Zaawansowane programowanie obiektowe i funkcyjne

Wykład 7: Programowanie generyczne, refleksje, klasy pośredniczące i adnotacje

dr inż. Marcin Luckner
mluckner@mini.pw.edu.pl

Wydział Matematyki i Nauk Informacyjnych

Wersja 1.2
4 marca 2021

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Programowanie generyczne - przypomnienie

- Programowanie generyczne pozwala pisać kod działający na różnych typach danych.
- Pozwala także ograniczać zakres dopuszczalnych typów.
- Eliminuje rzutowanie co ogranicza niebezpieczną konwersję zwężającą.
- W rezultacie otrzymujemy uniwersalny kod z kontrolą typów.

Typ surowy

- Maszyna wirtualna nie potrafi radzić sobie z typami generycznymi. Wymagana jest ich konwersja do zwykłych klas.
- Rozwiązaniem jest typ surowy, który zachowuje nazwę klasy, ale nie posiada parametrów, które są wymazywane.
- Wymazywanie polega na zastąpieniu parametrów przez typy graniczne lub `Object`

Pair<T>

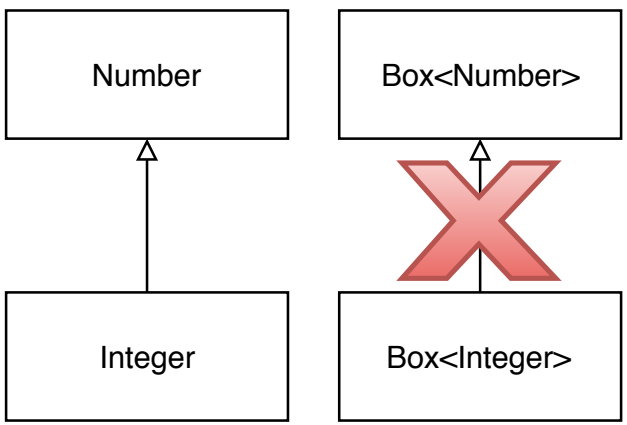
```
1  public class Pair{
2    private Object first;
3    private Object second;
4  }
```

Box<T extends Number>

```
1  public class Box{
2    private Number value;
3  }
```

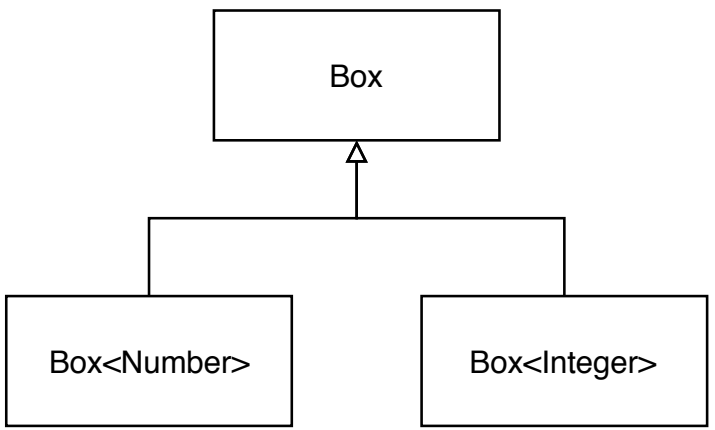
Hierarchia klas

- Klasy generyczne nie zachowują hierarchii typów, których dotyczą.



Rysunek 1: *Relacja między klasami `Number` i `Integer` nie przenosi się na klasę generującą*

- Wszystkie klasy generyczne znajdują się na tym samym poziomie hierarchii.



Rysunek 2: *Klasy `Box<Number>` i `Box<Integer>` bezpośrednio dziedziczą po typie surowym*

Ograniczenia przy tworzeniu klas generycznych

- Załóżmy, że utworzyliśmy dwie klasy `Man` i `Woman` dziedziczące po klasie `Human`.
- Korzystając z klasy generycznej `Pair<T>` możemy utworzyć obiekty `Pair<Man>` lub `Pair<Woman>`, ale nie możemy utworzyć pary mieszanej.
- Możemy utworzyć parę `Pair<Human>` i dowolnie mieszać wewnątrz niej oba typy pochodne.
- Załóżmy jednak, że urządzamy przyjęcie i wysyłamy zaproszenia do par osób korzystając z poniższej metody

```
1       public void sendInvitation(Pair<Human>)
```

