




Lesson 23

14.06.2021






```
public class Ex1 extends Test {
    public void display() throws IOException {
        System.out.println("Derived");
    }


    public static void main(String[] args) throws IOException {
        Ex1 object = new Ex1();
        object.display();
    }
}

class Test {

    public void display() throws IOException {
        System.out.println("Test");
    }

}
```





```
class Helper {  
    private int data;  
  
    private Helper() {  
        data = 5;  
    }  
}  
  
public class Ex2 {  
    public static void main(String[] args) {  
        Helper help = new Helper();  
        System.out.println(help.data);  
    }  
}
```

```
class Base {
    private int data;

    public Base() {
        data = 5;
    }

    public int getData() {
        return this.data;
    }
}

class Ex3 extends Base {
    private int data;


    public Ex3() {
        data = 6;
    }

    public int getData() {
        return data;
    }

    public static void main(String[] args) {
        Ex3 myData = new Ex3();
        System.out.println(myData.getData());
    }
}
```



```
public static void main(String[] args) {  
    int[] arr = {2, 1, 0};  
    for (int i : arr) {  
        System.out.println(arr[i]);  
    }  
}
```



```
public class Ex5 {
    private static void checkData() throws SQLException {
        try {
            throw new SQLException();
        } catch (Exception e) {
            e = null;
            throw e;
        }
    }


    public static void main(String[] args) {
        try {
            checkData(); //Line 17
        } catch (SQLException e) {
            System.out.println("NOT AVAILABLE");
        }
    }
}
```



Рефлексия кода, reflection

Рефлексия (от reflexio - обращение назад) - это механизм исследования данных о программе во время её выполнения. Рефлексия в Java осуществляется с помощью Java Reflection API, состоящий из классов пакетов `java.lang` и `java.lang.reflect`. В информатике рефлексия означает процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения.





Java Reflection API позволяет получать информацию о конструкторах, методах и полях классов и выполнять следующие операции над полями и методами объекта/класса :

- ❖ определение класса объекта;
- ❖ получение информации о полях, методах, конструкторах и суперклассах;
- ❖ получение информации о модификаторах полей и методов;
- ❖ создание экземпляра класса, имя которого неизвестно до момента выполнения программы;
- ❖ определение и изменение значений свойств объекта/класса;
- ❖ вызов методов объекта/класса.

Определение свойств класса

В работающем приложении для **получения класса** необходимо использовать метод `forName (String className)`.

```
public class ClassDefinition {  
    public static void main(String[] args) throws ClassNotFoundException {  
        Class fooRefl = Class.forName("Lesson24.reflection.Foo");  
        System.out.println(fooRefl.getName());  
    }  
}  
  
class Foo{  
  
    void print(){  
        System.out.println("Class >> Foo.java");  
    }  
}
```

Определение интерфейсов класса

Для получения в режиме run-time списка реализующих классом интерфейсов, необходимо получить Class и использовать его метод **getInterfaces()**

```
public class ClassGetInterfaces {  
    public static void main(String[] args) {  
        Class<?>  cls = ArrayList.class;  
        Class<?>[] ifs = cls.getInterfaces();  
  
        System.out.println("List of interfaces\n");  
        for(Class<?> ifc : ifs) {  
            System.out.println (ifc.getName());  
        }  
    }  
}
```



Определение конструкторов класса

Метод класса **getConstructors()** позволяет получить массив открытых конструкторов типа *java.lang.reflect.Constructor*. После этого, можно извлекать информацию о типах параметров конструктора и генерируемых исключениях :

```
public class ClassGetConstructor {  
    public static void main(String[] args) throws ClassNotFoundException {  
        Class<?> cls = Class.forName("Lesson24.reflection.Baz");  
        Constructor[] constructors = cls.getConstructors();  
        for (Constructor constructor : constructors) {  
            System.out.println(constructor);  
            Class<?>[] params = constructor.getParameterTypes();  
            for (Class<?> param : params) {  
                System.out.println(param.getName());  
            }  
        }  
    }  
}
```

Определение полей класса

Метод `getFields()` объекта `Class` возвращает массив открытых полей типа `java.lang.reflect.Field`, которые могут быть определены не только в данном классе, но также и в его родителях (суперклассе), либо интерфейсах, реализованных классом или его родителями. Класс `Field` позволяет получить имя поля, тип и модификаторы

```
public class ClassGetFields {  
    public static void main(String[] args) throws ClassNotFoundException {  
        Class<?> cls = Class.forName("Lesson24.reflection.Fee");  
        Field[] fields = cls.getFields();  
        for (Field field : fields) {  
            Class<?> fld = field.getType();  
            System.out.println("Class name : " + field.getName());  
            System.out.println("Class type : " + fld.getName());  
        }  
    }  
}
```



Определение методов класса

Метод **getMethods()** объекта `Class` возвращает массив открытых методов типа *java.lang.reflect.Method*. Эти методы могут быть определены не только в классе, но также и в его родителях (суперклассе), либо интерфейсах, реализованных классом или его родителями. Класс *Method* позволяет получить имя метода, тип возвращаемого им значения, типы параметров метода, модификаторы и генерируемые исключения.



