Классы решают все

Мы рассмотрели подавляющую часть конструкций, которые существуют в языке Java. Теперь пришло время осознать следующее — большинство задач, с которыми сталкиваются сегодняшние разработчики, очень часто используют либо уже готовые решения, либо стандартные конструкции для решения какой-то части. Это работа с файлами, работа с базами данных, работа со списками, множествами и многое другое. Именно это позволяет использовать уже готовые классы, которые можно назвать зрелыми решениями — на их создание потрачено много усилий и они являются результатом работы высококлассных специалистов. Поэтому с этого момента нашей задачей станет изучение самих классов и решений на основе этих классов.
Рассматривайте классы, как удобные инструменты (настоящие — молоток, дрель, пила) для постройки дома. Как вы понимаете, просто набор инструментов не построит дом — дом нужно придумать, нарисовать, рассчитать. Для этого надо понимать на каких принципах он строится — надо знать что такое проектирование.
В процессе создания программ проектирование не менее важный элемент — надо иметь опыт и знания для создания классов, умение использовать уже существующие классы, понимать как надо строить программу или при нестандартных задачах уметь придумывать отношения между классами.
Итак, мы проговорили два очень важных момента:

1. Надо иметь информацию об уже существующих классах. Их много — десятки тысяч. Понятно. что все помнить невозможно и все уважающие себя пакеты (библиотеки) содержат описание которое обычно называется Application Programming Interface (API) — интерфейс прикладного программирования (может быть и другой перевод). Например для Java API — Java Platform, Standard Edition 7 API Specification. Читать такую документацию непросто и часто она содержит просто описание методов и полей, но не содержит инструкцию как использовать этот класс. Но если вы знаете, что именно этот класс вам нужен, то API очень важен
2. Надо знать, когда какой класс можно и нужно использовать. Некоторые варианты использования уже можно назвать стандартными, в некоторых случаях надо подумать, как и что можно сделать.

Именно этим мы и займемся в дальнейшем — мы будем рассматривать стандартные задачи, с которыми сталкивается программист, нередко затрагивать технологии, которые не связаны с Java, но на основе которых строятся Java-классы и принципы построения того или иного куска программы.
Еще раз выстроим нашу цепочку:

Задача -> Классы для решения -> Построения взаимодействия классов для решения задачи.

Как видите, все начинается с задачи, с вопроса «Что я хочу получить?». Именно от этого и надо всегда отталкиваться. Иногда можно просто изучить тот или иной класс, но в подавляющем большинстве случаев даже опытные программисты изучают классы с точки зрения «А что за задачу позволит мне решить этот новый набор классов?».
В принципе вторая и третья задача идут часто параллельно, и даже меняются местами — в случае, если нужных классов просто не существует. Не применяйте выше приведенный порядок как догму — решайте задачу. Это самое важное. Вы можете решить ее не совсем оптимально, может не так красиво — но решение надо найти. Потом его можно (и нужно) улучшать — это и будет вашим опытом.
Когда нужных классов нет, тогда задачей программиста становится написание новых классов. Более частым вариантом бывает улучшение существующего набора классов. Т.е. набор классов есть. но они не совсем то, что надо — тогда программист создает свои, но на основе уже существующих.

В этом разделе я хотел бы познакомить вас с некоторым набором очень популярных классов, который надо знать для самых элементарных задач. Перечислю их сразу, а потом поговорим о каждом в отдельности:

- Object
- String, StringBuilder, StringBuffer
- Date, SimpleDateFormat
- Calendar
- Классы-обертки для чисел

Object

Самое важное в классе **Object** это то, что этот класс является самым первым во всей иерархии классов Java. Даже если вы не указываете, что наследуете класс от кого-либо, то на самом деле вы наследуете **Object**. В связи с этим на мой взгляд крайне важно иметь представление о методах, которые предоставляет этот класс.
Предлагаю прямо сейчас открыть эту ссылку — Class Object и начать просмотр всех методов. Их на самом деле не так уж и много. К тому же не все мы сразу и рассмотрим — некоторые касаются многопоточного программирования, до которого мы пока не добрались.