- Ze względu na brak dziedziczenia nie możemy wysłać zaproszenia do istniejącej pary `Pair<Man>`

```
1  Pair<Man> buddies = new Pair<>(new Man("Sherlock Holmes"),
2    new Man("John Watson"));
```

Typy wieloznaczne

- Rozwiązaniem problemu z wysyłaniem zaproszeń jest typ wieloznaczny, który pozwala na stworzenie obiektu generycznego obsługującego różne typy parametru.
- Notacja `Pair<? extends Human>` pozwala utworzyć parę dowolnych instancji klas nadpisujących klasę `Human`.
- Odpowiednio zmodyfikowana metoda rozsyłania zaproszeń rozwiązuje sprawę.

```
1        public void sendInvitation(Pair<? extends Human>)
```

Klasy nadrzędne w typach wieloznacznych

- Typ wieloznaczny może być także zdefiniowany dla klas nadrzędnych danego typu.
- Notacja `Pair<? super Men>` pozwala utworzyć pary zawierające klasę `Man`, ale także `Human` czy `Object`.
- Możliwość ta jest wykorzystywana przez interfejs `Collection` do filtrowania obiektów po cechach ich lub ich rodziców.

```
1       default boolean removeIf(Predicate<? super E> filter)
```

- Możemy użyć jej, aby odesłać na emeryturę niektórych pracowników Scotland Yardu.

```
1  ArrayList<Man> scotlandYard = new ArrayList<>();
2
3  scotlandYard.add(new Man("Inspector G. Lestrade"));
4  scotlandYard.add(new Man("Inspector Stanley Hopkins"));
5  scotlandYard.add(new Man("Inspector Tobias Gregson"));
6  scotlandYard.add(new Man("Inspector Bradstreet"));
7
8  Predicate<Human> predicate = h->h.getAge()>60;
9  scotlandYard.removeIf(predicate);
```

Ochrona typu wielowartościowego

- Rozważmy parę `Pair<? extends Human>`. Kompilator wie, że znajdować się w niej muszą dwa obiekty nadpisujące klasę `Human`.
- Jednakże kompilator nie wie, jakie konkretne klasy mają być w niej zawarte.
- Z tego powodu obiekty z typem wielowartościowym są chronione przed zmianami.

```
1  Pair<Man> buddies = new Pair<>(new Man("Sherlock Holmes"), new
       Man("John Watson"));
2  Pair<? extends Human> newBuddies = buddies;
3  // newBuddies.setSecond(new Woman("Irene Adler")); not allowed
```

Typy wieloznaczne bez ograniczeń

- Typ wieloznaczny można stosować bez odwołań do hierarchii klas stosując zapis `Pair<?>`.
- Na pierwszy rzut oka `Pair<?>` wydaje się nie różnić od typu surowego.
- Jednakże należy pamiętać, że w przypadku typu surowego elementy pary będą rozpatrywane jako instancje klasy `Object`.
- W przypadku typu wieloznacznego klasa nie jest określona.
- W rezultacie, w pierwszym przypadku jest możliwa modyfikacja pary, w drugim tylko jej odczyt.

Pair

```
1  public static void consume(Pair p){
2    p.setFirst(new Object());
3    System.out.println(p);
4  }
```

Pair<?>

```
1  public static void consume(Pair<?> p){
2    //p.setFirst(new Object()); - not
         allowed
3    System.out.println(p);
4  }
```

Przykład wykorzystania typu bez ograniczeń i surowego

- Typ wieloznaczny bez ograniczeń może być wykorzystywany gdy chcemy zbadać istnienie instancji będących parametrami.