Изменения значения закрытого поля класса

Чтобы изменить значение закрытого (*private*) поля класса необходимо получить это поле методом **getDeclaredField ()** и вызвать метод **setAccessible (true)** объекта **Field**, чтобы открыть доступ к полю. После этого значение закрытого поля можно изменять, если оно не *final*. В следующем примере определен внутренний класс *PrivateFinalFields* с набором закрытых полей; одно из полей *final*. При создании объекта класса поля инициализируются.



Получение типа возвращаемого значения

Получение аннотаций метода

Получение бросаемых исключений




Документирование javadoc

Javadoc — это генератор документации в HTML-формате из комментариев исходного кода Java и определяет стандарт для документирования классов Java. Для создания доклетов и тэглетов, которые позволяют программисту анализировать структуру Java-приложения, javadoc также предоставляет API. В каждом случае комментарий должен находиться перед документируемым элементом.

При написании комментариев к кодам Java используют три типа комментариев :

```
// однострочный комментарий;  
/* многострочный комментарий */  
/** комментирование документации */
```

С помощью утилиты **javadoc**, входящей в состав JDK, комментариев документации можно извлекать и помещать в HTML файл. Утилита **javadoc** позволяет вставлять HTML тэги и использовать специальные ярлыки (дескрипторы) документирования. HTML тэги заголовков не используют, чтобы не нарушать стиль файла, сформированного утилитой.

Дескрипторы **javadoc**, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ * игнорируется). Дескрипторы, начинающиеся с фигурной скобки, например **{@code}**, называются встроенными и могут применяться внутри описания.

Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом.

Дескриптор

@author

@version

@since

@see

@param

@return

@exception имя_класса описание

@throws имя_класса описание

@deprecated

{@link reference}

{@value}

Применение

Класс, интерфейс

Класс, интерфейс

Класс, интерфейс, поле, метод

Класс, интерфейс, поле, метод

Метод

Метод

Метод

Метод

Класс, интерфейс, поле, метод

Класс, интерфейс, поле, метод

Статичное поле

Описание

Автор

Версия. Не более одного дескриптора на класс

Указывает, с какой версии доступно

Ссылка на другое место в документации

Входной параметр метода

Описание возвращаемого значения

Описание исключения, которое может быть послано из метода

Описание исключения, которое может быть послано из метода

Описание устаревших блоков кода

Ссылка

Описание значения переменной

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.0.0-M1</version>
      <configuration>
        <stylesheetfile>${basedir}/src/main/javadoc/stylesheet.css</stylesheetfile>
        <show>public</show>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

mvn javadoc:javadoc



Debug

Подробные сообщения, используемые во время отладки приложения

Info

Информационные сообщения о том, что происходит в приложении

Warn

Предупреждения о возникновении нежелательной ситуации

Error

Ошибки при которых приложение способно продолжить работать

Fatal

Фатальные ошибки, обычно приводящие к завершению работы приложения

Что нужно логировать

Разумеется, логировать все подряд не стоит. Иногда это и не нужно, и даже опасно. Например, если залогировать чьи-то личные данные и это каким-то образом всплывет на поверхность, будут реальные проблемы, особенно на проектах, ориентированных на Запад. Но есть и то, что **логировать обязательно**:

- **Начало/конец работы приложения.** Нужно знать, что приложение действительно запустилось, как мы и ожидали, и завершилось так же ожидаемо.
- **Вопросы безопасности.** Здесь хорошо бы логировать попытки подбора пароля, логирование входа важных юзеров и т.д.
- **Некоторые состояния приложения.** Например, переход из одного состояния в другое в бизнес процессе.
- **Некоторая информация для дебага,** с соответственным уровнем логирования.
- **Некоторые SQL скрипты.** Есть реальные случаи, когда это нужно. Опять-таки, умелым образом регулируя уровни, можно добиться отличных результатов.
- **Выполняемые нити(Thread)** могут быть логированы в случаях с проверкой корректной работы.

Популярные ошибки в логировании

Нюансов много, но можно выделить несколько частых ошибок:

- **Избыток логирования.** Не стоит логировать каждый шаг, который чисто теоретически может быть важным. Есть правило: **логи могут нагружать работоспособность не более, чем на 10%**. Иначе будут проблемы с производительностью.
- **Логирование всех данных в один файл.** Это приведет к тому, что в определенный момент чтение/запись в него будет очень сложной, не говоря о том, что есть ограничения по размеру файлов в определенных системах.
- **Использование неверных уровней логирования.** У каждого уровня логирования есть четкие границы, и их стоит соблюдать. Если граница расплывчатая, можно договориться какой из уровней использовать.



Давайте рассмотрим уровни на примере log4j, вот они в порядке уменьшения:

- **FATAL:** ошибка, после которой приложение уже не сможет работать и будет остановлено, например, JVM out of memory error;
- **ERROR:** уровень ошибок, когда есть проблемы, которые нужно решить. Ошибка не останавливает работу приложения в целом. Остальные запросы могут работать корректно;
- **WARN:** обозначаются логи, которые содержат предостережение. Произошло неожиданное действие, несмотря на это система устояла и выполнила запрос;
- **INFO:** лог, который записывает важные действия в приложении. Это не ошибки, это не предостережение, это ожидаемые действия системы;
- **DEBUG:** логи, необходимые для отладки приложения. Для уверенности в том, что система делает именно то, что от нее ожидают, или описания действия системы: “method1 начал работу”;
- **TRACE:** менее приоритетные логи для отладки, с наименьшим уровнем логирования;
- **ALL:** уровень, при котором будут записаны все логи из системы.