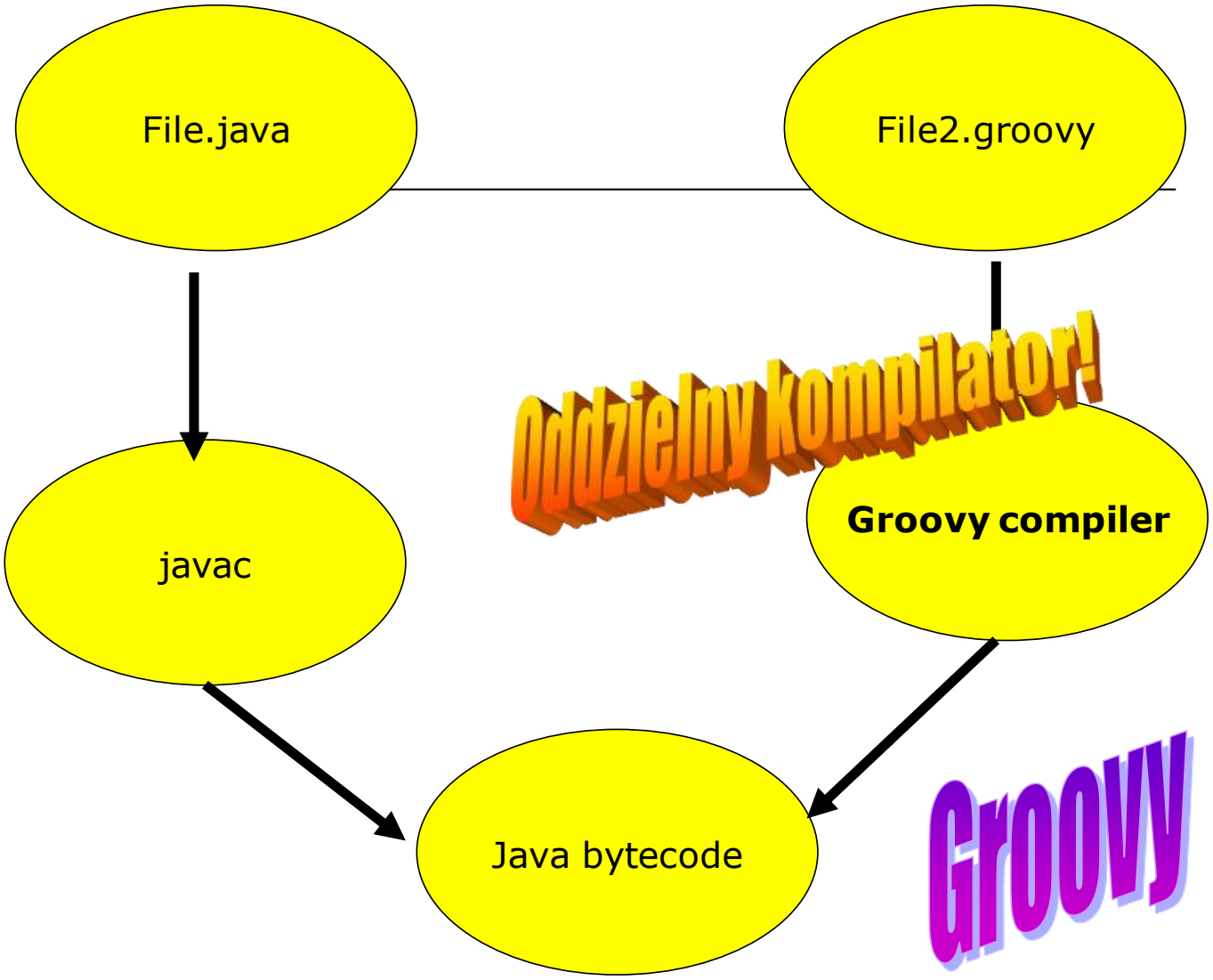
Projekt Lombok

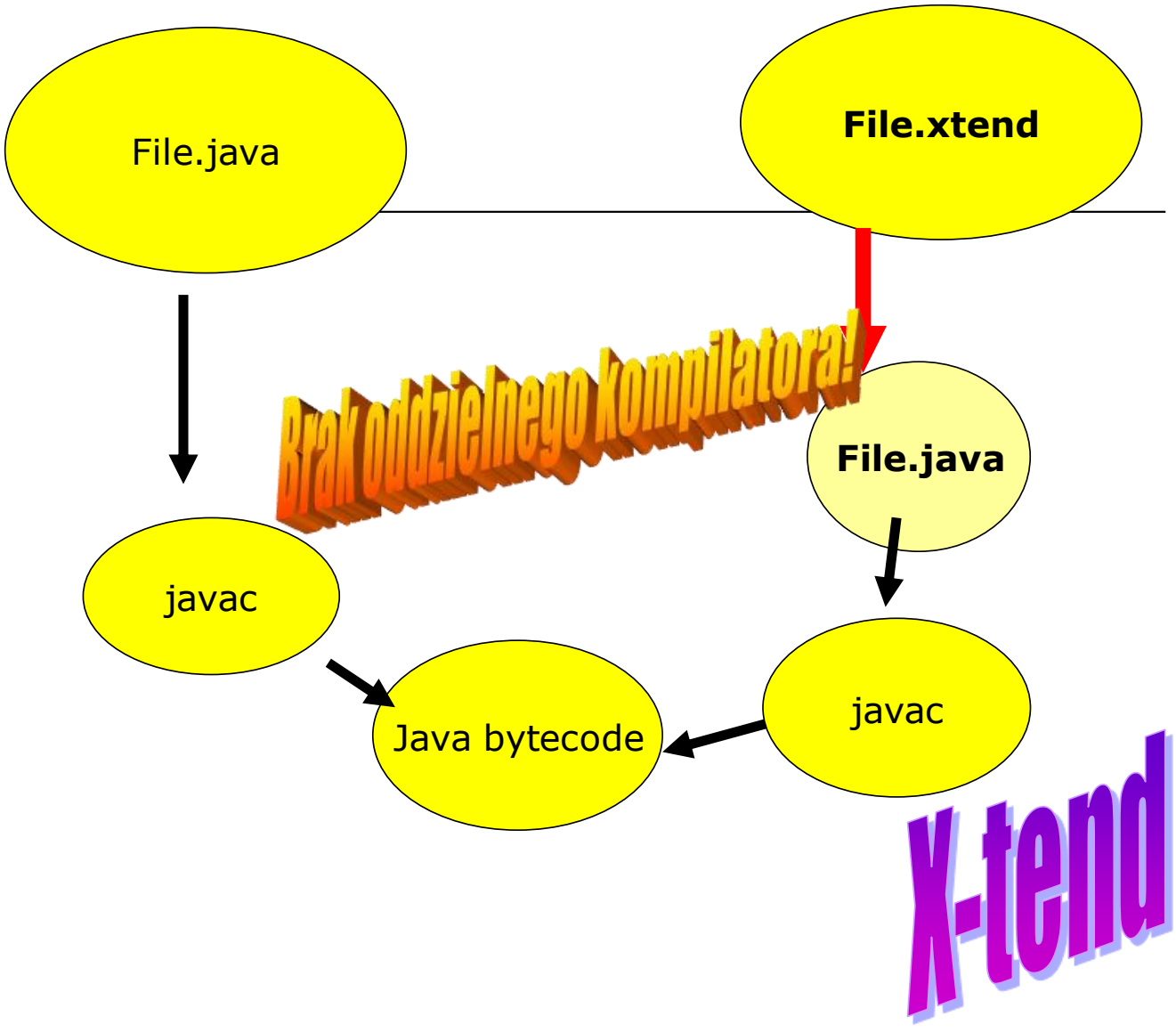
- Naruszenie kontraktu adnotacji
- Konieczność generowania AST dla każdego IDE