clone

Метод предназначен для создания копии объекта, у которого вы вызываете этот метод. Еще раз вспомним, что наши переменные на объекты — это ссылки. Не реальные объекты, а ссылки, которые указывают на объекты, которые создаются с помощью ключевого слова **new**. Я бы хотел, чтобы вы это очень хорошо усвоили. Подкрепим это примером.

```
1 Date o1 = new Date();
2 Date o2 = o1;
3
4 o1.setYear(2012);
```

В данном случае у нас две ссылки — o1, o2. И ОБЕ ЭТИ ССЫЛКИ УКАЗЫВАЮТ НА ОДИН ОБЪЕКТ типа Date. Еще раз подчеркиваю — ОБЕ ссылки. Если мы обратимся к полям объекта через ссылку o1, то при обращении к полям через ссылку o2 мы увидим наши новые значения. Это указано в третьей строке кода. (Учтите, что в реальности метод setYear уже устарел — я его привел только для наглядности примера).
clone должен создавать новую копию объекта. Именно копию — со всеми значениями полей. Если взять ту же дату, то метод должен создать новый объект и скопировать все значения из первого объекта во второй. Т.е. нижеприведенный код уже будет иметь другой результат — у нас будет два объекта и изменения поля одного объекта уже не затронет поля другого

```
1 Date o1 = new Date();
2 Date o2 = (Date) o1.clone();
3
4 o1.setYear(2012);
```

Обратите внимание также на то, что метод возвращает тип **Object** и нам приходится приводить значение типа. Метод, как вы заметили, имеет доступ protected, а значит его надо переопределять во всех предках. Для нашего примера класс Date уже имеет такой метод.
Есть момент, на который следует обратить внимание. Когда у вас класс содержит поля простых типов, то создание копии не вызывает вопросов. Но что делать, когда ваш объект содержит ссылки на другие объекты ? Например класс «Автомобиль» может включать ссылки на классы «двигатель», «колесо», «водитель». Что делать в этом случае ? Когда я только знакомился с Java, я подумал, что надо создавать полную копию всех связанных объектов. Но потом задумался. Даже если все эти классы имеют методы **clone** (переопределенные для каждого класса — двигатель, колесо, водитель), то далеко не факт, что создание клона для каждого будет правильным решением. Например, добавим автомобилю объект «Владелец». По идее у

одного владельца может быть несколько автомобилей. Но если мы «клонируем» автомобиль. то владельца тоже надо «клонировать» ? Конечно же не надо. Итак, мы получили ответ на вопрос «надо ли всегда клонировать все связанные объекты» — нет, не надо. Но вот КОГДА надо — это будет зависеть от вашей конкретной задачи.

toString

Как вы возможно уже догадались, этот метод превращает ваш объект в строку. У класса **Object** этот метод возвращает не очень интересную строку, что-то вроде такой:

java.lang.Object@5984cf08

Но что важно отметить — если метод есть у «прародителя» всех объектов, то этот метод есть всегда. Значит любой объект может быть представлен в текстовом виде. Что очень удобно. Например, если вы набираете такой код:

```
1 Date d = new Date();
2 System.out.println(d);
```

то на самом деле вы при печати объекта вызываете его метод toString(). Т.е. вот так:

```
1 Date d = new Date();
2 System.out.println(d.toString());   // Вот оно че, Михалыч :)
```

Метод toString удобно переопределять так, чтобы он возвращал удобный вид значений полей вашего объекта. Как Вам это удобно — так и возвращайте. Это мое мнение. Тот же NetBeans позволяет сгенерировать метод toString в достаточно удобном формате.

finalize

Метод finalize вызывается тогда, когда ваш объект уничтожается сборщиком мусора (я упоминал о нем в разделе ООП). Т.е. если вам очень важно совершить какие-то действия перед уничтожением объекта — переопределяйте этот метод.