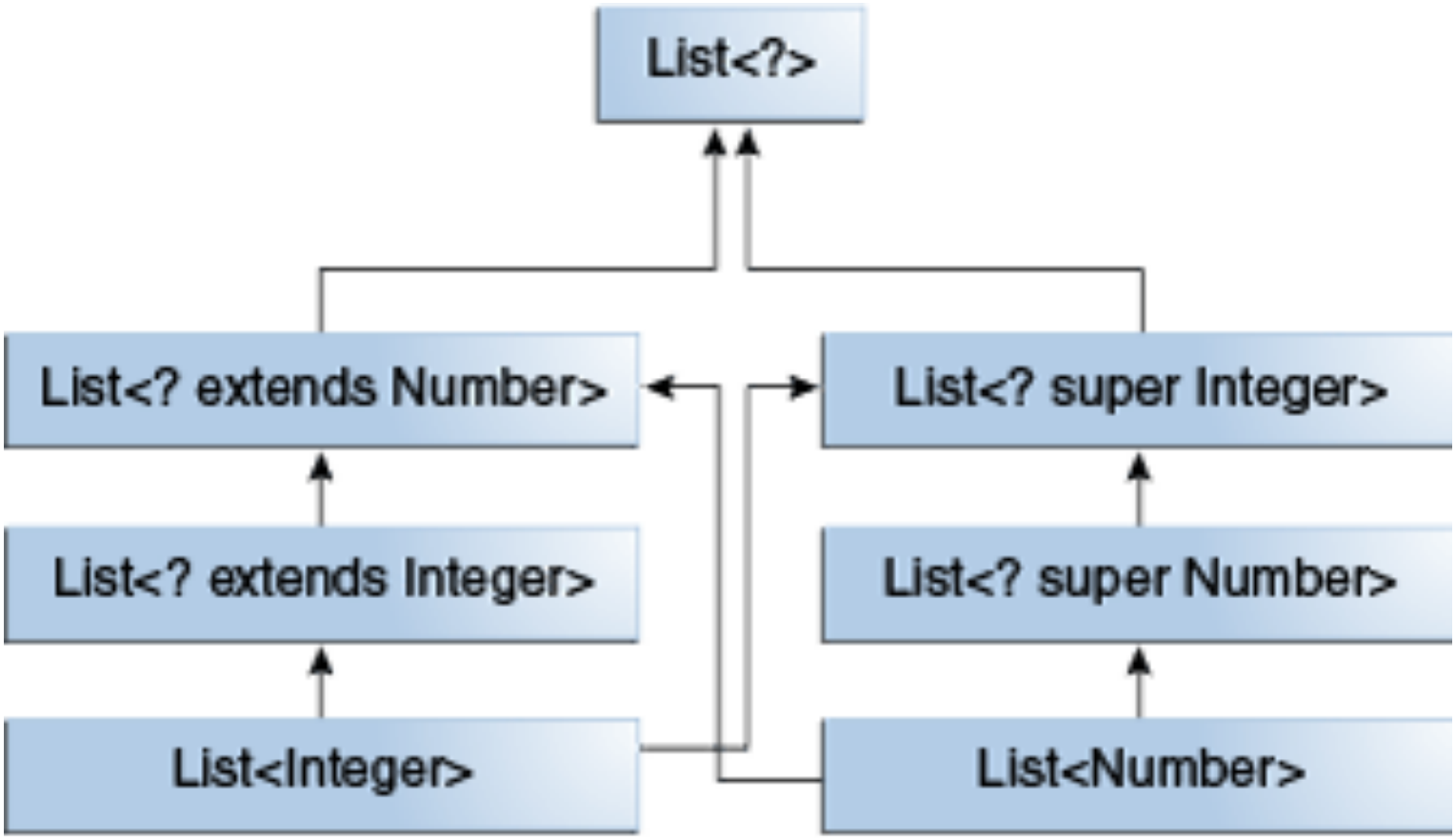
```
1  public static boolean hasNulls(Pair<?> p){
2    return p.getFirst() == null || p.getSecond() == null;
3  }
```

- Jest też bezpieczniejszy niż typ surowy i dlatego powinien być wykorzystywany zamiast niego.
- Jednakże, jeżeli chcemy użyć operatora `instanceof` musimy operować na typie surowym.

```
1  if (o instanceof Set) {
2    Set<?> m = (Set<?>) o;
3  }
```

Hierarchia typów wieloznacznych

- Typy wieloznaczne pozwalają nam na budowę hierarchii klas generycznych.



Rysunek 3: *Hierarchia klas generycznych*

Refleksje

- Refleksje umożliwiająca analizowanie obiektów w czasie działania programu.
  - Pozwalają na odczyt metadanych klasy.
  - Pozwalają na odczyt i zmianę wartości pól.
  - Pozwalają na wywołanie metod.