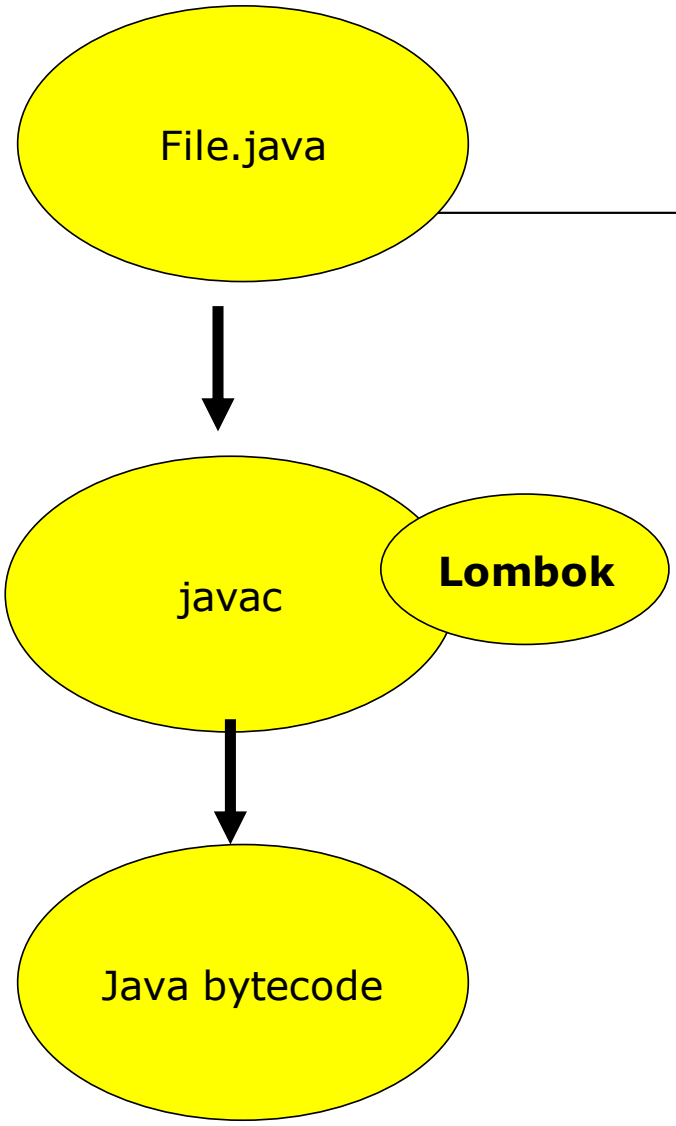
- Nauka w 1-2 godziny
- Możliwość definiowania własnych adnotacji – b. trudna


