

REFLEKSJE

ZAGADNIENIA:

- Klasy,
- Atrybuty i metody,
- Dynamiczne klasy proxy.

MATERIAŁY:

<http://docs.oracle.com/javase/tutorial/reflect/>



WPROWADZENIE

Programowanie refleksyjne polega na dynamicznym korzystaniu ze struktur języka programowania, które nie musiały być zdeterminowane w momencie tworzenia oprogramowania.

Najważniejsze klasy języka Java, które umożliwiają programowanie refleksyjne to **Class**, **Field**, **Method**, **Array**, **Constructor**. Są one zgrupowane w pakietach `java.lang` i `java.lang.reflect`.

KLASY

Każdy obiekt w Javie jest instancją klasy **Object**. Każdy typ (obiektowy, prymitywny, tablicowy itp.) jest reprezentowany przez instancję klasy **Class**, którą uzyskujemy za pomocą **getClass()**.

```
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  
    }  
}
```

KLASY

```
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
}
}
```



class

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
```

Ten sposób jest szczególnie użyteczny w przypadku typów prymitywnych:

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

Class.forName()

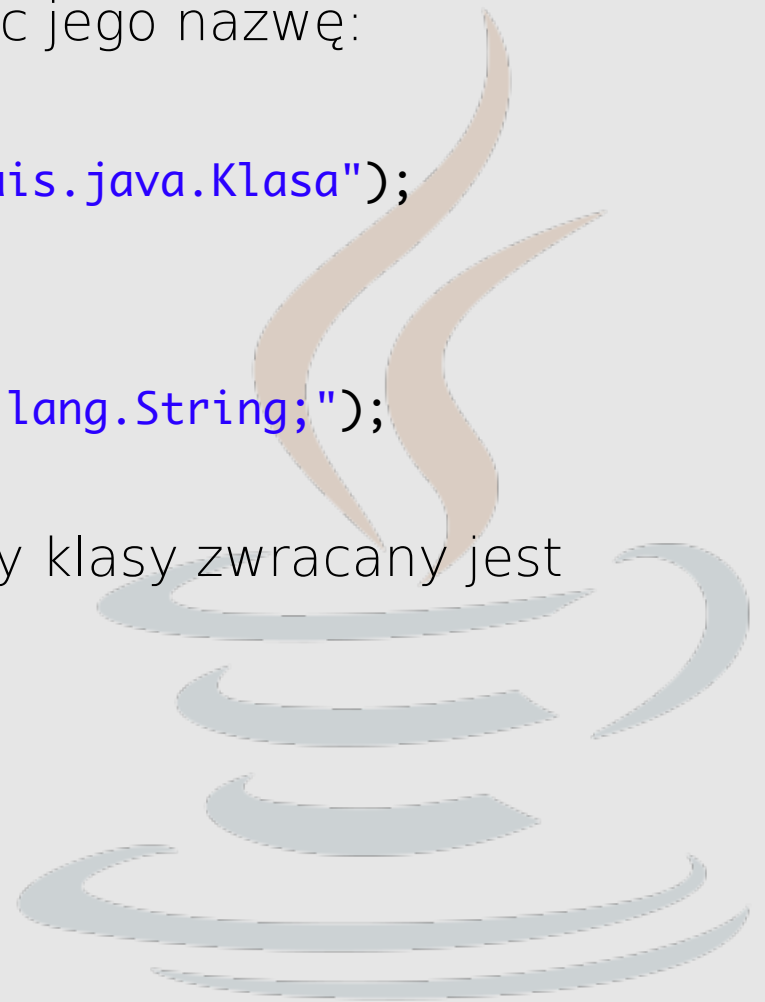
Obiekt **Class** można także otrzymać znając jego nazwę:

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

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

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

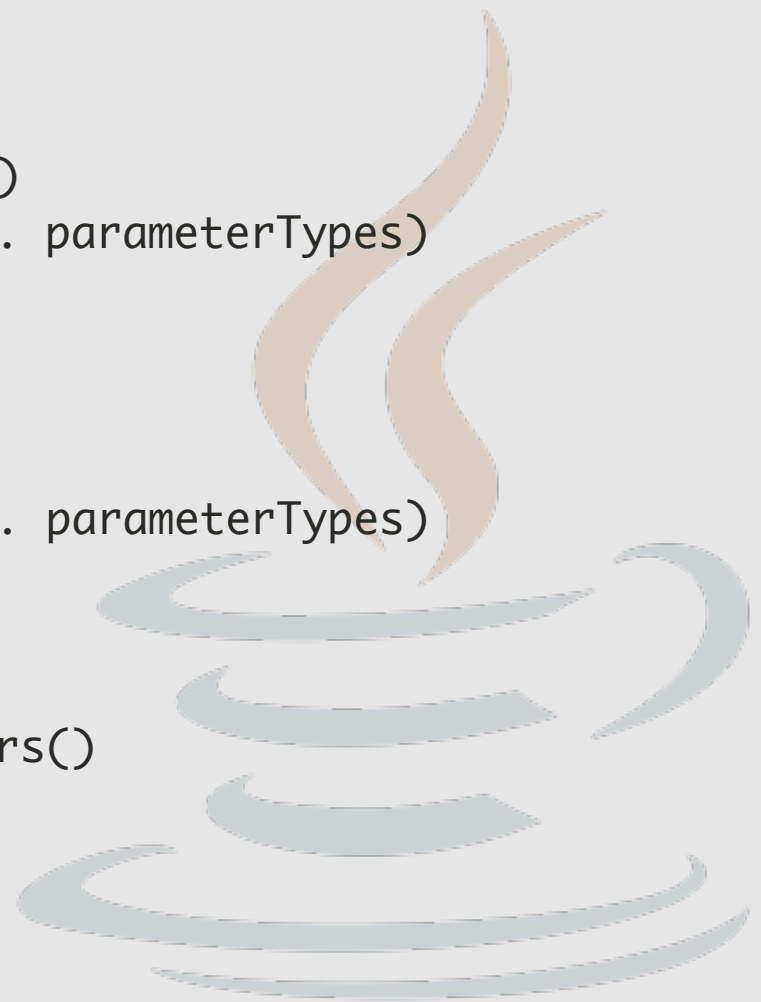
W przypadku podania niepoprawnej nazwy klasy zwracany jest wyjątek **ClassNotFoundException**.



KLASY

Wybrane metody klasy Class:

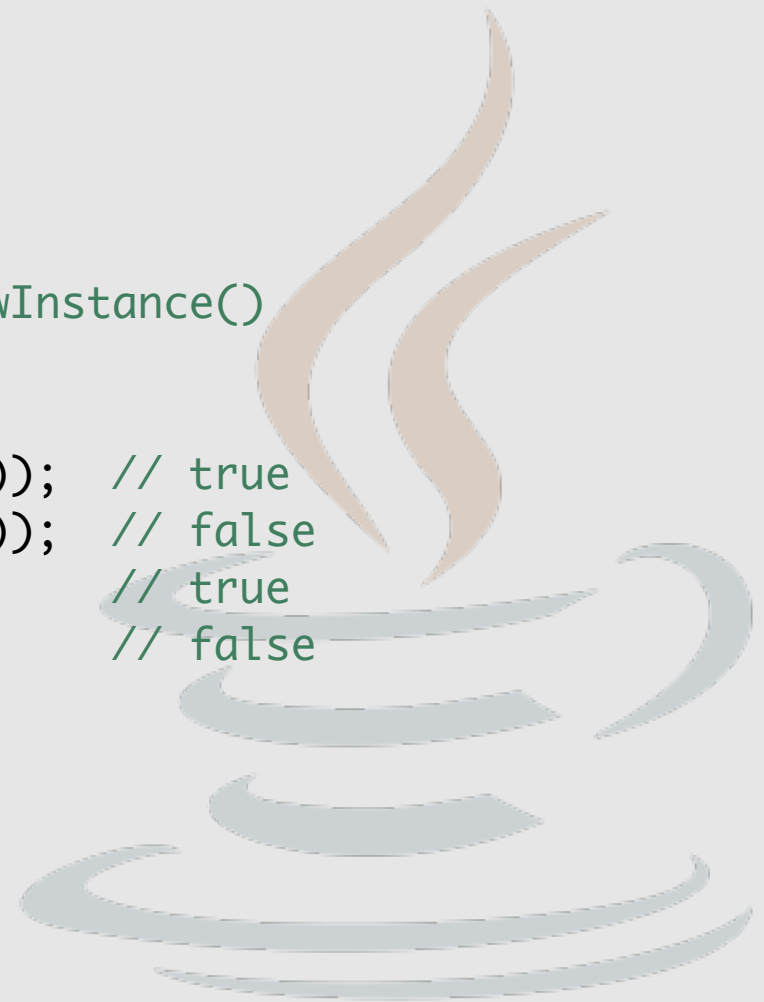
- `static Class<?> forName(String className)`
- `Constructor<T> getConstructor(Class<?>... parameterTypes)`
- `Constructor<?>[] getConstructors()`
- `Field getField(String name)`
- `Field[] getFields()`
- `Class<?>[] getInterfaces()`
- `Method getMethod(String name, Class<?>... parameterTypes)`
- `Method[] getMethods()`
- `int getModifiers()`
- `Class<? super T> getSuperclass()`
- `TypeVariable<Class<T>>[] getTypeParameters()`
- `boolean isArray()`
- `boolean isInterface()`
- `boolean isPrimitive()`
- `T newInstance()`



KLASY

