Wykład 11

Refleksje

Klasy

Atrybuty i metody

Dynamiczne klasy proxy

Refleksje

`Programowanie Refleksyjne` to mechanizm, który **umożliwia odczyt i modyfikację informacji o klasach w trakcie wykonywania programu**. Dzięki refleksji możesz na przykład odczytać nazwy pól i metod zapisanych w klasie, a nawet odwołać się do prywatnych pól klasy.

> Polega ono na dynamicznym korzystaniu ze struktur języka programowania, które nie musiały być zdeterminowane w momencie tworzenia oprogramowania.

Najważniejsze klasy języka Java, które umożliwiają programowanie refeksyjne to:

- `Class`
- `Field`
- `Method`
- `Array`
- `Constructor`
  Są one zgrupowane w pakietach ***java.lang*** i ***java.lang.reflect***.

Każdy obiekt w Javie jest instancją klasy Object.

Klasy

Klasa w Javie jest szablonem, który definiuje postać obiektu. Określa ona zarówno dane obiektu, jak i kod, który działa na tych danych.

Obiekty są instancjami klasy. Innymi słowy, klasa jest zestawem planów określających sposób konstrukcji obiektu. Fizyczna reprezentacja klasy w pamięci komputera powstaje dopiero na skutek utworzenia obiektu tej klasy.

Każdy typ (obiektowy, prymitywny, tablicowy itp.) jest reprezentowany przez instację klasy
Class, którą uzyskujemy za pomocą `getClass()`.

`getClass()` — jest używana do uzyskania metadanych o klasie obiektu, z którym pracujesz. Zwraca ona instancję klasy *Class*, która zawiera informacje o klasie, z której została wywołana.

```
import java.util.HashSet;
import java.util.Set;

enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

public class ReflectionExample {
    public static void main(String[] args){
        Class c;
        c = "foo".getClass();
        System.out.println(c.getName()); // wypisuje java.lang.String

        c = System.out.getClass();
        System.out.println(c.getName()); // wypisuje java.io.PrintStream

        c = Day.SUNDAY.getClass();
        System.out.println(c.getName()); // wypisuje Day

        byte[] bytes = new byte[1024];
        c = bytes.getClass();
        System.out.println(c.getName()); // wypisuje [B

        Set<String> s = new HashSet<String>();
        c = s.getClass();
        System.out.println(c.getName()); // wypisuje java.util.HashSet
    }
}
```

`[B` — Wypisuje "*[*" bo jest to tablica a "*B*" bo to tablica bajtów. gdby była to tablica Stringów: "*[Ljava.lang.String*" — "L" oznacza, że elementy tablicy są obiektami, a nie typami prostymi

Jeśli nie mamy obiektu (instancji klasy) możemy użyć atrybutu *.class*:

```
c = java.io.PrintStream.class; // java.io.PrintStream

c = int[][][].class; // [[[I

c = boolean.class; // boolean
```

Jest to szczególnie istotne w przypadku typów prymitywnych (Przecież nie można na nim wywołać *getClass()* — nie jest obiektem!)

Czyli:

```
boolean b;
Class c = b.getClass(); // compile-time error
```

Class.forName()

Drugim sposobem na uzyskanie tego obiektu *Class* jest zrobienie odwrotnej rzeczy, czyli uzyskanie go przez znajomość nazwy klasy.
Wcześniej robiliśmy to posiadając obiekt lub korzystając z atrybutu `.class`.