File.java
javac
Java bytecode

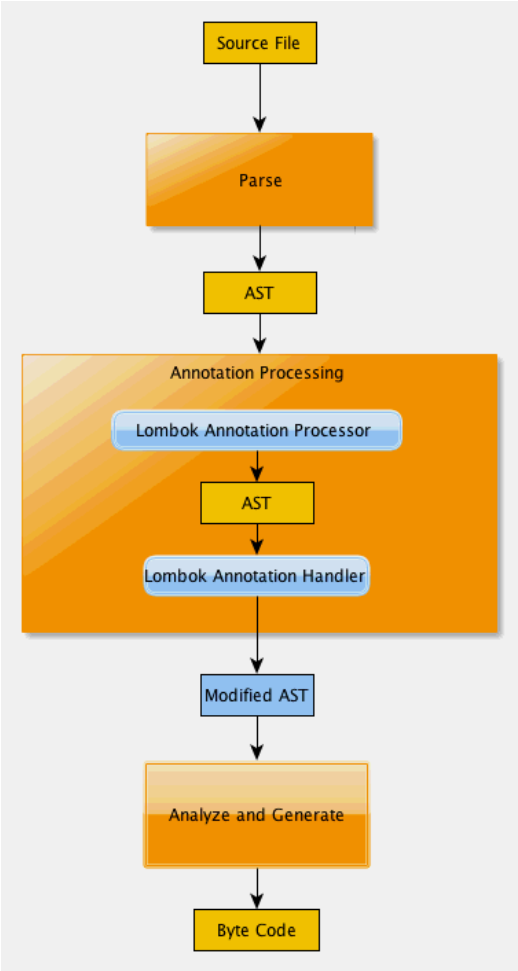

Java


File.java
File2.groovy
Oddzielny kompilator!
javac
Groovy compiler
Java bytecode
Groovy


File.java
File.xtend
Brak oddzielnego kompilatora!
File.java
javac
javac
Java bytecode
X-tend