```
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

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 wartość atrybutu w obiekcie `obj`
- `void setInt(Object obj, int i)` // ustawia wartość atrybutu typu `int`

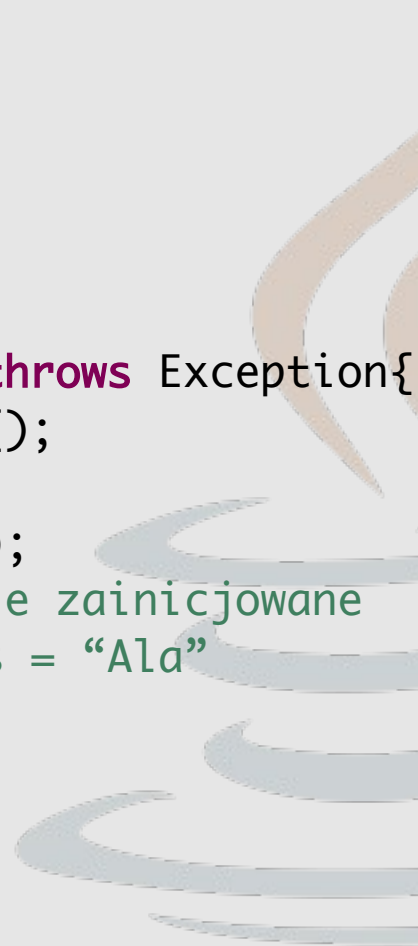
ATRYBUTY

```
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"); // FieldExample.s = "Ala"

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



METODY

Metoda jest reprezentowana przez instancję klasy **Method**. Wybrane metody:

- `Class<?>[] getExceptionTypes()` // zwraca klasy 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`. Wartość zwracana jest wynikiem działania wywołanej metody.
- `boolean isVarArgs()` // czy metoda ma nieokreśloną liczbę argumentów

invoke()

Przykład:

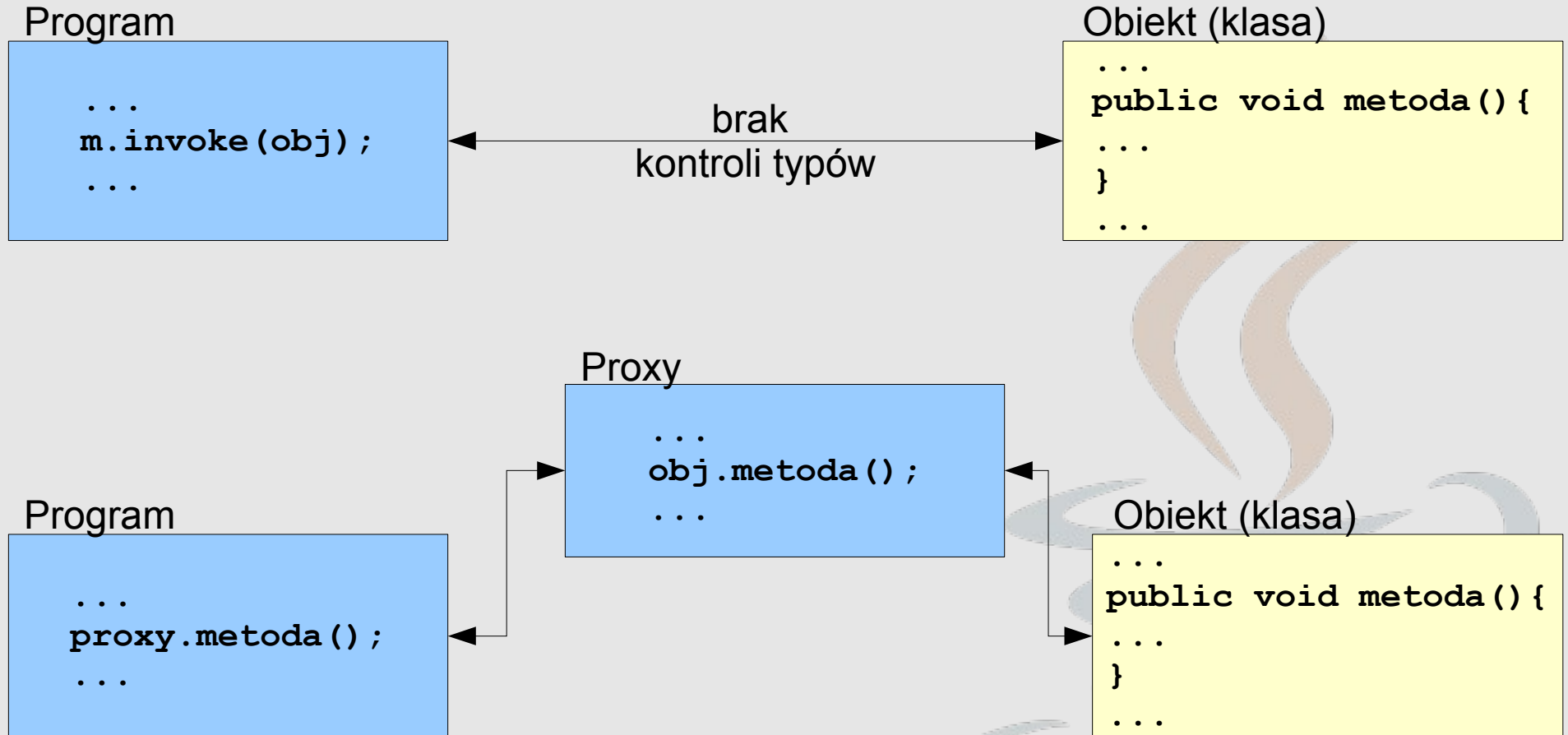
```
...  
Method m = Class.forName("MyClass").getDeclaredMethod(  
    "example3", null);  
m.invoke(null, null);  
...
```

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



klasa Proxy powinna być tworzona dynamicznie.

PROXY

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 });
```

PROXY

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**.

PROXY

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**.

PROXY

```
...  
Pisarz obj = new PisarzImpl();  
Method m = obj.getClass().getMethod(  
    "pisz",  
    new Class[] {String.class});  
m.invoke(obj, new Object[]{"hello world"});  
...
```

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

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

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);
    }
}
```

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

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

DZIĘKUJĘ ZA UWAGĘ