Obiekt Class można otrzymać znając jego nazwę:

```
Class cMyClass = Class.forName("pl.edu.uj.fais.java.Klasa");

Class cDoubleArray = Class.forName("[D");

Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

W przypadku podania niepoprawnej nazwy klasy zwracany jestwyjątek *ClassNotFoundException*.

Co możemy robić z taką klasą?

Wybrane metody klasy Class:

- `static Class<?> forName(String className)` — Zwraca obiekt *Class* dla podanej nazwy klasy
- `Constructor<T> getConstructor(Class<?>... parameterTypes)` — Zwraca konstruktor klasy dla podanych typów parametrów
- `Constructor<?>[] getConstructors()` — Zwraca wszystkie publiczne konstruktory klasy
- `Field getField(String name)` — Zwraca publiczne pole klasy o podanej nazwie
- `Field[] getFields()` — Zwraca wszystkie publiczne pola klasy
- `Class<?>[] getInterfaces()` — Zwraca wszystkie interfejsy, które implementuje klasa
- `Method getMethod(String name, Class<?>... parameterTypes)` — Zwraca publiczną metodę klasy o podanej nazwie i typach parametrów
- `Method[] getMethods()` — Zwraca wszystkie publiczne metody klasy
- `int getModifiers()` — Zwraca modyfikatory klasy jako liczbę całkowitą
- `Class<? super T> getSuperclass()` — Zwraca nadklasę klasy
- `TypeVariable<Class<T>>[] getTypeParameters()` — Zwraca informacje o typach generycznych klasy
- `boolean isArray()` — Sprawdza, czy klasa reprezentuje tablicę
- `boolean isInterface()` — Sprawdza, czy klasa reprezentuje interfejs
- `boolean isPrimitive()` — Sprawdza, czy klasa reprezentuje typ prymitywny
- `T newInstance()` — Tworzy nową instancję klasy

> `Class` jest typem generycznym, bo zwraca klasę dla określonego typu.

Jak utworzyć instancję klasy nie używając sobie słowa kluczowego `new` ?

Za pomocą metody `newInstance()` — jest to opcja tylko wtedy jeśli metodę chcemy tworzyć przy użyciu konstruktora bezparametrowego!

Przykład:

```
class C1{}
class C2 extends C1{}
...
C1 o1 = new C1(); // rownowazne C1.class.newInstance()
C2 o2 = new C2();

o1.getClass().isAssignableFrom(o2.getClass()); // true
o2.getClass().isAssignableFrom(o1.getClass()); // false
o1.getClass().isInstance(o2); // true
o2.getClass().isInstance(o1); // false
...
```

Atrybuty

Typ `Field` w Javie **reprezentuje pole w klasie lub interfejsie**. Może to być pole statyczne (klasowe) lub pole instancji. Typ *Field* dostarcza informacji o polu oraz umożliwia dynamiczny dostęp do niego.

> Pole może być typu prymitywnego lub referencyjnego.

Atrybut jest reprezentowany przez instancję klasy *Field*. Wybrane metody:

- `Object get(Object obj)` — zwraca wartość atrybutu w obiekcie *obj*
- `int getInt(Object obj)` — zwraca wartość atrybutu typu *int*
- `int getModifiers()` — modyfikatory dostępu: *public*, *private*, ...
- `Class<?> getType()` — klasa reprezentująca typ atrybutu
- `void set(Object obj, Object value)` — ustawia wartosć atrybutu w obiekcie *obj*
- `void setInt(Object obj, int i)` — ustawia wartosć atrybutu typu *int*

Przykład:

```
import java.lang.reflect.Field;
public class FieldExample {
    public static String s;
    public int i;

    public static void main(String[] args) throws Exception{
        FieldExample fex = new FieldExample();
        Field f;
        f = FieldExample.class.getField("s");
        f.get(null); // zwróci null bo s nie zainicjowane
        f.set(null, "Ala"); // Równoważne z FieldExamle.s = "Ala"

        f = fex.getClass().getField("i");
        f.set(fex, 10); // fex.i = 10
    }
}
```

`f.get(null)` — dlaczego tutaj jest *null*?
Dlatego, że tutaj byłby obiekt, którego wartość atrybutu chcemy pobrać. Ten atrybut jest statyczny, więc on nie jest przyporządkowany żadnemu obiektowi, a klasie.

można powiedzieć, że dostaliśmy bardziej okrężny sposób na zrobienie czegoś prostego. A jakie sa plusy tego zastosowania?

- nie musimy hardcodować przypisywanej wartości (w tym przypadku "*Ala*")
- nie musimy hardcodować zmiennej do której wartość przypisujemy (w tym przypadku "*s*") — w tym przypadku też jest zahardcodowane, ale można to pobrać z zewnątrz, w trakcie działania programu

Metody

Klasa `Method` w Javie reprezentuje pojedynczą metodę w klasie lub interfejsie. Może być metodą klasy lub metodą instancji (w tym metodą abstrakcyjną).

Wybrane metody:

- `Class<?>[] getExceptionTypes()` — zwraca clasy reprezentujące zadeklarowane, zwracane wyjątki
- `int getModifiers()` — *private*, *public*, *static*, ...
- `Object invoke(Object obj, Object... args)` — wywołuje metodę na rzecz obiektu *obj*. Argumenty wywołania znajdują się w tablicy *args*. Watrość zwracana jest wynikiem działania wywołanej metody.
- `boolean isVarArgs()` — czy metoda ma nieokreśloną liczbę argumentów

Przykład:

