Java: магия отражений

Часть I. Основы

**Один из самых удивительных и ярких механизмов языка Java – технология «отражения» (Java Reflection). К сожалению, в популярных учебниках нелегко найти подробную информацию об этой интереснейшей области. А тем более – о подводных камнях и неожиданных возможностях, возникающих при программировании с использованием отражений. Между тем, именно отражения позволяют Java с неподражаемым изяществом справляться с задачами, традиционно непростыми в других языках – такими, как создание оболочки для компиляции Java-проектов (наподобие Borland JavaBuilder или NetBeans), визуальное проектирование графических компонентов (JavaBeans), сериализация объектов и распределенные вычисления (RMI), и многие другие.**

Мне бы хотелось предложить вам небольшую экскурсию по миру отражений Java. По ходу дела я буду показывать, как использовать мощь этой технологии для изящного решения различных практических задач.

Мы начнем с простых вещей – простых для Java, но не столь тривиальных в других языках. Пример такой «простой вещи» – динамическое расширение вашей программы новыми классами от сторонних разработчиков, не требущее переписывания и даже перекомпиляции вашей системы. По мере знакомства с отражениями, задачи будут усложняться. В качестве завершающего примера, полностью использующего возможности отражений, я расскажу, как реализовать высокоэффективный интерпертатор выражения (формулы), написанного на языке Java – эквивалент оператора eval() в скриптовых языках типа Perl или JavaScript.

Все написанное ниже относится к последней (на момент написания статьи) версии Java: Sun Java SDK 1.4.

**Где искать мир отражений?**

Отражения в Java – это два класса Class и ClassLoader, расположенных в пакете java.lang, и специальный пакет java.lang.reflect, содержащий (в версии Java SDK 1.4) 12 вспомогательных классов: Array, Member, Constructor, Field, Method, Modifier, InvocationHandler, Proxy, ReflectAccess, ReflectPermission, InvocationTargetException, UndeclaredThrowableExceptioCfn.

Проще всего осваивать технику отражений, начиная с класса Class. Более сложные вещи потребуют применения классов из пакета java.lang.reflect, прежде всего классов, описывающих: Constructor, Field и Method. «Высший пилотаж» работы с отражениями – это, пожалуй, создание грамотных наследников ClassLoader, позволяющих реализовать собственную систему загрузки классов.

В этой статье я покажу, как грамотно использовать эти классы и их методы.

В любом случае, статья не заменяет справочную документацию фирмы Sun. Самую полную и свежую фирменную документацию можно найти в Интернете по адресу: [http://java.sun.com](http://java.sun.com/).

**Класс по имени Class**

В мире отражений все начинается с класса «java.lang.Class». Для каждого Java-класса или интерфейса, присутствующего в системе, существует экземпляр Class, содержащий подробную низкоуровневую информацию об этом классе. Например, с помощью этого экземпляра можно узнать списки членов класса – полей, методов, конструкторов, а также обратиться к этим членам. Кроме того, класс Class содержит ряд статических методов, обеспечивающих взаимодействие с внутренними механизмами Java, отвечающими за загрузку и управление классами.

Как получить экземпляр Class, соответствующий данному классу (или интерфейсу) – например, классу java.io.File? Для этого есть два основных способа.

A. Просто добавляем к имени класса суффикс «.class», например:

**Class clazz= byte.class**

(«clazz» – сознательно искаженное от «class»: компилятор не позволяет использовать в качестве идентификатора зарезервированное слово «class».)

B. Если мы располагаем экземпляром некоторого класса, может быть даже неизвестного в данной точке программы, можно вызвать метод getClass(), присутствующий в каждом Java-объекте (унаследованный от класса Object):

**void myFunction(Object o) {**

**Class clazz= o.getClass();**

**что-то делаем с обьектом clazz;**

**}**

**...**

**java.io.File f= new java.io.File("/tmp/1.txt");**

**myFunction(f);**

**...**

Здесь есть любопытный нюанс. Вообще-то, примитивные типы Java – boolean, char, byte, short, int, long, float, double – обычно не считаются полноценными классами. Они не унаследованы от Object, для них не работает наследование и т.д. Тем не менее, с ними тоже ассоциированы экземпляры Class, которые можно получить способом A, например:

**Class clazz= java.io.File.class;**

Существует даже специальный объект void.class – он используется в довольно экзотических ситуациях при вызове методов через отражения. Экземпляры типа Class есть также у любого массива, например:

**Class clazz= byte[].class**

или

**byte[] v= new byte[34]; Class clazz= v.getClass();**

Что же можно сделать, располагая переменной типа Class для некоторого класса?