Lombok



AST-

Abstract

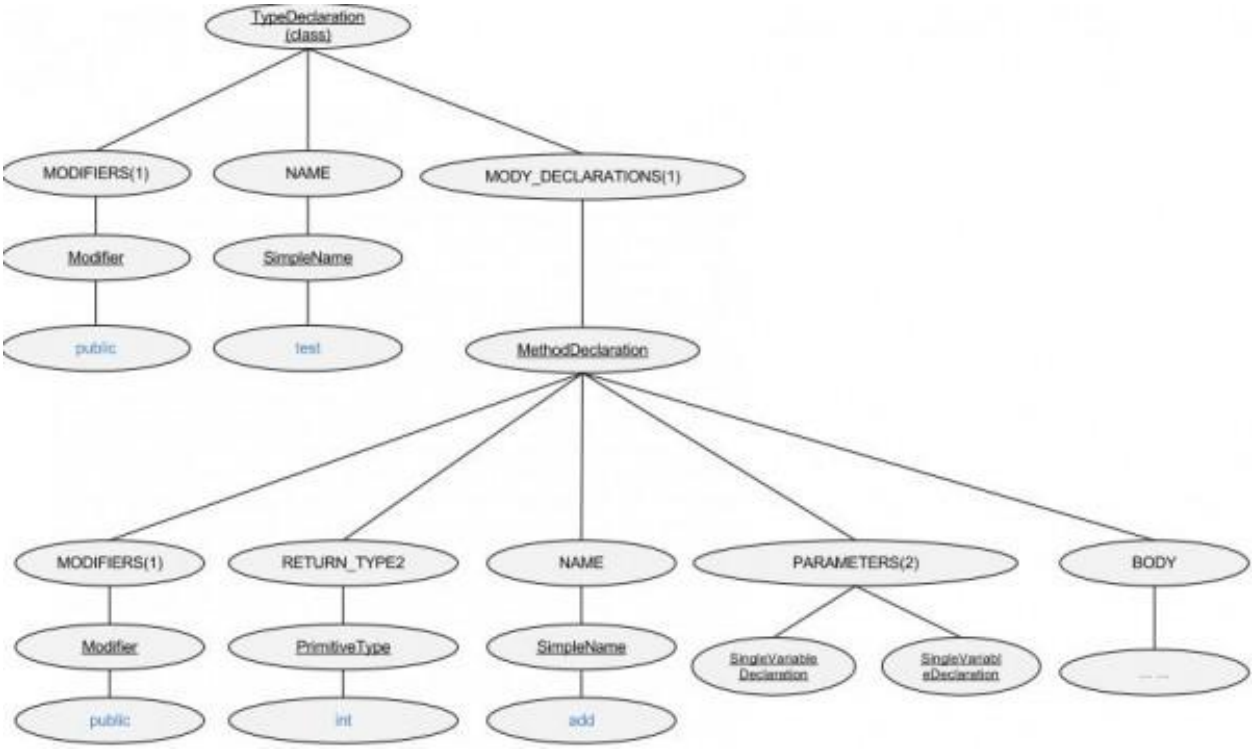
Syntax

Tree

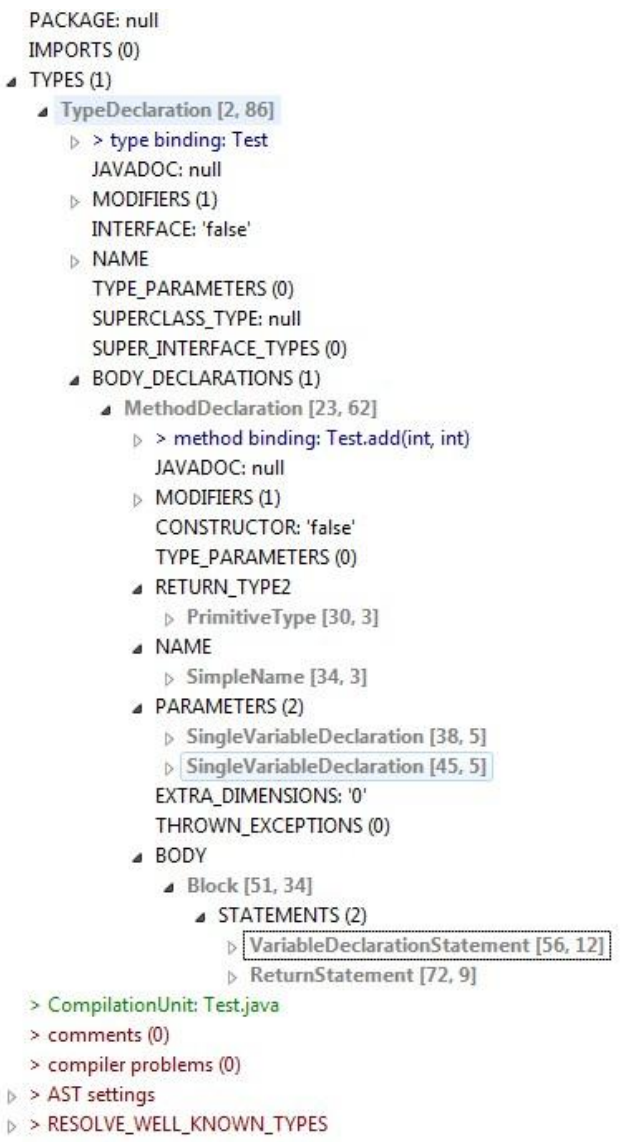
Transformacja do AST

```
public class Test {
        public int add(int a, int b){
                int c = a+b;
                return c;
        }
}
```

```java
public class Test {
        public int add(int a, int b){
                int c = a+b;
                return c;
        }
}
```



podgląd AST w IDE



Lombok

encapsulation

C#

```
//C#
public class Rectangle{
    private int width{get;set;};
    private int height{get;set;};
}
```

Plain Java

```
//Java
public class Rectangle{
    private int width;
    private int height;

    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
}
```

Single field

```
//Lombok
public class Rectangle{
    @Getter @Setter
    private int width;
    @Getter @Setter
    private int height;
}
```

All fields in class

```
//Lombok
@Getter @Setter
public class Rectangle{
    private int width;
    private int height;
}
```

constructors

With Lombok

```
@NoArgsConstructor
public static class NoArgsExample {
  @NonNull private String field;
}
```

@NoArgsConstructor
@RequiredArgsConstructor
@AllArgsConstructor

Plain Java

```
public static class NoArgsExample {
  @NonNull private String field;

  public NoArgsExample() {
  }
}
```

With Lombok

undefined type

```
public class ValExample {
  public String example() {
    val example = new ArrayList<String>();
    example.add("Hello, World!");
    val foo = example.get(0);
    return foo.toLowerCase();
  }
```

Plain Java

```
public class ValExample {
  public String example() {
    final ArrayList<String> example = new ArrayList<String>();
    example.add("Hello, World!");
    final String foo = example.get(0);
    return foo.toLowerCase();
  }
```