Klasa `Class`

- Java przechowuje informacje o typach wszystkich obiektów wykorzystywanych w działającej aplikacji.
- Można uzyskać do nich dostęp korzystając z klasy `Class`.
- Metoda `getClass()` klasy `Object` zwraca instancję dla danego obiektu.

```
1  Employee e;
2  Class cl = e.getClass();
3  System.out.println(cl.getName());
4  //Employee
```

Metody pozyskiwania instancji Class

- Obiekt Class można pozyskać bezpośrednio z obiektu.

```
1  Class cl = employee.getClass();
```

- Można też odwołać się do nazwy klasy.

```
1  Class cl = Class.forName("java.util.Date");
```

  - Nazwa powinna być pełna i zawierać informacje o pakiecie.
  - Jeżeli nie można znaleść klasy lub interfejsu o danej nazwie to rzucany jest wyjątek.

- Można wykorzystać słowo kluczowe class dostępne dla każdego typu.

```
1  Class cl1 = Random.class; // First, import
       java.util.*;.
2  Class cl2 = int.class;
3  Class cl3 = Double[].class;
```

Wczytywanie dynamiczne klas

- Podczas uruchamiania aplikacji wczytujemy klasę zawierającą metodę `main`.
  - Jej nazwa jest określona w manifeście pliku jar.
- Następnie wczytywane są wszystkie klasy, których potrzebują już wczytane klasy.
- Można w zamian wczytywać klasy dynamicznie korzystając z metody `forName` klasy `Class`.
- Technika ta jest w szczególności wykorzystywana gdy tworzymy aplikację mogącą współpracować z różnymi systemami baz danych, a która wczytuje tylko biblioteki potrzebne do połączenia się z daną bazą danych.

Tworzenie instancji

- Obiekt `Class` oferuje metodę `newInstance` do tworzenia instancji danej klasy.
- Metoda wywołuje konstruktor domyślny do utworzenia instancji.

```
1  Class cl = Class.forName("java.util.Date");
2  Object date = cl.newInstance();
```

- Możliwe jest także wywołanie konstruktora parametrycznego, używając klasę `Constructor`.

Analiza klasy

- Obiekt `Class` oferuje dostęp do składowych klasy.
- Są one opisane następującymi klasami

Field	reprezentacja pola klasy,
Method	reprezentacja metody klasy,
Constructor	reprezentacja konstruktora klasy.

- Metody `get(Fields|Methods|Constructors)` zwracają tablice publicznych elementów klasy.
- Metody `getDeclared(Fields|Methods|Constructors)` zwracają tablice wszystkich zadeklarowanych elementów klasy.
  - Nie wliczają się w to elementy klasy nadrzędnej.

Informacje o typach składowych

- Klasa `Field` ma metodę `getType`, zwracającą informacje o typie pola pod postacią obiektu klasy `Class`.
- Klasy `Method` i `Constructor` mają metodę `getParameterTypes` do odczytu typów parametrów.
- Dodatkowo klasa `Method` ma metodę `getReturnType` określającą zwracany typ.
  - Zauważmy, że metoda może być `void`. Obiekt `Class` może reprezentować tego `void` i legalne jest wywołanie `void.class`.

Informacje o modyfikatorach

- Każda z klas posiada metodę `getModifiers`.
- Metoda zwraca liczbę całkowitą określającą włączone modyfikatory składowych.
- Do jej interpretacji służy klasa `Modifier` i jej metody statyczne.
  - Metody `isPublic`, `isPrivate` lub `isFinal` informują czy dany modyfikator jest włączony.
  - Można także wydrukować modyfikatory metodą `toString`

```
1  Arrays.stream(obj.getClass().getDeclaredMethods()).forEach(
2    m-> System.out.println(Modifier.toString(m.getModifiers())+" "
3    +m.getName()+"; isPrivate:"+Modifier.isPrivate(m.getModifiers()))
4  );
```

Wynik

```
public contentEquals; isPrivate:false
private nonSyncContentEquals; isPrivate:true
public equalsIgnoreCase; isPrivate:false
```