Прежде всего, можно получить полное имя класса (скажем, для отладочной печати) методом getName(). Например: String.class.getName() возвращает «java.lang.String».

Интересно посмотреть на имена классов для примитивных типов и для массивов:

**float.class.getName() возвращает «float»**

**float[].class.getName() возвращает «[F»**

**float[][].class.getName() возвращает «[[F»**

**String[].class.getName() возвращает [Ljava.lang.String;»**

Полностью алгоритм построения такого имени описан в документации на Java.

Можно проверить, не является ли класс интерфейсом, массивом или примитивным типом: методы isInterface(), isArray() и isPrimitive(). Если класс – массив, можно получить тип его элементов (т.е. соответствующий объект Class): метод getComponentType(). Для не-массивов этот метод вернет null.

Даже такие простейшие механизмы позволяют сделать нечто полезное. Вот пример – функция для подсчета объема памяти, занятого массивом примитивных типов:

**public static int sizeOfArray(Object v) {**

**if (v==null) return 0;**

**Class componentClass= v.getClass().getComponentType();**

**if (componentClass==null)**

**throw new IllegalArgumentException("Not an array");**

**int s= componentClass==byte.class? 2:**

**componentClass==short.class? 4:**

**componentClass==char.class? 4:**

**componentClass==int.class? 4:**

**componentClass==float.class? 8:**

**componentClass==long.class? 8:**

**componentClass==double.class? 16:**

**-1;**

**// не обрабатываем тип boolean: для него размер элемента не специфицирован документацией по языку Java**

**if (s==-1)**

**throw new IllegalArgumentException("Unknown component type");**

**return java.lang.reflect.Array.getLength(v)\*s;**

**// метод getLength() класса java.lang.reflect.Array получает на входе массив произвольного типа и возвращает его длину**

**}**

**java.lang.reflect.Array – «странный» способ работать с массивами**

Прежде чем переходить к более мощным (и ценным) возможностям класса Class, давайте обратим внимание на java.lang.reflect.Array. Этот очень простой класс из пакета java.lang.reflect представляет собой библиотеку статических методов, позволяющих выполнять все основные примитивные действия с массивом неизвестного (на этапе компиляции) типа, представленного общим типом данных Object.

Пример мы видели в приведенной выше функции sizeOfArray() – метод getLength(). Этот метод объявлен в исходном коде фирмы Sun следующим образом:

**public static native int getLength(Object array)**

**throws IllegalArgumentException;**

Любой массив в Java является объектом, т.е. наследником Object, поэтому его можно передать в getLength() в качестве параметра. На самом деле, только массивы и можно передавать в этот метод – как и в большинство других методов класса Array. Для объектов других типов методы класса Array возбуждают исключение IllegalArgumentException.

Большинство остальных методов класса Array предназначены для чтения и записи элементов в массив:

**public static Object get(Object array, int index)**

возвращает элемент номер index; если массив состоит из элементов примитивного типа (byte, char и т.п.), возвращается объект-оболочка (Byte, Char, ...);

**public static boolean getBoolean(Object array, int index),**

**public static char getChar(Object array, int index)**

**и аналогичные методы для типов byte, short, int, long, float, double**

специальные версии get на случай, когда массив состоит из элементов соответствующего примитивного типа;

**public static void set(Object array, int index, Object value),**

**public static void setBoolean(Object array, int index, boolean z),**

**public static void setChar(Object array, int index, char c)**

**и т.д.**

обратные функции для записи элементов в массив.

Наконец, можно создать массив c помощью одного из двух методов:

**public static Object newInstance(Class componentType, int length),**

**public static Object newInstance(Class componentType, int[] dimensions)**

Первый метод удобен для одномерных массивов, второй – позволяет создавать сразу многомерные массивы. В качестве componentType передается класс элементов – например, один из классов примитивных типов boolean.class, char.class и т.д.

Все подробности – в документации фирмы Sun.

На первый взгляд у неискушенного Java-программиста все это вызывает некоторое недоумение. Зачем столь замысловатым способом манипулировать с массивом, когда можно просто воспользоваться встроенными языковыми средствами? Зачем писать:

**v.getByte(n)**

для массива v типа byte[], когда можно просто написать v[n]?