With Lombok

checked exceptions

```
public class SneakyThrowsExample implements Runnable {
  @SneakyThrows(UnsupportedEncodingException.class)
  public String utf8ToString(byte[] bytes) {
    return new String(bytes, "UTF-8");
  }
```

Plain Java

```
public class SneakyThrowsExample implements Runnable {
  public String utf8ToString(byte[] bytes) {
    try {
      return new String(bytes, "UTF-8");
    } catch (UnsupportedEncodingException e) {
      throw Lombok.sneakyThrow(e);
    }
  }
```

With Lombok

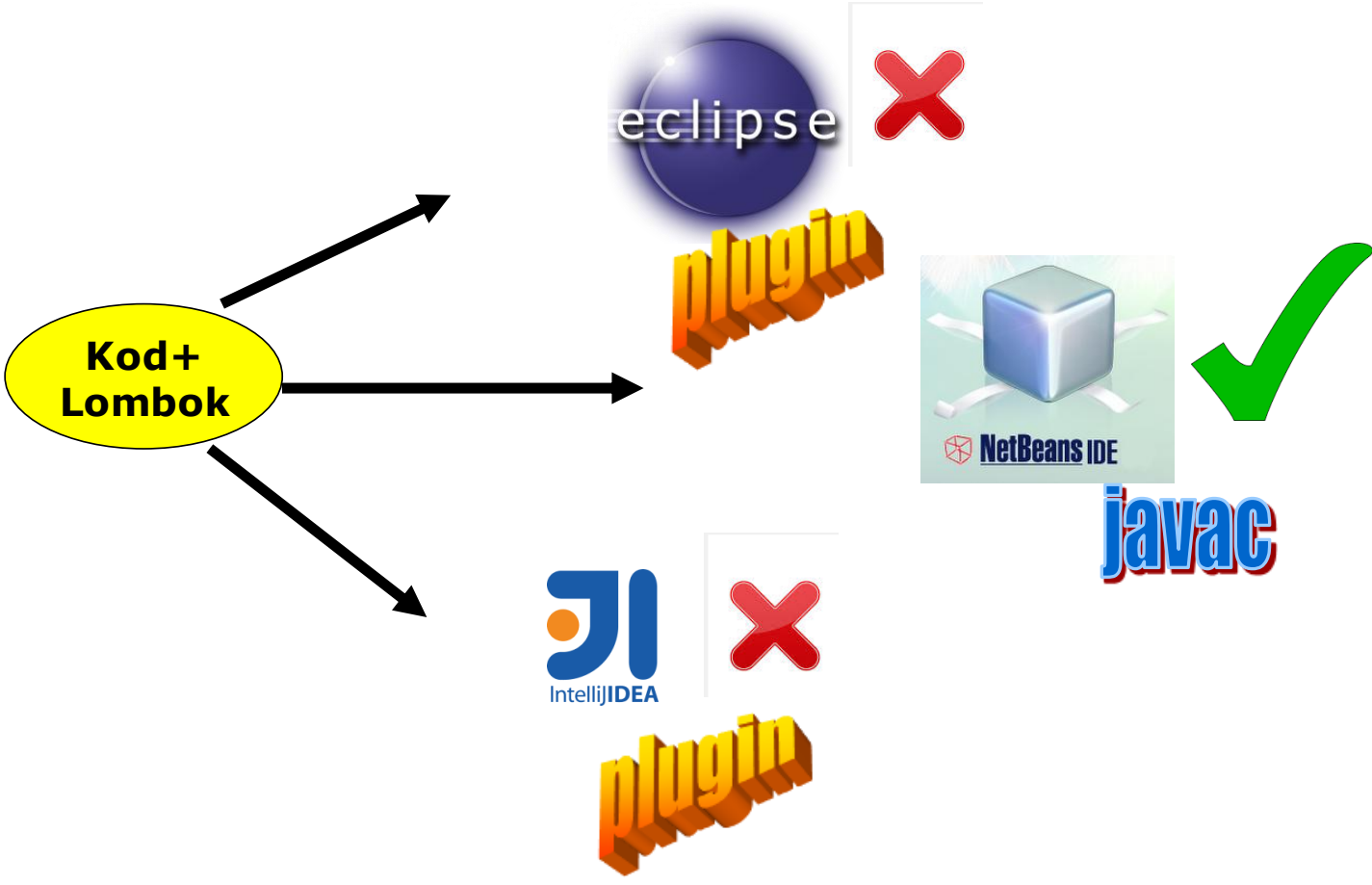

```java
public class WitherExample {
  @Wither private final int age;
  @Wither(AccessLevel.PROTECTED) @NonNull private final String name;

  public WitherExample(String name, int age) {
    if (name == null) throw new NullPointerException();
    this.name = name;
    this.age = age;
  }
}
```

