aoovcharenko Update final.md Latest commit 6a7ee69 on Apr 7, 2020 History

1 contributor

42 lines (26 sloc) | 2.89 KB Raw Blame

final

В Java существует специальное ключевое слово `final`, у которого сразу несколько предназначений:

1. `final` на уровне класса (например, `public final class String`) означает, что автор класса решил, что вы не можете наследоваться от этого класса.
2. `final` на уровне метода (например, `public final native Class<?> getClass()`) означает, что автор класса решил, что вы не можете переопределить этот метод при наследовании.
3. `final` на уровне полей и переменных* означает возможность инициализировать что-то (присвоить) всего один раз.

Примечание*: на самом деле `final` можно писать и перед параметром метода (в объявлении), но это достаточно редкая практика.

Мы не рекомендуем вам использовать 1-2 варианты и рассмотрим более подробно третий вариант.

Эмуляция констант

Рассмотрим эталонный пример - класс `Math`:

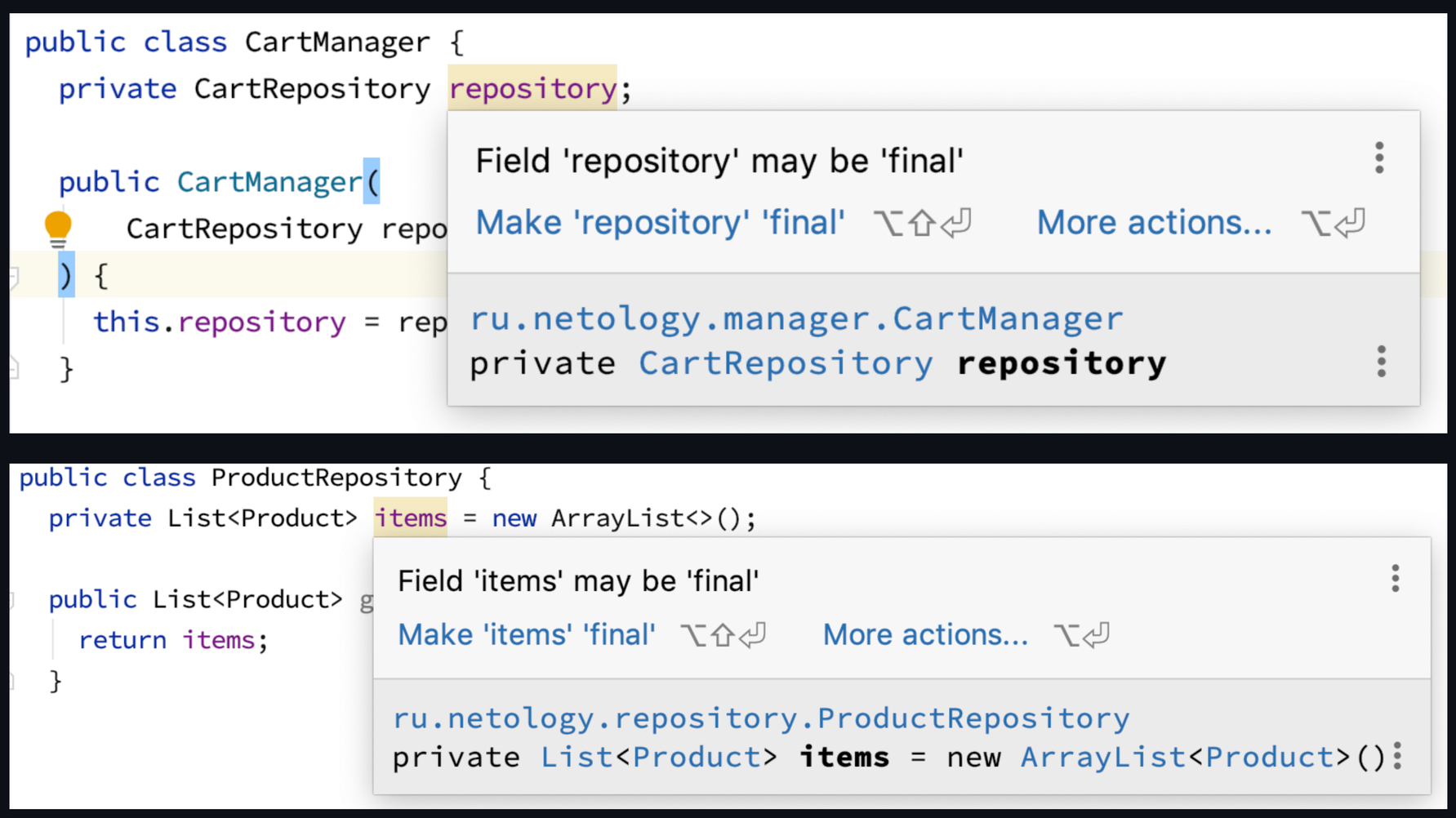
```
public final class Math {
    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
    ...
}
```

В данном случае `final` говорит, что поля могут быть инициализированы только один раз: т.е. только один раз к имени `PI` будет привязано значение.