Ответ – класс Array позволяет писать универсальные функции, принимающие на вход массив произвольного, неизвестного заранее типа. Пример такой функции мы уже видели – sizeOfArray() из предыдущего пункта. Если бы не класс Array, нам бы либо пришлось написать 7 версий этой функции для каждого варианта массивов примитивных типов, либо заменить вызов java.lang.reflect.Array.getLength(v) чем-то вроде:

**v instanceof byte[]? ((byte[])v).length:**

**v instanceof short[]? ((short[])v).length:**

**v instanceof char[]? ((char[])v).length:**

**v instanceof int[]? ((int[])v).length:**

**v instanceof long[]? ((long[])v).length:**

**v instanceof float[]? ((float[])v).length:**

**v instanceof double[]? ((double[])v).length: -1**

Приведем более практичный пример использования класса Array. Достаточно часто возникает задача преобразовать произвольный объект в текстовую строку – например, в целях отладки. Предназначенный для этого метод toString() обычно выдает достаточно внятную информацию о содержимом объекта. Но, к сожалению, для массивов этот метод ведет себя весьма тупо – просто возвращает внутренний адрес массива.

Я написал простую функцию toS(Object v, String separator), преобразующую произвольный массив в строку. Все элементы массива преобразуются (стандартным образом) в строки и конкатенируются через разделитель separator, заданный в качестве параметра функции. Вот текст этой функции:

**public static String toS(Object v, String separator) {**

**if (v==null) return "";**

**if (v.getClass().isArray()) {**

**int len;**

**if ((len=java.lang.reflect.Array.getLength(v))==0) return "";**

**StringBuffer result= new StringBuffer();**

**for (int k=0; k**

**if (k>0) result.append(separator);**

**result.append(String.valueOf(**

**java.lang.reflect.Array.get(v,k)));**

**}**

**return result.toString();**

**}**

**return String.valueOf(v);**

**}**

Если аргумент v не является массивом, действие toS не отличается от стандартного метода v.toString(). Если v==null, возвращается пустая строка (обычно это удобнее стандартной реакции – возврата строки «null»).

Без класса Array пришлось бы написать 9 вариантов такой функции – для 8 примитивных типов и для массива объектов произвольного типа Object[].

Здесь нужно сделать одно важное замечание. Хотя класс Array действительно позволяет существенно экономить текст программы и не писать разные варианты метода для массивов разных типов, следует иметь в виду – получаемый код сравнительно неэффективен. Скажем, цикл суммирования всех элементов числового массива через вызов java.lang.reflect.Array.getDouble() будет работать на порядок дольше банального:

**double s= 0.0;**

**for (int k=0; k**

В случае функции toS() разница была бы непринципиальной, так как преобразование числа в строку – сравнительно медленная операция.

**Class.getResourceAsStream() – ресурсы**

Один из самых очевидных примеров использования класса Class – загрузка ресурсов.

Ресурс в Java – это файл с данными, прилагаемый к вашей программе и обычно размещаемый «рядом» с class-файлами. В графических приложениях и апплетах это чаще всего изображения (jpg- или gif-файлы), но с таким же успехом это может быть и файл специального формата – например, содержащий справочные таблицы или какие-либо ваши объекты.

Наиболее общий способ прочитать файл-ресурс произвольного формата, расположенный «рядом» с некоторым классом – обратиться к методу getResourceAsStream() объекта Class, соответствующего данному классу. В качестве параметра этому методу нужно передать путь к файлу ресурса относительно каталога, в котором размещен ваш класс. В качестве результата getResourceAsStream() вернет объект InputStream, с помощью которого можно прочитать файл ресурса стандартными средствами ввода/вывода Java.

Для наиболее популярных типов ресурсов, таких как изображения, обычно существуют более удобные способы прочитать ресурс – например, метод getImage() класса java.applet.Applet. Но для текстовых файлов и файлов нестандартного формата getResourceAsStream(), как правило, – самое разумное решение.

Вот пример законченного класса, использующего эту технику:

**import java.io.\*;**

**public class MyClassWithResource {**

**public static final String myTextResourceName= "mydata.txt";**

**public static final String myTextResource;**

**static {**

**String s= "";**

**try {**

**InputStream stream= MyClassWithResource.class.getResourceAsStream(myTextResourceName);**

**if (stream==null)**

**throw new FileNotFoundException(myTextResourceName+" not found");**

**StringBuffer sb= new StringBuffer(stream.available());**

**InputStreamReader reader= new InputStreamReader(stream);**

**char[] buf= new char[32768];**

**int len;**

**while ((len=reader.read(buf,0,buf.length))>=0) {**

**sb.append(buf,0,len);**

**}**

**s= sb.toString();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**myTextResource= s;**

**}**

**public static void main(String[] args) {**

**System.out.println("Loaded resource:");**

**System.out.println(myTextResource);**

**}**

**}**

В этом примере файл «mydata.txt» должен быть расположен в том же каталоге, что и class-файл «MyClassWithResource.class».

