Модуль 2. Объектно-ориентированное программирование

Тема 2.2. Конструкторы и их перегрузка. Статические поля и методы

4 часа

Оглавление

2.2. Конструкторы и их перегрузка. Статические поля и методы

2.2.1. Конструкторы

Для работы с объектами недостаточно просто объявить переменную определенного класса. Ее также нужно создать. Во время создания объекту отводится память для хранения полей объектов и ссылок на методы. При этом поля объекта автоматически инициализируются значениями по умолчанию, принятыми в Java. Чтобы инициализировать объект по своим правилам используют специальные функции - конструкторы. Конструктор в Java обладает следующими свойствами:

- обязательно имеет имя, совпадающее с названием класса;
- вызывается при создании объекта;
- в отличие от метода не имеет явным образом определённого типа возвращаемых данных и не наследуется.

Опишем простейший класс натуральной дроби, состоящий из двух полей: числитель и знаменатель.

```
//Класс "Натуральная дробь"
public class Fraction {
	int numerator; // Числитель
	int denominator; // Знаменатель
}
```

Но где же конструктор? На самом деле, даже если конструктор в классе не описан явно, он есть. Компилятор автоматически создает конструктор, не имеющий параметров. Он то и проинициализирует поля класса значениями по умолчанию. Такой конструктор называется "**конструктором по умолчанию**".

Конструктор по умолчанию задаст знаменателю значение 0 (значение по умолчанию для типа int), а нас такой вариант не устраивает, потому что мы можем получить ошибку деления на ноль. Опишем свой конструктор, который присвоит знаменателю единицу.

```
//Класс "Натуральная дробь"
public class Fraction {
	int numerator; // Числитель
	int denominator; // Знаменатель

	public Fraction() {
		denominator = 1;
	}
}
```

Такой конструктор очень примитивный и вполне закономерно замечание, что тот же функционал можно получить, если инициализацию знаменателя совместить с его же объявлением, например, так:

```
public class Fraction {
	int numerator;// Числитель
	int denominator = 1;// Знаменатель
}
```

В этом случае необходимость собственного конструктора отпадет.

Однако, если необходимо предоставить возможность совместить создание объекта и инициализацию его полей собственными значениями, то тогда используют **конструктор с параметрами**. Создадим такой конструктор и не забудем проверку на недопустимость нулевого знаменателя:

```
public class Fraction {
	int numerator;// Числитель
	int denominator = 1;// Знаменатель

	public Fraction(int num, int denom) {
		if (denom == 0) {
			System.out.println("Denominator can't be zero. Choose
another one.");
			return;
		}
		numerator = num;
		denominator = denom;
	}
}
```

Оператор return также, как в методе, завершает выполнение конструктора. Как правило, его используют для прекращения инициализации объекта в нештатных ситуациях

Чтобы не произошло конфликта имен переменных, нам пришлось назвать параметры другими именами. Но в Java есть способ избежать такой конфликт с использованием ключевого слова this. Это ключевое слово ссылается на объект текущего класса и обращаться к полям или методам класса можно через конструкцию *this.название_поля* (подробнее см. в 2.2.3).

```
public class Fraction {
	int numerator;// Числитель
	int denominator = 1;// Знаменатель

	public Fraction(int numerator, int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose
another one.");
			return;
		}
```

```
            this.numerator = numerator;
            this.denominator = denominator;
        }
}
```

В приведенном примере, видим, что, используя *this*, можно легко различить к чему мы обращаемся: к одноименному параметру или полю объекта.

Класс может иметь несколько конструкторов, различающихся количеством и типами параметров. Это возможно, благодаря приему, который в программировании называется **перегрузкой.**

Для класса, в котором определён конструктор с параметрами, конструктор по умолчанию создаваться автоматически не будет. Если необходим конструктор без параметров - напишите его.

Конструкторы имеют следующие особенности:

- не могут напрямую вызываться (необходимо использовать ключевое слово *new*);
- вызываются только один раз - при создании объекта в памяти;
- называются так же, как называется класс;
- им нельзя задать возвращаемое значение (всегда возвращают this).

Как вы заметили, инициализировать поля можно в месте объявления (в нашем примере denominator устанавливается в единицу) либо в конструкторе. В конструкторе, как правило, инициализируют поля, без которых существование объекта не имело бы смысла.

Вернемся к нашему классу. Конструктор дроби защищен от неправильного присвоения значения знаменателю при создании объекта, но ничто не мешает изменить его на ноль после создания объекта. Исправить это нам поможет один из принципов ООП - инкапсуляция.