```
...
Method m = Class.forName("MyClass").getDeclaredMethod(
                                            "example3", null);
m.invoke(null, null);
...
```

Metoda *invoke* może zwrócić kilka rodzajów wyjątków związanych z dostępem do metody i zgodnością typów argumentów.
Wady wynikające z używania `invoke`:

- brak kontroli (w trakcie kompilacji) typów przekazywanych parametrów,
- ograniczenie obsługi wyjątków do *Throwable*.

Rozwiązaniem tych problemów może być tzw. ***Dynamic Proxy Class***.

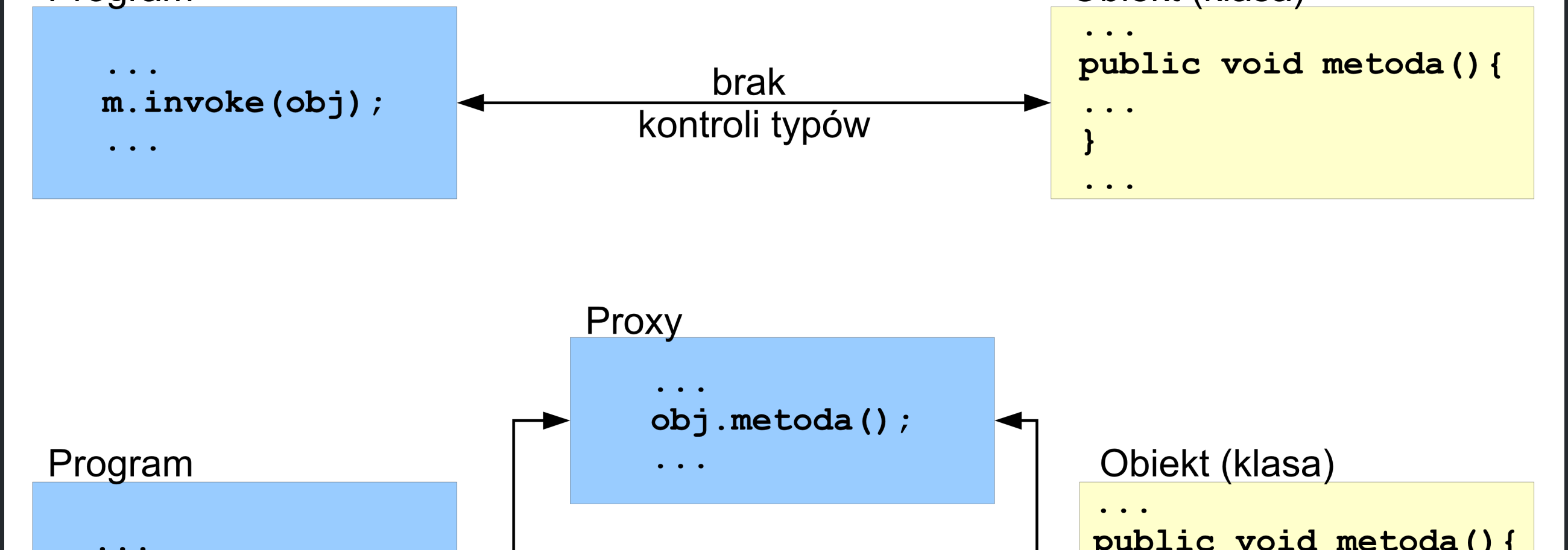
Dynamic Proxy

`Dynamic Proxy Class` w Javie to klasa, która implementuje listę interfejsów **określonych w czasie wykonywania**, tak że **wywołanie metody przez jeden z interfejsów na instancji klasy będzie kodowane i przekierowane do innego obiektu** przez jednolity interfejs.

Jak działa?

`Dynamic Proxy` polega na tym, że będzie jakiś obiekt, na którym będzie wywoływana ta metoda, przy czym ten obiekt to pośrednik. On nie będzie wykonywał tej metody, tylko on będzie prosił obiekt docelowy o wykonanie tej metody.

> Obiekt pośredniczący będzie w naszym programie tworzony dynamicznie oraz będzie się dostosowywał do obiektu którego będzie pośrednikiem.



Jak to zrobić?

`Klasa Proxy` udostępnia metody statyczne służące do tworzenia tzw. **dynamicznych klas proxy** oraz ich instancji.
Utworzenie proxy dla określonego interfejsu (np. MyInterface):