Файл ресурса необязательно размещать в том же каталоге, что и class-файл. Если он расположен в одном из подкаталогов этого каталога, в качестве имени ресурса нужно передать относительный путь, разделяя имена подкаталогов символом «/» (как это принято в Internet и Unix). Можно также указать в качестве имени ресурса «абсолютный» путь, начинающийся с символа «/». Тогда Java будет искать ресурс во всех каталогах, перечисленных в путях поиска классов CLASSPATH – т.е. по тем же правилам, по которым отыскиваются class-файлы программы.

Может возникнуть вопрос – зачем нужен специальный метод класса Class, когда можно прочитать файл ресурса обычными средствами файлового ввода/вывода?

Основная причина – использование метода getResourceAsStream() является гораздо более общим решением, работающим в большем числе ситуаций.

Например, по традиции, законченные наборы классов – Java-приложения или библиотеки – принято упаковывать в архивы JAR и устанавливать на компьютер именно в таком виде. Классы Java прекрасно загружаются непосредственно из архива JAR, без предварительной распаковки. То же самое относится и к ресурсам, загружаемым методом getResourceAsStream() – или более специальными методами типа java.applet.Applet.getImage(). В то же время, обычные средства файлового ввода/вывода для чтения ресурса из JAR уже непригодны – нужно использовать специальные классы для анализа и чтения JAR-файлов.

Аналогичная ситуация – апплеты, когда файлы ресурсов и class-файлы находятся на сервере. Метод getResourceAsStream() в этом случае обеспечит чтение ресурса с сервера через Internet.

Тем не менее, в некоторых случаях для чтения ресурса все-таки может быть целесообразным использование традиционных средств файлового ввода/вывода – конечно, если вы не используете JAR и пишете сервлет или приложение, а не апплет. Например, для достижения максимальной производительности может понадобиться отобразить ресурс в память средствами отображения файлов из пакета java.nio, появившегося в Java начиная с версии SDK 1.4. Или, может быть, вы захотите просканировать каталог с ресурсами и загрузить все файлы с определенным расширением – для сканирования каталога в классе Class нет соответствующих средств.

Если ресурс, как это принято, расположен рядом с class-файлами – или, скажем, если вы хотите поработать с самими class-файлами вашего приложения – то придется решить следующую задачу: определить дисковый каталог, в котором находится ваш class-файл. Будьте внимательны: это отнюдь не простая задача.

Ниже приведен текст функции, возвращающей полный путь к class-файлу.

**public static java.io.File getClassFile(Class clazz) {**

**// The file will exist only if it is usual class-file,**

**// not a part of JAR or Web resource**

**String s= clazz.getName();**

**s= s.substring(s.lastIndexOf(".")+1);**

**s= clazz.getResource(s+".class").getFile();**

**try {**

**s= java.net.URLDecoder.decode(s,"UTF-8");**

**} catch(java.io.UnsupportedEncodingException e) {**

**}**

**return new java.io.File(s);**

**}**

Прежде всего мы получаем имя класса без имени пакета, добавляем к нему расширение «.class» и передаем методу getResource() объекта Class. Этот метод возвращает экземпляр класса java.net.URL, представляющий файл в виде универсального пути к ресурсу (URL, Universal Resource Locator). В нашем случае (когда классы расположены в обычных файлах в локальной файловой системе) URL будет выглядеть примерно так:

**file://путь\_к\_дисковому\_каталогу/имя\_класса.class**

Метод getFile() объекта URL «отрежет» префикс «file://», оставив все остальное без изменений.

А вот дальше начинаются сложности. Оставшийся путь к файлу записан в стандарте URL, который может отличаться от формата имен файлов, принятого в текущей файловой системе.

Первое отличие: в URL подкаталоги всегда разделяются прямым слэшем /, в то время как в операционных системах, отличных от Unix, могут использоваться другие символы (например, обратный слэш в случае Windows). Это различие несущественно для Java – классы пакета java.io прекрасно будут работать и с именем файла, записанным через прямые слэши.

Но есть и второе отличие. Если путь к файлу содержит пробелы, русские буквы или другие символы, недопустимые в стандарте URL, то они будут закодированы комбинациями вида %XX, где XX – ASCII-код символа. Такое имя файла непригодно для обработки средствами ввода/вывода Java. Для восстановления «нормального» имени файла нужно вызвать метод java.net.URLDecoder.decode(), обязательно указав при этом кодировку символов (encoding).