А поскольку значение - примитив, то и поменять мы его не сможем.

`final` поля

Достаточно часто `final` используют для зависимостей, инициализируемых либо через инициализаторы полей, либо через конструкторы:



Используется это чаще всего для того, чтобы вы "случайно" не перезаписали это поле.

`final` переменные

В случае с `final` переменными ситуация та же, что и с полями - используют это для того, чтобы "случайно" не перетереть значение переменной.

Использовать эту практику или нет, целиком зависит от стиля кодирования, который будет принят в вашей команде.

Generic'и и примитивы

Generic'и в Java работают только с объектами, поэтому, вот этот код является невалидным:

```
List<int> numbers = new ArrayList<>();
```

Что делать, если нам по каким-то причинам всё-таки нужно хранить примитивы?

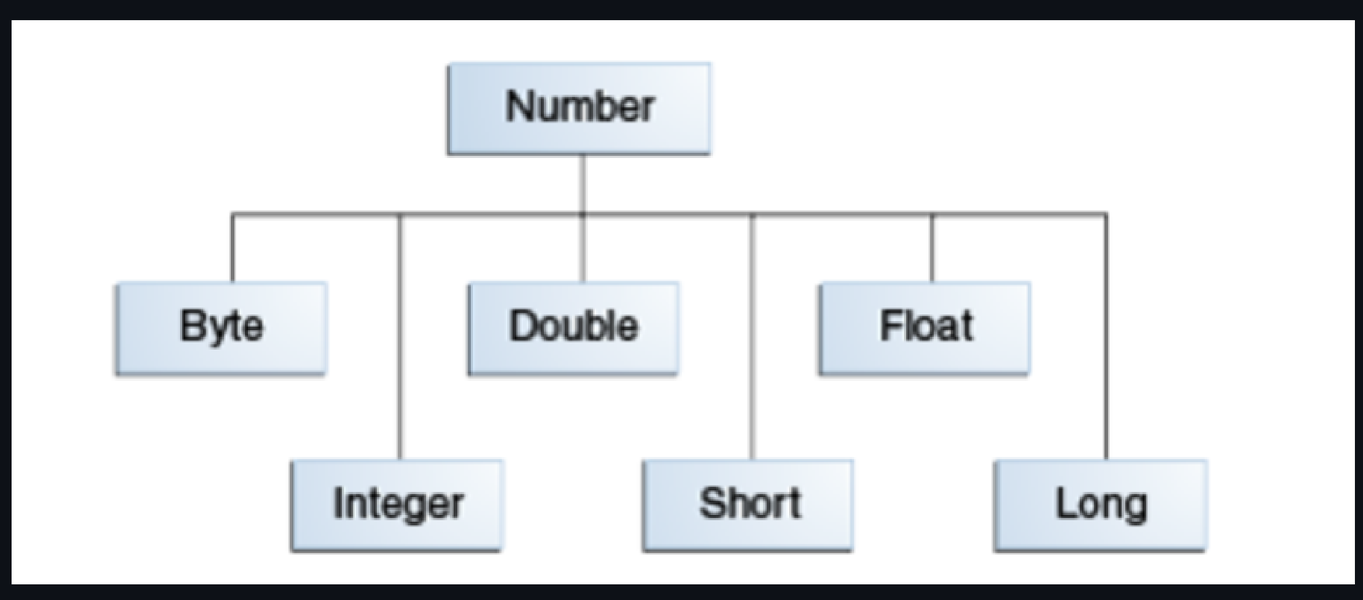
Для этого в Java придумали Wrapper'ы (классы-обёртки) и автоматические операции boxing и unboxing.

О чём идет речь?