2.2.2. Перегрузка методов

Java позволяет создавать несколько методов с одинаковыми именами, но другим набором параметров. Это называется **перегрузкой**.

Метод называется **перегруженным**, если существует несколько его версий с одним и тем же именем, но с различным списком параметров (сигнатурой).

Сигнатура метода в Java - это совокупность имени метода с набором параметров. Т.е. тип возвращаемого значения не входит в сигнатуру, а порядок следования параметров и их типы - входит.

```
void pr(double a) {
  System.out.println(a);
}
// 1-я перегрузка
void pr(String a) {
  System.out.println(a);
}
// 2-я перегрузка
void pr(int[] a) {
  for (int i=0; i<a.length; i++) {
```

```
      System.out.print(a[i]+" ")
  }
  System.out.println();
}
```

Пример использования метода:

```
int a = 5;
int [] m = {1, 2, 8, 3}
String s = "Мир";
pr(a) //работает исходный метод
pr(a + s); //сработает 1-я перегрузка и будет выведено:5 Мир
pr(m); //сработает 2-я перегрузка и будет выведено:1 2 8 3
pr(m + a); // ошибка
```

В приведенном примере при вызове `pr(a)` компилятор сначала будет искать метод с полным совпадением сигнатуры, т.е. метод `pr()`, у которого 1 параметр типа `int`. Не найдя такой, компилятор ищет перегруженный метод с совпадающим количеством параметров и совместимыми по преобразованию типами. В нашем примере переменная *a* не относится к типу *double*, но поскольку возможно автоприведение типа *int* в *double*, то запускается метод с сигнатурой `void pr(double a)`.

Сработало ли бы это в обратной ситуации? Если бы в сигнатуре метода был определен параметр типа *int,* а в вызове был фактический параметр типа *double?* Нам известно, что тип *double* не может быть автоматически преобразован в *int*. Значит это было бы невозможно!

Перегрузка реализует «раннее связывание» (объект и вызов метода связываются между собой на этапе компиляции). В данной теме мы ограничимся рассмотрением перегрузки в рамках одного класса. Перегрузка методов из разных классов будет рассмотрена в следующих темах.

Статические методы могут перегружаться нестатическими и наоборот – без ограничений. При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод.

Пример вызова перегруженных методов:

```
package chapt1;

public class NumberInfo {
	public static void viewNum(Integer i) {// 1
		System.out.printf("Integer=%d%n", i);
	}

	public static void viewNum(int i) {// 2
		System.out.printf("int=%d%n", i);
	}

	public static void viewNum(Float f) {// 3
		System.out.printf("Float=%.4f%n", f);
	}

	public static void viewNum(Number n) {// 4
		System.out.println("Number=" + n);
```

```
    }

    public static void main(String[] args) {
        Number[] num = { new Integer(7), 71, 3.14f, 7.2 };
        for (Number n : num)
            viewNum(n);
        viewNum(new Integer(8));
        viewNum(81);
        viewNum(4.14f);
        viewNum(8.2);
    }
}
```

Может показаться, что в результате компиляции и выполнения данного кода будут последовательно вызваны все четыре метода, однако в консоль будет выведено:

```
Number=7
Number=71
Number=3.14
Number=7.2
Integer=8
int=81
Float=4,1400
Number=8.2
```

То есть, во всех случаях при передаче в метод элементов массива был вызван четвертый метод. Это произошло вследствие того, что выбор варианта перегруженного метода происходит на этапе компиляции и зависит от типа массива *num*. То, что на этапе выполнения в метод передается другой тип (для первых трех элементов массива), не имеет никакого значения, так как выбор уже был осуществлен заранее.

При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции. С одной стороны, этот механизм снижает гибкость, с другой – все возможные ошибки при обращении к перегруженным методам отслеживаются на этапе компиляции, в отличие от переопределенных методов, когда их некорректный вызов приводит к возникновению исключений на этапе выполнения.

При перегрузке рекомендуется придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- заменять при возможности перегруженные методы на несколько разных методов.
- избегайте произвольного изменения имен параметров в перегрузках. Если параметр в одной перегрузке представляет, то же входное значение, что и параметр в другой перегрузке, параметры должны иметь одинаковые имена.
- будьте последовательны при упорядочении параметров в перегружаемых членах. Параметры с одинаковыми именами должны находиться во всех перегрузках на одном и том же месте.