Опытным путем я установил, что для кодирования не-латинских букв в имени файла Java использует кодировку UTF-8 – по крайней мере, на Windows-платформе. К сожалению, я не нашел в документации прямых указаний, что это всегда будет так на всех платформах. Поэтому я бы порекомендовал, по возможности, все же размещать свои классы в каталогах, путь к которым состоит только из латинских символов – тогда приведенная выше функция будет достаточно надежна.

**Class.forName() и Class.newInstance() – динамическая загрузка классов**

Мы подошли к рассмотрению по-настоящему важных и интересных технологий мира отражений. Речь пойдет о фундаметальном и чрезвычайно мощном механизме Java: динамической загрузке произвольного класса по заданному имени. Эта возможность встроена непосредственно в Java и реализуется классом Class. В других языках типа С++ или Delphi аналогичных целей можно достигнуть, используя специальные средства конкретной операционной системы (типа загрузки dll в Windows функцией loadLibrary()), но в Java это сделано по-настоящему удобно.

Итак, вашему вниманию предлагается статический метод Class.forName(). Вот его формальное объявление:

**public static Class forName(String className)**

**throws ClassNotFoundException**

Метод отыскивает в системе (среди путей поиска классов CLASSPATH) класс с заданным именем className и возвращает соответствующий экземпляр класса Class. Имя className должно быть полным, т.е. включать имя пакета. Например:

**Class clazz= Class.forName(«java.lang.String»);**

Если такой класс отсутствует, возбуждается исключение ClassNotFoundException.

После получения переменной типа Class, следующее наиболее типичное действие – создание экземпляра только что загруженного класса. Для этого служит метод Class.newInstance(). Его объявление:

**public Object newInstance()**

**throws InstantiationException, IllegalAccessException**

В нашем примере вызов newInstance() мог бы выглядеть так:

**Object object= clazz.newInstance();**

Чтобы этот метод мог выполниться, у загруженного класса должен существовать пустой конструктор (без аргументов), либо – что по существу то же самое – не должно быть описано вообще никаких конструкторов. В противном случае будет возбуждено исключение InstantiationException.

Конечно, этого еще мало, чтобы работать с полученным объектом. Если вы не знаете, что умеет делать класс – какие у него есть методы, что они ожидают получить на входе и для чего они предназначены – вы не сможете извлечь из него ничего полезного. Для формального определения, что «умеет» класс, в Java существует стандартный механизм – интерфейсы. Остается просто применить этот механизм.

Например, предположим, ваша программа должна в некоторые моменты выполнять перевод с одного языка на другой. Формально это можно описать интерфейсом:

**public interface LanguageTranslator {**

public String translate(String source,String sourceLanguage,String targetLanguage);

**// переводит текст source с языка sourceLanguage на язык targetLanguage**

}

Пусть ваша программа сама по себе не умеет выполнять перевод, но ее можно расширить классами сторонних разработчиков, которые эту задачу решать умеют. Все эти классы реализуют интерфейс LanguageTranslator и имеют конструктор без параметров (либо лишены конструктора). В настройках вашей программы пользователь указывает имя такого класса, независимо инсталлированного в систему (или выбирает из списка классов), после чего для выполнения перевода ваша программа использует следующие операторы:

**String source= "текст, требующий перевода";**

**String sourceLanguage= "Russian";**

**String targetLanguage= "English";**

**Class clazz= Class.forName("полное\_имя\_класса\_переводчика");**

**Object object= clazz.newInstance();**

**if (!(object instanceof LanguageTranslator)) {**

**throw new Exception("...");**

**// сообщаем об ошибке: указанный класс не реализует требуемый интерфейс, т.е. не является переводчиком**

**}**

**String result= ((LanguageTranslator)object).translate(source,sourceLanguage,targetLanguage);**

Описанная техника может оказаться очень полезной практически в любой достаточно большой и серьезной системе, рассчитанной на разработку многими участниками. Многие компоненты таких систем являются достаточно изолированными, и их набор может быть совершенно неизвестен на этапе компиляции основной программы – известны лишь интерфейсы, который они обязуются реализовывать. Например, так обычно строятся системы plugin’ов – модулей, добавляемых к уже работающей системе. В подобных случаях механизм отражений – методы Class.forName() и Class.newInstance() – становится единственным грамотным решением.

