



СБЕРБАНК ТЕХНОЛОГИИ

Reflection

- Какую метаинформацию можно получить в рантайме о классах?
- Можно ли звать приватные методы класса из других классов?
- Зачем и как это делать ?

```
public final class Class<T> implements java.io.Serializable,  
                                       GenericDeclaration,  
                                       Type,  
                                       AnnotatedElement { ... }
```

Содержит методы для получения полной информации о классе, вызова методов и изменения полей.

- Имя, пакет класса
- Методы: (список, тип возвращаемого значения, имена и типы аргументов, модификаторы видимости)
- Поля (список, имена, типы, модификаторы видимости)
- Иерархия класса

- Определить класс объекта.
- Получить информацию о модификаторах класса, полях, методах, конструкторах и т.п.
- Выяснить, какие константы и методы принадлежат интерфейсу.
- Создать экземпляр класса, имя которого неизвестно до момента выполнения программы.
- Получить и установить значение свойства объекта.

○ Вызвать метод объекта

Области применения:

- Сериализация (Serializable, GSON, Jackson, JaxB, Kryo, etc...)
- Сервера приложений
- Фреймворки
- да почти что угодно что должно работать с «чужим» кодом

```
Class<Integer> c = Integer.class;
```

```
Class<Integer> c = int.class;
```

```
Class<String> c = String.class;
```

```
Class<long[]> cls = long[].class;
```

```
someObject.getClass();
```

*//Список всех **public** методов, объявленных в классе или унаследованных*

```
public Method[] getMethods()
```

//Список всех методов, объявленных в классе

```
public Method[] getDeclaredMethods()
```

//Метод с заданным именем и аргументами

```
public Method getMethod (String name,  
                          Class<?>...parameterTypes)
```

```
Method m = String.class.getMethod("replaceAll",  
                                   String.class, String.class)
```


*//Список всех **public** полей, объявленных в классе или унаследованных*

```
public Field[] getFields()
```

//Список всех полей, объявленных в классе

```
public Field[] getDeclaredFields()
```

//поле по имени

```
public Field getField(String name)
```

//Поле, объявленное в классе

```
public Field getDeclaredField(String name)
```

//Возвращает класс родителя

```
public native Class<? super T> getSuperclass();
```

```
System.out.println(String.class.getSuperclass());
```

```
System.out.println(Object.class.getSuperclass());
```

```
//Object
```

```
System.out.println(String.class.getSuperclass());
```

```
//null
```

```
System.out.println(Object.class.getSuperclass());
```

В слайдах содержатся простые для понимания примеры, которые показывают **ЧТО** можно делать через reflection.

Используйте reflection API, только если без него не обойтись.

КАК СОЗДАТЬ ЭКЗЕМПЛЯР КЛАССА ?

```
try {  
    //Зовётся конструктор без параметров  
    Person p = Person.class.newInstance();  
} catch (InstantiationException | IllegalAccessException e) {  
    ...  
}
```

// Зовётся конструктор со String аргументом

```
Person p2 = Person.class.getConstructor(String.class)
            .newInstance («Строка»);
```

```
private void setName(Object o, String name)
    throws NoSuchMethodException, InvocationTargetException, IllegalAccessException {

    Class<?> clazz = o.getClass();
    Method m = clazz.getMethod("setName", String.class);

    //мы передаем объект, у которого вызовется метод,
    //и параметры метода
    m.invoke(o, name);

}
```



```
Method m = clazz.getDeclaredMethod("setName",  
                                   String.class);  
m.setAccessible(true);  
m.invoke(o, name);
```

```
public class Person {  
    private final String name; // Можно поменять?  
  
    ...  
}
```

```
Person person = get();
```

```
Field name = Person.class.getDeclaredField("name");  
name.setAccessible(true);  
name.set(person, "Julia");
```

Можно достать метаинформацию о дженериках на уровне **класса**.

Информация, чем параметризованы **локальные объекты** стирается.

```
public class Runtime<T extends Number>
    implements Callable<Double> {
    private final List<Integer> integers = emptyList();

    public List<T> numbers() {return emptyList();}

    public List<String> strings() {return emptyList();}

    @Override
    public Double call() {return 0d;}
}
```

Давайте смотреть примеры!

Позволяют добавлять метайнформацию в класс.

Использовать эту информацию можно для разных целей.

- `@Override`
- `@Deprecated`
- `@SuppressWarnings`
- ...

Помечаются устаревшие методы, не рекомендованные к использованию в новом коде

Пример из класс Date:

```
* @deprecated As of JDK version 1.1,  
* replaced by Calendar.get(Calendar.HOUR_OF_DAY).  
*/
```

```
@Deprecated
```

```
public int getHours() {  
    return normalize().getHours();  
}
```

Показывает что текущий метод переопределяет метод родителя или реализует интерфейс.