Dostęp do danych

- Dotychczas odczytywaliśmy metadane klasy (typy, modyfikatory, nazwy).
- Jednakże mechanizm refleksji pozwala także odczytać wartości pól.
- Klasa `Field` posiada metodę `get(obj)`, która zwraca wartość pola zapisaną w obiekcie `obj`.
  - Obiekt `obj` musi należeć do klasy, z której pobrano instancję `Field`.
  - Pole `Field` musi być dostępne (nie może być prywatne).
  - Jednakże można odblokować dostęp do pola korzystając z metody `setAccessible(true)`.
- Na podobnej zasadzie działa metoda `set(obj, value)` zmieniająca wartość pola.

Analiza klasy anonimowej

- Nadpisujemy klasę `Person`, która ma jeden atrybut `name` przekazywany przez konstruktor i metodę `getName` do jego odczytu.

```
Person superhero = new Person("Clark Kent"){
  public String getNick(){
    return "Superman";
  }
  public String getName(){
    return getNick();
  }
};
```

- Czy uda nam się odczytać wartość pola `name`?

Analiza klasy anonimowej II

- Tworzymy klasę `Villain`, która ma odczytać atrybut `name` z obiektu `superhero`.
- Na początku uzyskajmy dostęp do klasy opisującej obiekt.

```
1  Class cl = superhero.getClass();
2  System.out.println(cl.getName());
3  //pl.edu.pw.mini.mluckner.aofp.lecture06.Metropolis$1
```

- Próbujemy odczytać wartości deklarowanego pola `name`.

```
1     Field field  = cl.getDeclaredField("name");
```

Wynik

java.lang.NoSuchFieldException: name

- Mimo, iż używamy metody `getDeclaredField` nie dowiemy się o polu `name`, bo jest ono deklarowane w klasie nadrzędnej.

Analiza klasy anonimowej III

- Odwołajmy się do pól klasy nadrzędnej.

```
1       Field field  =
            cl.getSuperclass().getDeclaredField("name");
```

Wynik

java.lang.IllegalAccessException: class Villain cannot access a member of class Person with modifiers ""

- Nie mamy dostępu do pola, które jest domyślnie prywatne.

Analiza klasy anonimowej - grande finale

- Dodajmy element, który osłabi zabezpieczenia.

```
1  Field field  =
       cl.getSuperclass().getDeclaredField("name");
2  useKryptonite(field);
3  System.out.println("His secret identify is "+
       field.get(superhero));
```

Wynik

His secret identify is Clark Kent

- Co kryje metoda useKryptonite?

```
1  private void useKryptonite(Field f){
2    f.setAccessible(true);
3  }
```

Dostęp do metod

- Java nie powtórzyła rozwiązania z C czyli przekazywania wskaźników do metod.
- Uznano, że jest to mechanizm niebezpieczny i zastąpiono go interfejsami.
- Jednakże można przekazywać obiekty klasy `Method` do wywoływania metod.
- Metoda `invoke` klasy `Method` przekazuje parametry do jej wywołania.

```
1  public Object invoke(Object implicitParameter,
       Object[] explicitParameters)
```

  - Pierwszy argument to wskazanie obiektu,
  - Kolejne argumenty to parametry metody,
  - W przypadku metody statycznej pierwszy argument jest ignorowany.

Używanie dostępu do metod

- Dostęp do metod można uzyskać w sposób analogiczny jak dostęp do pól.

```
1  Method m1 = Employee.class.getMethod("getName");
```

- Jednakże ze względu na możliwość duplikacji nazw należy czasami przekazać informację o klasie parametrów.

```
1  Method m2 = Math.class.getMethod("sqrt", double.class)
```

Używanie dostępu do metod - krytyka

- Klasa `Method` działa jak wskaźnik do metody, którą możemy uruchomić metodą `invoke`.
- Jednakże ma ona kilka poważnych ograniczeń.
  - Metoda jest podatna na wywołanie ze złymi parametrami co spowoduje wyjątek.
  - Parametry i zwracany obiekt są klasy `Object`. Nie ma możliwości sprawdzania kodu, jest za to rzutowanie.
- Z powyższych powodów lepiej używać interfejsów i wyrażeń lambda.