```
InvocationHandler handler = new MyInvocationHandler(...);

Class proxyClass = Proxy.getProxyClass(
                        MyInterface.class.getClassLoader(),
                        new Class[] { MyInterface.class });

MyInterface mi = (MyInterface) proxyClass.getConstructor(
        new Class[] { InvocationHandler.class }).newInstance(
                                     new Object[] { handler });
```

`InvocationHandler handler` — obiekt, który będzie realizował refleksyjne wywołania metod.

lub alternatywnie:

```
MyInterface mi = (MyInterface) Proxy.newProxyInstance(
            MyInterface.class.getClassLoader(), // loader dla zasobów
            new Class[] { MyInterface.class }, // tablica interfejsów
            handler); // obiekt do którego będą przekazywane
                  // wywołania
```

> Stworzono obiekt *mi*, który "*z zewnątrz*" wygląda jak klasa
> implementująca *MyInterface*, natomiast obsługa metod będzie
> w rzeczywistości realizowana przez ***handler***.

Obiekt handler nie musi implementować *MyInterface*!
Musi za to implementować interfejs *InvocationHandler*, czyli metodę:

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

Jak to działa?
Wszystkie wywołania metod na obiekcie *mi* są
przekierowane do metody `invoke` obiektu ***handler***, przy czym
pierwszym parametrem jest obiekt proxy.

Przykład:

```
interface Pisarz {
    public void pisz(String s);
}
```

```
class PisarzImpl implements Pisarz {
...
    public void pisz(String s){
        ...
    }
    ...
}
```

Chcemy wywołać na obiekcie metodę "*pisz*" w sposób **reflekcyjny**:

```
...
Pisarz obj = new PisarzImpl();

Method m = obj.getClass().getMethod(
                                    "pisz",
                                    new Class[] {String.class});

m.invoke(obj, new Object[]{"hello world"});
...
```

Co sie dzieje w **fioletowym kodzie**?

1. Tworzymy sobie nową instancję klasy `PisarzImpl` i przypisujemy pod obiekt `obj`
2. Pobieramy metodę `pisz` (która jako argumenty przyjmuje *String*) i przypisujemy do `m`
3. Wywołujemy metodę `m` z argumentem *String "hello world"* na obiekcie `obj` (używając do tego metody `invoke` na metodzie `m`)

Chcemy wywołać na obiekcie metodę "*pisz*" korzystając z **Dynamic Proxy**:

```
...
Pisarz p = (Pisarz) Proxy.newProxyInstance(
                Pisarz.class.getClassLoader(),
                new Class[] { Pisarz.class },
                new MyHandler());
p.pisz("hello world");
...
```

```
class MyHandler{
    private Pisarz obj = new PisarzImpl();

    public static Object invoke(Object proxy,
                Method m, Object[] args){
        return m.invoke(obj, args);
    }
}
```

Co sie dzieje w **zielonym kodzie**?

1. tworzony jest **obiekt proxy** dla interfejsu `Pisarz` za pomocą metody `Proxy.newProxyInstance()` — Ta metoda przyjmuje trzy argumenty:
   - `ClassLoader`, który ma **załadować klasę *proxy***.
   - Tablicę **interfejsów**, które ma **implementować klasa *proxy***.(chcemy aby PisarzImpl implementował Pisarza)
   - Obiekt `InvocationHandler`, który definiuje, co ma się stać, gdy metoda jest wywoływana na obiekcie *proxy* (W tym przypadku, *MyHandler* jest klasą *InvocationHandler*, która przekierowuje wszystkie wywołania metod na obiekt *PisarzImpl* — w środku działamy jakbyśmy robili to *refleksyjnie*).
2. wywołanie `p.pisz("hello world")` — wywołanie to jest przekierowywane do metody *invoke()* w *MyHandler*, która z kolei wywołuje metodę *pisz()* na obiekcie *PisarzImpl*.

> W ten sposób, możesz kontrolować, co się dzieje, gdy metoda jest wywołana na obiekcie Pisarz, dodając dodatkową logikę przed lub po wywołaniu metody na prawdziwym obiekcie. To jest kluczowa idea wzorca *Proxy*.

W tym rozwiązaniu wszystko jest sprawdzane przez kompilator, więc nawet jeśli popełnimy literówkę, to kompilator nam to powie.

> Obiekt `proxy` może byc stworzony dla kilku interfejsów.