Компилятор проверяет, что помеченный метод действительно это делает.

```
@Override  
public int run () {  
    ...  
}
```



```
//@Inherited
//@Documented
@Target (ElementType.FIELD)
@Retention (RetentionPolicy.RUNTIME)
public @interface NotNull {
}
```

```
@Target ({ElementType.FIELD, ElementType.METHOD})
@Retention (RetentionPolicy.RUNTIME)
public @interface NotNull {
}
```

Показывает на что можно вешать данную аннотацию

```
public enum ElementType {  
    TYPE,  
    FIELD,  
    METHOD,  
    PARAMETER,  
    CONSTRUCTOR,  
    LOCAL_VARIABLE,  
    ANNOTATION_TYPE,  
    PACKAGE,  
    TYPE_PARAMETER,  
    TYPE_USE  
}
```

Показывает на каком уровне доступна аннотация

```
public enum RetentionPolicy {  
    SOURCE,  
    CLASS, // по умолчанию  
    RUNTIME  
}
```

Возвращаемые значения могут быть примитивами, String, Class, enums

```
@Target (ElementType.FIELD)  
@Retention (RetentionPolicy.RUNTIME)  
public @interface ValidLength {  
    int min ();  
  
    int max ();  
}
```

Можно задавать default значения

```
@Target(ElementType.FIELD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface MinLength {  
    int value() default 3;  
}
```

```
public class Person {  
    @ValidLength(min = 4, max = 10)  
    private final String name;  
    ...  
}
```

Название атрибута можно не указывать, если оно называется value

```
public class Person {  
    @MinLength(3)  
    private final String name;  
    ...  
}
```

```
Field f = ...
```

```
if (f.isAnnotationPresent(ValidLength.class)) {  
    ValidLength an=f.getAnnotation(ValidLength.class);  
    int max = an.max();  
    int min = an.min();  
    ...  
}
```


ПРИМЕР ПРОВЕРКИ ПОЛЕЙ ПО АННОТАЦИЯМ

```
public void validateStringLength(Object o) throws Exception {
    Class<?> clazz = o.getClass();
    for (Field field : clazz.getDeclaredFields()) {
        if (field.isAnnotationPresent(ValidLength.class)) {
            ValidLength an= field.getAnnotation(ValidLength.class);
            int max = an.max();
            int min = an.min();

            String value = field.get(o).toString();
            if (value.length() < min) {
                throw new IllegalStateException(field.getName()
+ " length should be between " + min + " and " + max);
            }
        }
    }
}
```

Позволяет перехватывать в рантайме вызовы методов интерфейса и обрабатывать их.

Прокси может притворяться любым интерфейсом.

```
Calculator calculator = new CalculatorImpl();  
calculator.calc(1);  
calculator.calc(1); // повторный расчет
```

```
Calculator cached = ProxyUtils.makeCached(calculator);  
cached.calc(1);  
cached.calc(2);  
cached.calc(1); // результат из кеша
```

Прокси перехватывает вызовы интерфейса и перенаправляет их по сети другому серверу и возвращает результат.

Это позволяет быстро создать клиент любого интерфейса(На удаленной машине должны быть слушатели вызова, созданные, например, тоже через Proxy).

```
Calculator calc = ProxyUtils.client(Calculator.class);  
calc.calc(1); // перехват вызова и отправка удаленной  
               машине
```

```
Service service = ProxyUtils.client(Service.class);  
service.run();
```


Поведение прокси задается в реализации интерфейса InvocationHandler

```
public interface InvocationHandler {  
    Object invoke(Object proxy, Method method,  
                  Object[] args) throws Throwable;  
}
```

```
public class LogHandler implements InvocationHandler {  
    private final Object delegate;  
  
    public LogHandler(Object delegate) {  
        this.delegate = delegate;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method,  
        Object[]args) throws Throwable {  
        System.out.println("Started " + method.getName());  
        Object result = method.invoke(delegate, args);  
        System.out.println("Finished " + method.getName() + ".  
            Result " + result);  
        return result;  
    }  
}
```

```
List<String> loggedList = (List<String>)
    Proxy.newProxyInstance(
        ClassLoader.getSystemClassLoader(),
        new Class[]{List.class},
        new LogHandler(new ArrayList<String>())
    );

//реализация методов интерфейса List зависит
//от передаваемого класса в конструкторе LogHandler
```


Вывести на консоль все методы класса, включая все родительские методы

(включая приватные)

Вывести все геттеры класса

Проверить что все String константы имеют значение = их имени

```
public static final String MONDAY = "MONDAY";
```

Реализовать кэширующий прокси

Просмотреть основные моменты работы с reflection и dynamic proxy: <http://tutorials.jenkov.com/java-reflection/index.html>

Реализовать следующий класс по документации

```
public class BeanUtils {  
    /**  
     * Scans object "from" for all getters. If object "to"  
     * contains correspondent setter, it will invoke it  
     * to set property value for "to" which equals to the property  
     * of "from".  
     * <p/>  
     * The type in setter should be compatible to the value returned  
     * by getter (if not, no invocation performed).  
     * Compatible means that parameter type in setter should  
     * be the same or be superclass of the return type of the getter.  
     * <p/>  
     * The method takes care only about public methods.  
     *  
     * @param to Object which properties will be set.  
     * @param from Object which properties will be used to get values.  
     */  
    public static void assign(Object to, Object from) {... }  
}
```