Refleksja, a programowanie generyczne

- Klasa `Class` jest tak naprawdę klasą generyczną.

```
1  Class<Math> classMath = Math.class;
```

- Dzięki temu, wykorzystując `newInstance` możemy tworzyć obiekty generyczne dla danego typu.

```
1  public static <T> Pair<T> makePair(Class<T> c)
2  throws InstantiationException,
3  IllegalAccessException{
4    return new Pair<T>(c.newInstance(), c.newInstance());
5  }
```

- Teraz, w bezpieczny sposób, możemy stworzyć parę określonego typu.

```
1  Pair<Employee> pair = makePair(Employee.class)
```

Klasy pośredniczące

- Klasy pośredniczące (ang. *Proxy*) są stosowane do tworzenia nowych implementacji interfejsów w trakcie działania programu.
- Są potrzebne tylko w rzadkich wypadkach, kiedy przed kompilacją nie wiadomo jeszcze jakie interfejsy dana klasa ma implementować.
- Są pomocne przy automatycznym dodawaniu prostej funkcjonalności do kodu.

Klasy pośredniczące a refleksje

- Refleksje pozwalają na dynamiczne tworzenie instancji klasy.
- Jednakże nie można przy ich pomocy utworzyć implementacji interfejsu poprzez utworzenie nowej klasy.
- Klasa pośrednicząca może tworzyć nowe klasy w trakcie działania programu implementując określone przez programistę interfejsy.

Budowa klasy pośredniczącej

- Klasa pośrednicząca zawiera:
  - wszystkie metody wymagane przez określone interfejsy.
  - wszystkie metody zdefiniowane w klasie `Object`.
- Ponieważ nie można w czasie działania programu napisać kodu nowej klasy dostarczamy zamiast tego obiekt obsługujący wywołanie (ang. *invocation handler*).
- Może być to obiekt dowolnej klasy implementującej interfejs `InvocationHandler` składający się z jednej metody

```
1  Object invoke(Object proxy, Method method, Object[] args)
```

- Wywołując metodę na rzecz obiektu klasy pośredniczącej przekazujemy wywołanie do metody `invoke` przekazując informacje o metodzie i oryginalnych argumentach.

Tworzenie obiektów pośredniczących

- Obiekty klasy pośredniczącej tworzy metoda statyczna `newProxyInstance` klasy `Proxy`.
- Metoda przyjmuje trzy parametry:
  - Mechanizm ładowania klas. Parametr `null` oznacza zastosowanie domyślnego mechanizmu ładowania.
  - Tablica obiektów klasy `Class` określający interfejsy, które mają być implementowane.
  - Obiekt obsługujący wywołanie.

Właściwości klas pośredniczących

- Każda klasa pośrednicząca rozszerza klasę `Proxy` i ma tylko jedno pole obiektowe – obiekt obsługujący wywołanie.
- Wszystkie dodatkowe dane potrzebne do przeprowadzenia działań obiektu pośredniczącego muszą być zapisane w obiekcie obsługi wywołania.
- Każdy mechanizm ładowania klasy i uporządkowany zestaw interfejsów na tylko jedną klasę pośredniczącą, którą można doczytać poprzez `getProxyClass`.
- Klasy pośredniczące są zawsze publiczne i finalne.
- Wszystkie implementowane interfejsy muszą być publiczne lub należeć do tego samego pakietu. W tym drugim przypadku klasa również będzie należeć do pakietu.
- Możemy sprawdzić czy obiekt reprezentuje klasę pośredniczącą wywołując `isProxyClass` klasy `Proxy`.

Pozyskiwanie informacji z proxy I

- Utwórzmy kod, który będzie informował o fakcie i parametrach wywołania metody [Horstmann, 2016].

```
1  static class TraceHandler implements InvocationHandler {
2
3    private Object target;
4
5    public TraceHandler(Object t) {
6      target = t;
7    }
8    public Object invoke(Object proxy, Method m, Object[]
         args) throws Throwable {
9      System.out.print(target);
10     //informacje o metodzie
11     System.out.print("." + m.getName() + "(");
12     //informacje o argumentach
13     if (args != null) {
14       for (int i = 0; i < args.length; i++) {
15         System.out.print(args[i]);
16         if (i < args.length - 1) System.out.print(", ");
17       }
18     }
19     System.out.println(")");
20     return m.invoke(target, args); //wywołanie metody
21   }
22 }
```