**Constructor, Field, Method – работа с классами через отражения**

На самом деле технология отражений позволяет сделать гораздо больше, чем просто загрузить по имени некоторый (заранее неизвестный) класс и создать его экземпляр. Можно получить полный список всех конструкторов, полей и методов класса и обратиться к любому из них, передав (в случае конструктора или метода) список всех параметров. Для этого служат следующие методы класса Class:

**public Constructor[] getConstructors(),**

**public Field[]       getFields(),**

**public Method[]      getMethods(),**

**public Constructor[] getDeclaredConstructors(),**

**public Field[]       getDeclaredFields(),**

**public Method[]      getDeclaredMethods()**

Перечисленные методы возвращают массивы объектов типа Constructor, Field и Method. Эти классы содержатся в пакете java.lang.reflect и обеспечивают исчерпывающий доступ к полям, конструкторам и методам.

Сразу бросается в глаза наличие двух версий методов: getXXX и getDeclaredXXX (где XXX – «Constructors», «Fields» или «Methods»). Поначалу это может даже несколько сбить с толку – какой версией следует пользоваться?

getDeclaredXXX возвращает список членов класса (конструкторов, полей или методов), объявленных при описании класса, но не унаследованных от классов-предков. При этом в список включаются все члены, независимо от их уровня защиты – т.е. public, protected, private и дружественные члены.

getXXX возвращает полный список членов, объявленных в самом классе либо унаследованных от одного из предков. Но в этот список уже попадают только public-члены.

При желании, конечно, можно получить и максимально полную информацию – список всех членов, объявленных в самом классе либо в любом из его предков. Для этого достаточно организовать цикл по цепочке классов-предков, пользуясь специальным методом Class.getSuperclass().

Кроме получения полного списка, можно также отыскать в классе конкретный член. Для этого служат методы:

**public Constructor   getConstructor(Class[] parameterTypes),**

**public Field       getField(String name),**

**public Method   getMethod(String name, Class[] parameterTypes),**

**public Constructor getDeclaredConstructor(Class[] parameterTypes),**

**public Field       getDeclaredField(String name),**

**public Method     getDeclaredMethod(String name, Class[] parameterTypes)**

(Собственно, чаще используются как раз эти методы, а не описанные выше методы получения списков.)

Аргумент name в этих методах должен содержать имя требуемого члена. Аргумент parameterTypes связан с возможностью Java перегружать конструкторы и методы – определять несколько конструкторов, либо несколько методов с одинаковым именем, отличающихся только типами параметров. (В случае конструкторов это единственный способ создать много конструкторов класса.) В качестве parameterTypes нужно отдать массив объектов типа Class, соответствующих типам всех параметров конструктора или метода.

Разница между вариантами getXXX и getDeclaredXXX здесь та же самая, что и в случае методов получения списков.

Здесь я бы порекомендовал написать небольшой тест, который распечатает списки всех конструкторов, полей и методов (объявленных и унаследованных) какого-нибудь класса. Для преобразования объектов Constructor, Field, Method в строки можно использовать стандартный метод toString(). Попробуйте распечатать такие списки для классов без конструкторов, с пустым конструктором, с конструктором, обладающим параметрами, с private- и public-полями. Попробуйте унаследовать класс и переопределить в нем (под тем же именем) public-, protected- или private-поле. Такой текст хорошо помогает понять, как в Java устроены классы, конструкторы и наследование.

Как реально работать с классами Constructor, Field, Method?

Прежде всего, заметим, что все они реализуют общий интерфейс Member, позволяющий:

* получить символьное имя члена (конструктора, поля или метода) с помощью метода:

**public String getName();**

* получить обратную ссылку на класс, в котором объявлен данный член (т.е. тот класс, который возвращает этот член в соответствующем массиве getDeclaredXXX()), с помощью метода:

**public Class getDeclaringClass();**

* получить набор битовых флагов – модификаторов данного члена с помощью метода:

**public int getModifiers();**

Модификаторы – это битовые флаги, описывающие, обладает ли данный член следующими свойствами: public, private, protected, static, final, synchronized, volatile, transient, native, abstract или strictfp. Полный список модификаторов и средства работы с ними можно найти в классe java.lang.reflect.Modifier – см. документацию фирмы Sun.

Кстати заметим, что некоторые модификаторы (например, public) определены также у классов; их можно получить методом Class.getModifiers(). Для классов определен еще один модификатор – java.lang.reflect.Modifier.INTERFACE, означающий, что объект Class ассоциирован не с классом, а с интерфейсом.

Главное назначение классов Constructor, Field, Method – конечно, получить доступ к соответствующим членам, т.е. в случае конструкторов – создать экземпляр класса, в случае полей – прочитать или изменить поле, в случае методов – вызвать метод.