2.2.3. Ключевое слово this

Если вы имеете два объекта одного и того же типа, с именами a и b, вы можете задуматься, как вы можете вызвать метод f( ) для этих обоих объектов:

```
class Banana {
      void f(int i) { /* ... */
      }

      public static void main(String[] args) {

            Banana a = new Banana(), b = new Banana();
            a.f(1);
            b.f(2);
      }
}
```

Если есть только один метод с именем *f( )*, как этот метод узнает, был ли он вызван объектом *a* или *b*?

Чтобы позволить вам писать код в последовательном объектно-ориентированном синтаксисе, в котором вы “посылаете сообщения объекту”, компилятор выполняет некоторую скрытую от вас работу. Секрет в первом аргументе, передаваемом методу f( ). Он отражает ссылку на объект, с которым происходит манипуляция. Так что эти два вызова метода, приведенные выше, становятся похожи на следующие вызовы:

```
Banana.f(a,1);
Banana.f(b,2);
```

Это внутренний формат, и вы не можете записать эти выражения и дать компилятору доступ к ним, но это дает вам представление об идее происходящего.

Представим ситуацию: вы находитесь внутри метода класса и хотите обратиться к непосредственно текущему объекту этого класса. Как получить ссылку на этот объект? Для этих целей существует ключевое слово ***this***.

Таким образом ключевое слово *this*, которое может использоваться только внутри метода, содержит ссылку на объект, который вызвал метод. Вы можете трактовать эту ссылку, как и любой другой объект.

Методы одного и того же класса вполне могут вызывать друг друга и без использования *this*. Вполне корректно написать:

```
class Apricot {
      void pick() { /* ... */
      }

      void pit() {
            pick(); /* ... */
      }
}
```

Внутри *pit( )* можно было бы написать *this.pick( )*, но в этом нет необходимости. Компилятор делает это автоматически.

Когда же нужно использовать ключевое слово *this*? Наиболее часто оно используется в инструкции *return*, когда необходимо вернуть явно ссылку на текущий объект:

```
// Использование ключевого слова "this" в return
public class Leaf {
      int i = 0;

      Leaf increment() {
            i++;
            return this;
      }

      void print() {
            System.out.println("i = " + i);
      }

      public static void main(String[] args) {
            Leaf x = new Leaf();
            x.increment().increment().increment().print();
      }
}
```

Поскольку в приведенном примере метод *increment( )* через ключевое слово this возвращает ссылку на текущий объект возможно многократное увеличение поля *i* одного и того же объекта.

Слово *this* часто используется в сеттерах и конструкторах, чтобы отличить поле объекта от параметра метода (если параметр имеет то же имя, что и поле):

```
public void setField(String field) {
      this.field = field;
}
```

Вызов перегруженных конструкторов через this()

При определении перегруженных конструкторов зачастую для того, чтобы не дублировать код также используют ключевое слово this. В этом случае внутри одного конструктора можно вызвать другой. Компилятор, когда встречает вызов конструктора через this ищет соответствующий по сигнатуре перегруженный конструктор и выполняет его. Затем управление переходит на операторы, которые следуют за вызовом this(), т.е. продолжается выполнение исходного конструктора.

Пример:

```
public class Person {
      private String firstName;
      private String lastName;
      private char gender; // m-male, f- female
        // Конструктор 1
      Person() {
```

```
            this("", "",'-'); //Вызывается конструктор 4

      }
      // Конструктор 2
      Person(String lastName) {
            this(lastName, "",'-'); //Вызывается конструктор 4
      }
         // Конструктор 3
      Person(String lastName, String firstName) {
                  this.firstName = firstName;
                  this.lastName = lastName;
      }
         // Конструктор 4
      Person(String lastName, String firstName, char gender) {
                  this(lastName, firstName);//Вызывается конструктор 3
            this.gender = gender;
      }

      public void PrintPerson() {
                  System.out.print(this.lastName + " ");
                  System.out.print(this.firstName + " ");
                  System.out.print(this.gender);
                  System.out.println();
            }

}
```

При этом вызов конструктора this() должен быть первым оператором в конструкторе.

В приведенном примере, если мы захотим, задать значение полю gender до вызова this, следующим образом:

```
      Person(String lastName, String firstName, char gender) {
                  this.gender = gender;
                  this(lastName, firstName);
      }
```

компилятор выдаст ошибку.

2.2.4. Спецификаторы доступа

Инкапсуляция означает, что данные объекта недоступны другим объектам непосредственно. Вместо этого они инкапсулируются — скрываются от прямого доступа извне. Инкапсуляция предохраняет данные объекта от нежелательного доступа, позволяя объекту самому управлять доступом к своим данным.