Pozyskiwanie informacji z proxy II

- Użyjemy proxy, aby pozyskać informacje o przebiegu wyszukiwania binarnego.

```
1  Object[] elements = new Object[1000];
2  //inicjacja tabeli obiektami proxy
3  for (int i = 0; i < elements.length; i++) {
4    Integer value = i + 1;
5    InvocationHandler handler = new TraceHandler(value);
6    Object proxy = Proxy.newProxyInstance(null, new Class[] {
         Comparable.class } , handler);
7    elements[i] = proxy;
8  }
9  //wyszukiwanie losowego elementu
10 Integer key = new Random().nextInt(elements.length) + 1;
11 int result = Arrays.binarySearch(elements, key);
12 if (result >= 0){
13   System.out.println(elements[result]);
14 }
```

Pozyskiwanie informacji z proxy III

- Wynik działania algorytmu dla losowej wartości (740).

```
500.compareTo(740)
750.compareTo(740)
625.compareTo(740)
687.compareTo(740)
718.compareTo(740)
734.compareTo(740)
742.compareTo(740)
738.compareTo(740)
740.compareTo(740)
740.toString()
740
```

Adnotacje

- Adnotacje nie wpływają bezpośrednio na program.
- Mogą być rozpoznawane przez narzędzia i biblioteki, które wpłyną na działanie programu.
- Mogą być odczytane z plików źródłowych, plików klas lub podczas wykonywania programu.

Powiązanie adnotacji z refleksjami

- Adnotacje umieszcza się przy elementach składowych klas np. polach czy metodach.
- Mechanizm refleksji, który pozwala analizować te elementy, pozwala również na odczyt adnotacji.
- Dzięki temu możemy operować na adnotacjach w trakcie działania programu.

Tworzenie adnotacji

- Adnotacje tworzy się przy pomocy składni zbliżonej do interfejsu i adnotacji `@interface`.
- Każda deklaracja metody tworzy element adnotacji.
- Metody nie mogą mieć parametrów, ani rzucać wyjątków.
- Zwracane typy są ograniczone do podstawowych, `String`, `Class`, wyliczeniowych i tablic tych typów.
- Metody mogą mieć wartości domyślne definiowane słowem kluczowym `default`.
- Zakres działania adnotacji jest determinowany przez meta-adnotacje.

Meta-adnotacja @Target

- `@Target` określa do czego możemy podłączyć daną adnotację.
  - Cel adnotacji określa wartość typu `ElementType`.

```
1  public enum ElementType {
2      TYPE,
3      FIELD,
4      METHOD,
5      PARAMETER,
6      CONSTRUCTOR,
7      LOCAL_VARIABLE,
8      ANNOTATION_TYPE,
9      PACKAGE,
10     TYPE_PARAMETER,
11     TYPE_USE,
12     MODULE
13 }
```

  - Brak parametru oznacza dowolny cel.

Meta-adnotacja @Retention

- @Retention określa sposób zachowywania adnotacji.
  - Jest to wartość typu wyliczeniowego RetentionPolicy.

```
1  public enum RetentionPolicy {
2      SOURCE,
3      CLASS,
4      RUNTIME
5  }
```

  - RUNTIME pozwala używać adnotacji w czasie wykonywania programu.

Przykładowe adnotacje

- Adnotacja z jednym parametrem.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  @interface BugCount{
4    int value();
5  }
```

- Adnotacja o wielu parametrach.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  @interface ToDo{
4    PRIORITY priority() default PRIORITY.LOW;
5    String[] description();
6    Class<?> source() default AnnotationsDemo.class;
7  }
```

- Adnotacja korzysta z trybu wyliczeniowego.

```
1  public enum PRIORITY {LOW, MEDIUM, HIGH}
```

Używanie adnotacji parametrycznych

- Adnotacje z jednym parametrem mogą być używane w uproszonej formie.

```
1  @BugCount(1)
2  private static void showToDo(Method m){
3  //Do something
4  }
```

- Adnotacje o wielu parametrach muszą być wywoływane z podaniem ich nazw.

```
1  @ToDo(priority = PRIORITY.MEDIUM,description = "Replace + with
       stringBuilder." )
2  public static void main(String[] args) {
3  //Do something
4  }
```