В Java спроектировали классы, объекты которых "заворачивают" (wrap) примитивные типы:

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Для чисел эти классы образуют иерархию:



Как использовать

На самом деле, всё достаточно просто и прозрачно: в generic'ах вы пишете классы обёртки:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1); // Java за нас "завернёт" примитив в объект (boxing)
int first = numbers.get(0); // Java за нас "развернёт" объект в примитив (unboxing)
```

Иногда возникает соблазн везде вместо примитивов использовать объекты классов обёрток. Здесь всё зависит от стиля, принятого в команде, но есть два ключевых нюанса:

1. В переменные/параметры и поля примитивного типа нельзя положить `null`, а объектного - можно*
2. Любые объектные типы нужно сравнивать через `equals` **

Примечание*: попробуйте ради интереса выполнить следующий код и разобраться (по исходникам класса `Integer`), почему так происходит:

```
Integer wrapper = null;
int primitive = 0;
System.out.println(wrapper + primitive);
```

Примечание**: попробуйте ради интереса выполнить следующий код:

```
Integer a = 10;
Integer b = 10;
System.out.println(a == b); // true***
System.out.println(a.equals(b)); // true
```

```
Integer a = 1024;
Integer b = 1024;
System.out.println(a == b); // false***
System.out.println(a.equals(b)); // true
```

Примечание**: на самом деле результат выполнения этого кода будет зависеть от настроек JVM (здесь указан для дефолтных)ю

Полезности

Кроме всего прочего, классы-обёртки предоставляют полезные методы для работы с объектами примитивного типа (например, преобразования в строку или парсинг из строки).

Поэтому, настоятельно рекомендуем вам посмотреть на список их методов и почитать JavaDoc'и.

Reflection

Тема рефлексии может показаться сложной, но общее представление о ней иметь нужно, т.к. на этом строится большая часть фреймворков (включая JUnit).

Рефлексия - API, позволяющих анализировать приложение в момент исполнения.

Рефлексия начинается с того, что у нас есть класс, который называется `Class`.

Объекты этого класса описывают классы, загруженные в память JVM.

Например, у вас есть класс `Product`. Когда JVM загружает этот класс из байт-кода, то она создаёт объект класса `Class`, который и описывает класс `Product`.

Что значит описывает? Это значит смотрит на него примерно как мы с вами:

1. Название
2. Набор полей
3. Набор методов
4. и т.д.

Каждый объект знает, к какому классу он принадлежит. Когда IDEA генерирует `equals` вот эта строка и проверяет, что объект относятся к одному и тому же классу: `if (o == null || getClass() != o.getClass()) return false;`.

Зачем это нужно? Рефлексия позволяет нам буквально в цикле перебирать поля и методы, выполняя необходимые действия:

- JUnit смотрит, написано ли над методом `@Test`
- Mockito смотрит, написано ли над полем `@Mock`, `@InjectMocks`
- и т.д.

И самое главное - благодаря рефлексии JUnit, Mockito и другие инструменты могут создавать объекты прямо во время исполнения программы.

Пародия на JUnit

Это часть "продвинутая", ничего страшного, если вы её опустите или оставите до лучших времён. Она не является критичной и необходимой для понимания остальной части курса и написана только для того, чтобы интересующиеся могли получить базовую информацию.

Итак задача: мы хотим написать некоторую пародию на JUnit, которая:

1. Берёт класс
2. Анализирует все его методы
3. Находит все, над которыми стоит аннотация `@Test`
4. Создаёт для каждого такого метода новый объект и запускает на нём этот метод (тот, над которым стояло `@Test`)

Естественно, мы для простоты изложения опустим кучу нюансов и продемонстрируем основную идею. Если вам будут интересны детали, пишите в Slack-чат вопросы, мы обязательно ответим.

Итак, поехали (наш подопытный)*:

```
package ru.netology;

import org.junit.jupiter.api.Test;

public class DemoTest {
  @Test
  public void shouldBeCalled() {
    System.out.println("Test method called");
  }

  public void shouldNotBeCalled() {
    System.out.println("Invalid method called");
  }
}
```

Примечание*: целиком код проекта вы можете найти в репозитории с кодом к лекциям в проекте `reflection-sample`.

Создаём "запускалку" наших тестов:

```
package ru.netology;

import org.junit.jupiter.api.Test;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Launcher {
  // что такое исключения мы разберём на следующей лекции
  public static void main(String[] args) throws IllegalAccessException, InstantiationException, InvocationTargetException {
    // создаём объект типа `Class` (generic'и мы пока не знаем, поэтому и так "сойдёт")
    // clazz или cls - общепринятое имя, т.к. class - зарезервировано
    Class clazz = DemoTest.class;
    // берём все методы класса
    Method[] methods = clazz.getDeclaredMethods();
    // перебираем методы
    for (Method method : methods) {
      // смотрим, есть ли над методом аннотация @Test
      if (method.isAnnotationPresent(Test.class)) {
        // создаём объект класса (newInstance помечен аннотацией @Deprecated, но для простоты мы будем использовать его, в противном с
        Object object = clazz.newInstance();
        // вызываем метод на объекте
        method.invoke(object);
      }
    }
  }
}
```

Как вы видите, код "не особо приятный", и, в большинстве случаев, вы такой код писать не будете, если не станете сами разрабатывать библиотеки и инструменты (т.к. инструменты тестирования уже это делают за вас).

Но свою работу он выполняет, вы можете запустить, подебажить, посмотреть, как он работает.

Ключевое: вы должны понимать, что Java предоставляет нам инструменты манипулирования кодом, которые и позволяют создавать такие мощные вещи как JUnit и Mockito*.

Примечание*: на самом деле мы немного лукавим, т.к. Mockito использует ещё и библиотеку ByteBuddy, которая генерирует байт-код на лету (но это уже совсем другая история).

static

В Java существует специальное ключевое слово `static`, позволяющее указать, что конкретное поле или метод* принадлежит не объекту, а классу.

Примечание*: на самом деле `static` могут быть не только поля и классы, но пока мы будем рассматривать только их.

Что значит "не объекту, а классу"?

Это значит, что для использования этого поля или метода не нужно создавать объект.

С одной стороны это может показать "неправильным", т.к. мы с вами говорили, что весь мир моделируется в виде объектов и взаимодействия между ними.

Но как решили, так решили.

Предназначение `static` в рассматриваемых нами случаях можно описать следующим образом:

- для полей - это чаще всего попытка эмуляции констант, например `PI` или `E`
- для методов - это вспомогательные методы (так называемые хелперы), либо методы, которым не нужен объект для своей работы

Рассмотрим эталонный пример - класс `Math`:

```
public final class Math {
    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    public static final double E = 2.7182818284590452354;
    public static final double PI = 3.14159265358979323846;
    ...
}
```

Помимо этих двух констант (что такое `final` мы разберём на следующих лекциях), этот класс содержит математические функции:

```
public final class Math {
    ...
    public static double sin(double a) {
        ...
    }

    public static double cos(double a) {
        ...
    }
    ...
}
```

Использовать `static` поля и методы можно ставя оператор `.` после имени класса, например:

```
package ru.netology;

public class Main {
  public static void main(String[] args) {
    double result = Math.sin(Math.PI / 2);
    System.out.println(result);
  }
}
```

Это всё здорово, но в тестировании нам такое нужно не часто. Зачем же нам `static` 'и?

Assertions

На самом деле, они нам очень нужны, т.к. все используемые нами assert'ы - это `static` методы класса `Assertions`:

```
public class Assertions {
    ...
    public static void assertTrue(boolean condition) {
        ...
    }
    ...
        public static void assertFalse(boolean condition) {
        ...
        }
}
```

Это позволяет нам не создавать объект из этого класса (тем более конструктор у него `protected`), а просто использовать нужные нам методы:

```
package ru.netology;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class SampleTest {
  @Test
  public void shouldPass() {
    Assertions.assertTrue(true);
  }
}
```

Но раньше же мы не писали `Assertions`, а просто писали `assertTrue`?

`static` import

В Java есть специальный механизм, который позволяет нам использовать статические поля и методы без указания класса.

Для этого используется специальный static import:

```
package ru.netology;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue; // <- static import

class SampleTest {
  @Test
  public void shouldPass() {
    assertTrue(true); // <- теперь можно не писать Assertions.
  }
}
```

Когда вы используете много методов (а не один), то обычно static import превращается в import со * (что означает все статические поля и методы):

```
package ru.netology;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*; // <- static import

class SampleTest {
  @Test
  public void shouldPass() {
    assertTrue(true); // <- теперь можно не писать Assertions.
  }
}
```

static import повсеместно используется в тестировании (не только в JUnit), поэтому привыкайте к этому. Если вы видите где-то вызов метода, но перед методом ничего нет, то с большой вероятностью, это статический метод, импортированный с помощью static import.

Утилитные классы

Есть общепринятое соглашение (которому, к сожалению, следуют не все), что классы, которые содержат **только** статические методы, называть утилитными и использовать специальный формат оформления:

1. Класс пишется во множественном числе и оканчивается на `s`, например `Assertions`, `Collections`, `Arrays`
2. Класс не имеет публичного конструктора (во избежание создания объектов этого класса)

Мы рекомендуем вам придерживаться этого соглашения (если вы собираетесь разрабатывать подобные классы).