Это делается достаточно просто. Чтобы не дублировать документацию по Java, я просто приведу пример готового тестового класса, где использованы все основные приемы доступа к членам:

**import java.lang.reflect.\*;**

**public class TestClass {**

**public int a;**

**public TestClass(int a)  {this.a= a;}**

**public void b()          {a= 1;}**

**public void b(int p1)    {a= p1;}**

**public String toString() {return a+"";}**

**public static void main(String[] args) throws Exception {**

**1.  Class clazz= TestClass.class;**

// п.2.3. – создание объекта класса через конструктор

**2.  Constructor c= clazz.getConstructor(new Class[] {int.class});**

**3.  Object o= c.newInstance(new Object[] {new Integer(23)});**

**4.  Field f= clazz.getField("a");**

**5.  System.out.println(f.getInt(o));**

**6.  f.setInt(o,24);**

**System.out.println(o);**

// п.4.5. – вызов метода без параметров

**4.  Method m= clazz.getMethod("b",new Class[] {});**

**5.  m.invoke(o,new Object[] {});**

**System.out.println(o);**

// п.4.5. – вызов метода с параметрами

**4.  m= clazz.getMethod("b",new Class[] {int.class});**

**5.  m.invoke(o,new Object[] {new Integer(2)});**

**System.out.println(o);**

**}**

**}**

Т.е. логика создания экземпляра объекта с помощью конструктора следующая:

1. Создается через конструктор объект класса

2. Создается объект поля/метода с использованием объекта «clazz»

3. К объекту поля/метода применяется метод get(), set() / invoke() в который подставляется объект класса и возможно параметры.

Описание верхнего кода:

- Создание экземпляра объекта с помощью конструктора похоже на вызов метода Class.newInstance(). Но в данном случае можно легко создать экземпляр класса, не обладающий конструктором без параметров – нужно только знать список типов параметров требуемого конструктора.

- Для передачи параметров в конструкторы и методы используется массив объектов (типа Object[]). Если нужно передать параметр примитивного типа, он «заворачивается» в соответствующий класс-оболочку – в случае int это объект Integer. Такое «заворачивание» – традиционная практика в мире отражений.

- Для доступа к полю в классе Field реализованы общие методы get() и set(), рассчитанные на произвольный объект, и частные версии getBoolean(), getInt(), ..., setBoolean(), setInt(), ..., рассчитанные на примитивный тип поля. Первым параметром у этих методов всегда передается экземпляр объекта, к полю которого нужно получить доступ.

- Для вызова метода служит метод invoke() класса Method. Ему передается экземпляр объекта, метод которого следует вызвать, и список параметров в виде массива Object[].

- Если метод класса возвращает результат (а не описан как void, как в данном примере), этот результат будет возвращен в качестве результата invoke – в виде общего типа Object. (Примитивный тип в этом случае, как обычно, «заворачивается» в класс-оболочку.) т.е. Object ob = m.invoke(o, new Object[]{});

Располагая объектом Constructor или Method, можно узнать список типов всех параметров в виде массива Class[]: для этого служат методы Constructor.getParameterTypes() и Method.getParameterTypes(). В случае метода можно также узнать тип результата: Method.getReturnType(). Кстати, это та самая экзотическая ситуация, когда находит применение исключительно редкий объект void.class (я упоминал об этой объекте в разделе 2): void.class будет возвращен getReturnType(), если метод объявлен как void.

С помощью классов Field и Method можно также получить доступ к статическим полям и методам класса, даже не создавая экземпляра объекта. При этом в качестве первого параметра методов getXXX(), setXXX() или invoke() допускается передать null. Можно, скажем, легко написать Java-код, аналогичный по действию стандартной утилите «java» – а именно, запускающий статическую функцию «public static void main(String[] args)» у произвольного класса. Техника интерфейсов, описанная в предыдущем разделе, не позволила бы это сделать.

**Обход защиты Java**

В очень многих учебниках по Java подчеркивается, что модификаторы protected и private вместе с «дружественным» уровнем доступа (отсутствие модификаторов) позволяют обеспечить «100-процентную защиту» ваших полей и методов от использования посторонними классами. Например, вы никогда не сможете добраться до поля

**private char value[];**

объявленного в исходном коде класса String и представляющего реальное содержимое строки.