Plain Java

```java
public class WitherExample {
  private final int age;
  private @NonNull final String name;

  public WitherExample(String name, int age) {
    if (name == null) throw new NullPointerException();
    this.name = name;
    this.age = age;
  }

  public WitherExample withAge(int age) {
    return this.age == age ? this : new WitherExample(age, name);
  }

  protected WitherExample withName(@NonNull String name) {
    if (name == null) throw new java.lang.NullPointerException("name");
    return this.name == name ? this : new WitherExample(age, name);
  }
}
```

Integracja z IDE

IDE - plugins


Kod+
Lombok
eclipse
plugin
NetBeans IDE
javac
IntelliJIDEA
plugin

IDE - OK

```
@Getter
@Setter
public class A {
    private Integer id;

    public static void main(String[] args) {
        A a = new A();
        a.setId(11);
        a.
    }
}
```

setId(Integer id)
getId()
id

IDE – nie OK

```
@Getter
@Setter
public class A {
    private Integer id;

    public static void main(String[] args) {
        A a = new A();
        a.setId(11);
        a.
    }
}
```

id
clone ()
equals (Object)

Lombok

Rozszerzenia

Projekty rozszerzające

- Lombok-pg
- Lombok-experimental features

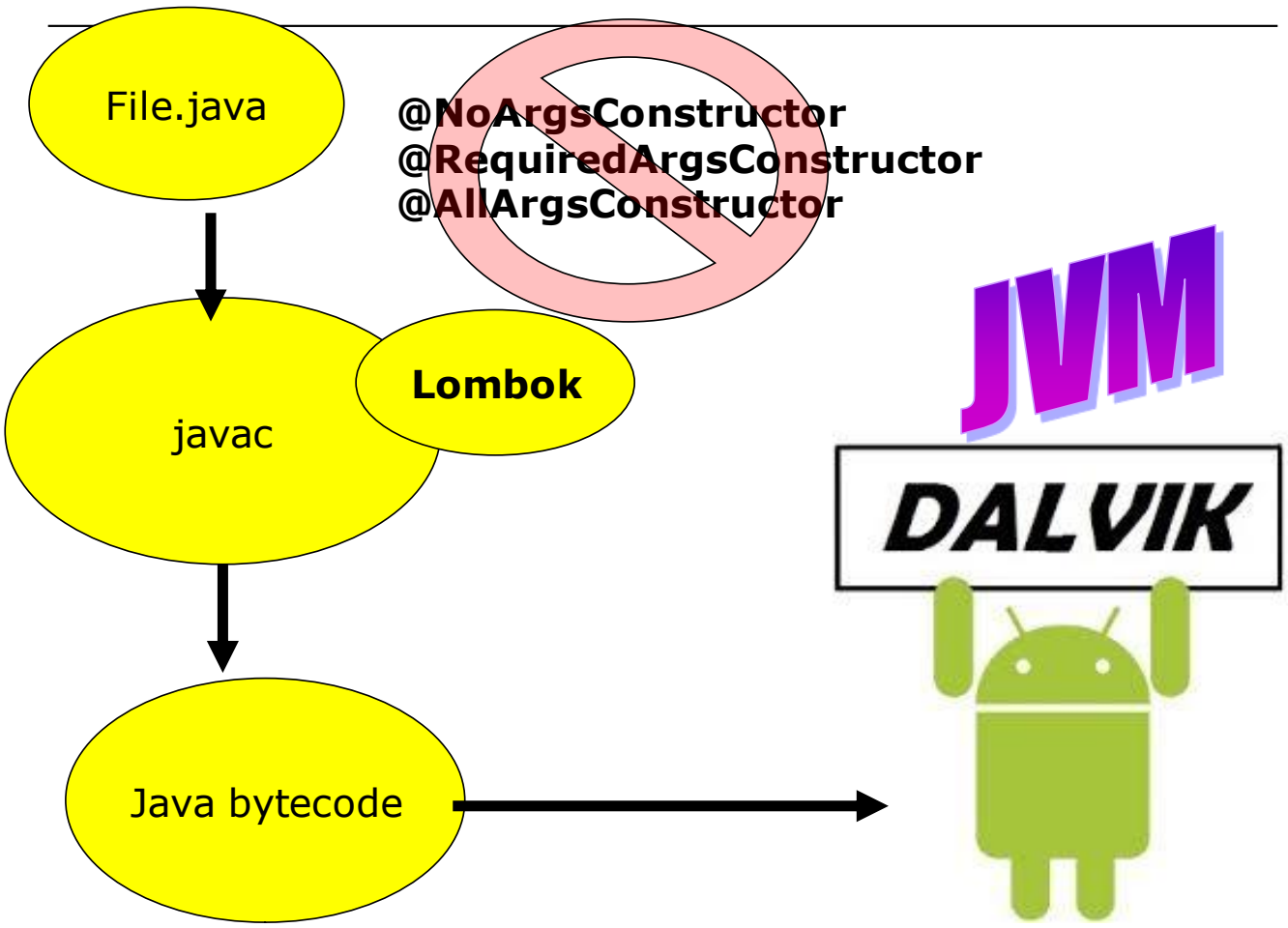
Lombok – własne adnotacje

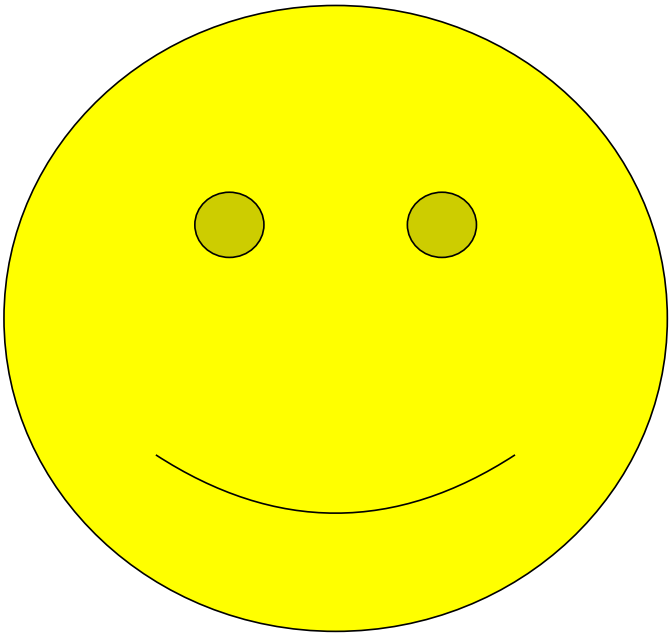
- Utworzyć adnotację
- Zaimplementować węzły AST dla kompilatora javac:

```
public abstract void handle(
        AnnotationValues<T> annotation, JCTree.JCAnnotation ast, JavacNode annotationNode);
```

- (*) Zaimplementować węzły AST dla kompilatorów pod inne IDE (IJ, Eclipse)

Lombok + Android





Dziękuję za uwagę!