Sprawdzanie adnotacji I

- Możemy zliczyć liczbę odnotowanych bugów w kodzie programu.

```
1  public static void main(String[] args) {
2    int bugs =
         Arrays.stream(AnnotationsDemo.class.getDeclaredMethods())
3        .mapToInt(AnnotationsDemo::calculateBugs).sum();
4    System.out.println("Bugs in project: "+ bugs);
5  }
6  private static int  calculateBugs(Method m){
7    return m.isAnnotationPresent(BugCount.class)?
8      m.getAnnotation(BugCount.class).value():0;
9  }
```

Sprawdzanie adnotacji II

- Możemy stworzyć raport o zadaniach do zrobienia.

```
1  public static void main(String[] args) {
2    for (Method method : AnnotationsDemo.class.getDeclaredMethods()) {
3              showToDo(method);
4    }
5  }
6  private static void showToDo(Method m){
7    if(m.isAnnotationPresent(ToDo.class)){
8      System.out.println("Source: "
           +m.getAnnotation(ToDo.class).source()+"."+m.getName());
9      System.out.println("\tpriority: "
           +m.getAnnotation(ToDo.class).priority());
10     System.out.println("\tdescription:\n\t " +
           Arrays.stream(m.getAnnotation(ToDo.class).description())
11       .collect(Collectors.joining("\n\t ")));
12   }
13 }
```

Sprawdzanie adnotacji - wynik

```
Bugs in project: 3
-------------------
Source: class AnnotationsDemo.main
priority: MEDIUM
description:
 Replace + with stringBuilder.
Source: class AnnotationsDemo.showToDo
priority: HIGH
description:
 Replace + with stringBuilder.
 Reduce m.getAnnotation calls.
```

Automatyczne generowanie kodu I

- Adnotacje w połączeniu z referencjami mogą być użyte do automatycznego generowania kodu.
- Omówimy pokrótce kroki potrzebne do automatycznego przypisywania akcji do przycisków.
- Pełny kod i omówienie przykładu znajduje się w pozycji [Horstmann and Cornell, 2017].

Automatyczne generowanie kodu II

- Naszym celem jest połączenie obiektu nasłuchującego ze zródłem zdarzenia.

```
1    myButton.addActionListener(()-> doSomething());
```

- Możemy połączyć źródło z obiektem poprzez adnotację.

```
1  @ActionListenerFor(source="myButton")
2    void doSomething()
```

Automatyczne generowanie kodu III

- Podczas uruchomienia aplikacji odnajdziemy wszystkie metody z naszą adnotacją.

```
1  Class<?> cl = obj.getClass();
2  for (Method m : cl.getDeclaredMethods()){
3    ActionListenerFor a =
         m.getAnnotation(ActionListenerFor.class);
4    if (a != null){
5      Field f = cl.getDeclaredField(a.source());
6      f.setAccessible(true);
7      addListener(f.get(obj), obj, m);
8    }
9  }
```

- Metoda addListener będzie łączyć odnalezione pole z `f` z metodą `m` wewnątrz obiektu `obj`.

Automatyczne generowanie kodu IV

```
1  public static void addListener
2    (Object source, final Object param, final Method m)
3    throws ReflectiveOperationException{
4
5    InvocationHandler handler = new InvocationHandler(){
6      public Object invoke(Object proxy, Method mm,Object[]
           args) throws Throwable{
7        return m.invoke(param);
8      }
9    };
10   Object listener = Proxy.newProxyInstance(null, new
       Class[]{java.awt.event.ActionListener.class},
       handler);
11   Method adder =
       source.getClass().getMethod("addActionListener",
12     ActionListener.class);
13   adder.invoke(source,listener);
```

- Połączenie mechanizmów refleksji, proxy i adnotacji pozwoliło połączyć źródło zdarzenia z klasą nadsłuchującą.

Bibliografia

[Horstmann, 2016] Horstmann, C. S. (2016).
Java. Podstawy.
Helion.

[Horstmann and Cornell, 2017] Horstmann, C. S. and Cornell, G. (2017).
Java. Techniki zaawansowane.
Helion.