На самом деле все это не так. Смотрите, как можно добраться до этого самого поля и «нелегально» изменить строку – вопреки известному утверждению, что тип String является абсолютно неизменяемым:

**import java.lang.reflect.\*;**

**public class HackString {**

**public static void main(String[] args) throws Exception {**

**String s= "Hello!";**

**System.out.println(s);**

**Field f= s.getClass().getDeclaredField("value");**

**// Именно getDeclaredField, а не getField: последний метод просто не нашел бы скрытого поля**

**f.setAccessible(true);**

**char[] value= (char[])f.get(s);**

**value[5]= '?';**

**System.out.println(s);**

**}**

**}**

Здесь «магический» метод – setAccessible(). Этот метод (и симметричный getAccessible()) имеется у всех классов Constructor, Field, Method и предназначен специально для того, чтобы отключить стандартную проверку модификаторов, осуществляемую Java-машиной.

(Естественно, все это сработает только при условии, что в вашей версии Sun Java SDK реализация класса String точно так же основана на private-поле «char value[]». Так как это поле скрытое, фирма Sun вправе в любой момент переименовать его или вообще заменить чем-нибудь другим.) Спрашивается – зачем же это сделано? И разве это не является брешью в системе безопасности?

Что до второго вопроса – разумеется, метод setAccessible() контролируется менеджером безопасности Java (так же как, например, работа с файлами), и никакая мало-мальски защищенная Java-система не позволит злоупотребить подобной возможностью.

А чтобы понять, зачем это нужно, взгляните на любую среду разработки Java-проектов – скажем, NetBeans или JavaBuilder. Традиционная возможность подобных сред – показать все поля и методы некоторого класса, например, визуальной компоненты – в том числе и скрытые, а в некоторых случаях – дать возможность отредактировать значения полей. Язык Java уникален в том отношении, что подобные действия можно легко выполнить совершенно законными средствами самого языка, не прибегая, скажем, к анализу исходного текста программы.

На самом деле неограниченный доступ ко всем конструкторам, полям и методам даже гораздо более ценен. Этот механизм мира отражений дает возможность удобно и естественно реализовать чрезвычайно мощные технологии, нереализуемые (или крайне сложно реализуемые) другими способами.

Приведем два примера. Во-первых, в Java поддерживается механизм сериализации. Достаточно реализовать в вашем классе пустой интерфейс-индикатор java.io.Serializable, и появляется возможность полностью (т.е. со всеми полями и вложенными объектами) записать этот объект в поток java.io.ObjectOutputStream и впоследствии прочитать из потока java.io.ObjectInputStream.

Те, кто изучал механизм сериализации Java, согласятся – во многом этот механизм напоминает черную магию. Каким-то образом классы java.io.ObjectInputStream и java.io.ObjectOutputStream, без всяких дополнительных подсказок со стороны разработчика класса, «догадываются», как записать или прочитать все поля объекта, включая private-поля. Более того, для «подсказки», когда она все же требуется, используются private-методы writeObject() и readObject() – классы java.io.ObjectOutputStream и java.io.ObjectInputStream каким-то образом обнаруживают и вызывают эти методы.

Конечно же, в действительности эта «черная магия» – не что иное, как магия отражений. Описанные выше технологии, в том числе метод setAccessible(), в принципе,  позволяют даже разработать свою собственную схему сериализации, отличную отстандартной, предлагаемой фирмой Sun.

Второй пример – технология RMI. Это чрезвычайно мощный механизм, позволяющий распределять вычисления по сети – так, чтобы с точки зрения кода Java вызов метода выглядел, как обычно, а на самом деле этот метод отрабатывался на другом компьютере. Здесь тоже не обойтись без технологий отражения. Только они позволяют динамически превратить объект (со всеми своими полями) в поток данных, передать его по сети, реконструировать объект на другом компьютере и вызвать нужный метод – и все это скрыто от пользователя класса.

Конечно, все это не повод, чтобы использовать отражения не по назначению для «взлома» защиты Java. Отражения следует использовать только тогда, когда без этого нельзя обойтись – иначе вы потеряете все преимущества, которые дает идеология объектно-ориентированного программирования.

**Заключение**

Я постарался описать самые, на мой взгляд, важные и интересные аспекты технологии отражений. Статья – не справочник и не учебник. Многие вещи «остались за бортом». Не все методы классов были описаны; некоторые специфичные классы, такие как java.lang.reflect.Proxy, я вообще не рассматривал. Чтобы получить полную и точную информацию, всегда лучше обращаться к первоисточнику – документации фирмы Sun. Кроме сайта [http://java.sun.com](http://java.sun.com/), документацию почти всегда можно найти в комплекте поставки Java или извлечь из комментариев к исходным текстам фирмы Sun.