getClass

В Java существует понятие **Reflection**. В двух словах — это механизм, который позволяет узнать методы, поля, тип объекта внутри кода. Это понятие очень востребовано в Java, так что обязательно изучите его — как только я смогу, обязательнро напишу про него. Так вот метод возвращает объект

типа **Class** (вот такой вот класс — Class) у которого как раз и можно все это спросить. Больше пока ничего говорить не буду — придет время и мы все разберем более подробно.

hashCode

Метод возвращает цифровое значение, которое можно назвать «цифровым слепком» конкретного объекта. Чтобы не сравнивать все поля объекта, можно сравнивать только их хешкоды. На основе хешкода создано большое количество алгоритмов.

equals

Метод предназначен для того, чтобы ответить на вопрос — эти два объекта равны ? Если метод возвращает true — значит равны. Для примера можно посмотреть реализацию метода equals для класса String.

```
    public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        // Вызов instanceof определяет - является ли объект экземпляром типа String (или другого класса)
        if (anObject instanceof String) {
            String anotherString = (String) anObject;
            int n = value.length;
            if (n == anotherString.value.length) {
                char v1[] = value;
                char v2[] = anotherString.value;
                int i = 0;
                while (n-- != 0) {
                    if (v1[i] != v2[i])
                            return false;
                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

Что КРАЙНЕ важно отметить — если два объекта равны, то у них должен быть одинаковый результат при вызове метода **hashCode**. Но при равных хешкодах объекты совершенно не обязательно должны быть равны, т.е. метод **equals** может вернуть **false**.
Остальные методы, как я уже говорил, предназначены для многопоточного программирования и мы их затронем в свое время.

Класс **Object** играет важную роль, не забывайте про его методы, которые мы перечислили. В тоже время он не столь заметен на практике. Я бы сравнил его с воздухом — все привыкли, что он есть и не сильно обращают на него внимание. Но когда его нет — очень плохо.

String, StringBuilder, StringBuffer

Строковое представление информации наверно самое распространенное в сегодняшнем мире. Поэтому класс, который позволяет хранить и обрабатывать такого рода данные, крайне важен. Поэтому обратите внимание на этот класс — **String**.
перечислять все интересные методы в этом классе наверно бесполезно — проще зайти на страницу с его описанием java.leng.String и посмотреть. Там есть возможность получать часть строки, осуществлять поиск подстроки, делать замены, преобразовывать регистр. Одним словом — замечательный класс. Я крайне настоятельно совету. вам просмотреть все методы, которые есть у этого класса и использовать их в своих задачах.
Но есть одна особенность, которая должна быть у вас в голове при разработке — объекты класса String НЕ МОДИФИЦИРУЮТСЯ (immutable). Т.е. каждая операция, которая как будто меняет строку, на самом деле создает новую. Отметим этот момент — он достаточно важен. Класс String относится к так называемым immutable (не модифицируемым). Объекты этого класса нельзя изменить. Есть реальные причины, почему это сделано. Например, по тем же соображениям безопасности.
Но это порождает проблему — при большом количестве модификаций производительность крайне низкая. Рассмотрим достаточно несложный пример — в цикле будет производится модификация строки путем прибавления к ней нескольких символов.

```
1  package edu.javacourse;
2
3  import java.util.Date;
4
5  public class TestString
6  {
7      public static void main(String[] args) {
8          String s = "1";
9          System.out.println(new Date());
10         for(int i=0; i<100000; i++) {
11             s += "" + i;
12         }
13         System.out.println(new Date());
14     }
15 }
```

Если мы запустим это пример, то скорость его выполнения вас не обрадует. На моей машине конкатенация (соединение, складывание) строк занимает аж 40 секунд.
Для сложных и многократных операция со строками используйте другие классы — StringBuilder и StringBuffer. Например скорость работы предыдущего примера с классом StringBuilder выглядит вот так:

```java
package edu.javacourse;

import java.util.Date;

public class TestString {

    public static void main(String[] args) {
        StringBuilder s = new StringBuilder("1");
        System.out.println(System.currentTimeMillis());
        for(int i=0; i<100000; i++) {
            s.append("" + i);
        }
        System.out.println(System.currentTimeMillis());
    }

}
```

и работает доли секунды — на моей машине порядка 50 миллисекунд. Я упоминал два класса — StringBuilder и StringBuffer. Оба обладают достаточно богатым набором функций для модификации строк — в этом случае объекты действительно модифицируются. В чем же разница ? Разница в том, что класс StringBuilder не является потокобезопасным (threadsafe). Мы пока не касались вопросов многопоточности (это очень сложная и интересная тема), но в двух словах можно сказать так — при попытке одновременного изменения объекта из нескольких потоков объект StringBuilder не гарантирует корректного результата. Если на бытовом уровне — если два человека одновременно пытаются собирать один кубик Рубика, то ничего хорошего не получится. Ели же они это делают последовательно, соблюдая очередность, то результат может быть получен, правда не так быстро. Так вот класс StringBuilder — это класс для одного потока и если вам требуется по каким-то причинам модифицировать его из нескольких потоков, то вам надо использовать класс StringBuffer.
Вопрос — какой класс из этих двух использовать — решается достаточно просто. Если поток один, то StringBuilder. За счет игнорирования потокобезопасности он работает очень быстро. Но если вам нужно модифицировать объект из разных потоков, то используйте StringBuffer.

Классы-обертки

Классы-обертки — это классы, которые представляют собой механизм работы с числами (int, long, double) как с объектами. Если вы уловили идею, то дальше можно не читать. Просто посмотрите список этих классов — Integer, Long, Double, Float, Character, Boolean (не уверен, что перечислил все классы).
Появление этих классов обусловлено в первую очередь тем, что часто даже для работы с числами необходимо, чтобы эти числа были объектами. Например те же коллекции не работают с числами — они работают с объектами. До Java 1.5 приходилось писать явное создание объекта из числа. Например так:

Integer d = new Integer(10);

Теперь это не требуется, можно написать вот так:

Integer d = 10;

В Java 1.5 появилось такое понятие как Autoboxing/Unboxing — автоматическое преобразование чисел в объекты и обратно, когда это требуется. Также надо учитывать, что подобно String все эти объекты immutabel — неизменяемые. В общем это все, что я хотел бы рассказать о классах-обертках.

И теперь нас ждет следующая статья: Список контактов — начало.

7 comments to *Решения на основе классов*

- 
  Январь 16, 2015 at 12:34
  dietmar says:

  Откуда взялось value:
  int n = value.length;
  в последнем примере?

  или это тоже самое что:
  int n = this.value.length;
  ?

  Reply

  - 
    Январь 16, 2015 at 12:39
    admin says:

    Честно говоря я не особо вдавался в подробности этого метода — просто скопировал из иходников.

    Reply

- 

Январь 16, 2015 at 12:48
dietmar says:

А нашел в описании класса:

/** The value is used for character storage. */
private final char value[];

Reply

- 
Сентябрь 19, 2015 at 16:24
Grif says:

«Если взять ту же дату, то ПО метод должен создать новый объект и скопировать все значения из первого объекта во второй. Т.е.»

Лишнее слово «ПО»

Reply

- 
Октябрь 3, 2016 at 01:14
Лера says:

if (anObject instanceof String) {
String anotherString = (String) anObject;
}
Здравствуйте,
Такой вот вопрос: если мы определяем принадлежность объекта определенному классу, то зачем на в дальнейшем его явно приводить к проверяемому типу, раз он и так ему принадлежит?

Reply

  - 
Октябрь 3, 2016 at 09:45

admin says:

Чтобы ссылка теперь указывала не на Object, а на нужный нам класс с его методами. Иначе вызывать метод или обратиться к полу не получится — у Object ведь таких нет.

Reply

- 

  Октябрь 3, 2016 at 17:01
  Лера says:

  Спасибо, теперь все встало на место

  Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

восемь − 5 =

Add comment