Поля и методы объекта могут обладать различной степенью открытости (или доступности).

Открытые члены класса составляют внешний интерфейс объекта. Это та функциональность, которая доступна другим классам. Закрытыми обычно объявляются все поля класса, а также вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

Благодаря сокрытию реализации за внешним интерфейсом класса можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы.

Обеспечение доступа к свойствам класса только через его методы также дает ряд преимуществ. Во-первых, так гораздо проще контролировать корректные значения полей, ведь прямое обращение к свойствам отслеживать невозможно, а значит, им могут присвоить некорректные значения.

Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного класса, а не вводить эту функциональность во все части системы.

Наконец, программный код, написанный с использованием данного принципа, легче отлаживать. Для того, чтобы узнать, кто и когда изменил свойство интересующего нас объекта, достаточно добавить вывод отладочной информации в тот метод объекта, посредством которого осуществляется доступ к свойству этого объекта. При использовании прямого доступа к свойствам объектов программисту пришлось бы добавлять вывод отладочной информации во все участки кода, где используется интересующий нас объект.

В языке Java существует 3 модификатора разграничения доступа: ***public***, ***private, protected.*** Если модификатор доступа не указан явно, то подразумевается возможный доступ для всех классов, которые находятся в том же пакете (*package-private*). Также этот модификатор называют модификатором доступа по умолчанию (default).

В таблице сведено применение различных модификаторов к классам, полям, методам, конструкторам и блокам.

Модификаторы	Область доступа	Класс	Поле	Метод	Конструктор	Блок
private	Только из данного класса	-	+	+	+	-
без модификатора	Для всех классов данного пакета	+	+	+	+	+
protected	Классы данного пакета и потомки	-	+	+	+	-
public	Без ограничений	+	+	+	+	-

Уровень доступа *private* используется для сокрытия методов или переменных класса от других классов, т.е. они доступны только внутри класса. Если вспомнить пример с микроволновой печью из предыдущего урока, то ток и напряжение, которые пользователю знать не обязательно - это как

раз переменные с доступом *private*. Если необходимо получить доступ к этим переменным, то определяют внутри класса специальные методы: ***getter*** и ***setter***, которые позволяют включить дополнительную логику перед тем как вернуть либо присвоить значение.

Модификатор *protected* и использование в целом модификаторов к классам будет рассматриваться в теме 2.7 “Наследование”. *protected* используется для определения видимости членов класса в самом классе и в его наследниках.

Модификатор доступа *public* означает, что метод или поле видны и доступны любому классу. В одном файле может быть только один *public* класс.

Когда Вы создаете файл исходного текста в Java, он обычно называется модулем компиляции (иногда модулем трансляции). Каждый модуль компиляции имеет расширение .java, и внутри него должен быть расположен ***только один*** *публичный класс, в противном случае, компилятор выдаст ошибку. Имя этого публичного класса и имя файла должны быть одинаковыми (учитывается регистр, но без расширения .java). Остальные классы в этом модуле компиляции, если они есть, не видны за пределами текущего пакета, т.к. они не публичные, и представляют классы “поддержки” для главного публичного класса.*
Такое ограничение позволяет по названиям файлов легко определить, какие классы в них находятся.

Если выделенные 4 уровня доступа расставить в порядке расширения области доступа, то получится пирамида - каждый нижний слой расширяет предыдущий, добавляя свою логику[1].

Вернемся к нашему классу “Натуральная дробь”: скроем поля от прямого изменения. Для получения и изменения полей создадим свои методы getter и setter (в русской литературе их иногда называют геттеры и сеттеры). Эти методы получили названия от английских слов get и set , т.е. получить либо установить значение.

Среда разработки Eclipse обладает рядом полезных возможностей по генерации кода. Например, можно сгенерировать конструкторы, методы getter и setter. Для этого необходимо нажать правой кнопкой по коду, навести на пункт ***Source*** *и выбрать интересующий* ***Generate****. Как альтернатива - можно нажать на пункт* ***Source*** *главного меню и выбрать необходимую генерацию кода.*

В Intellij IDEA и Android Studio нужно щелкнуть правой кнопкой по коду, нажать ***Generate*** *и в появившемся окне выбрать нужный метод****,*** *или выбрать в главном меню* ***Code -> Generate,*** *также можно воспользоваться горячими клавишами* ***Alt+Insert.***

[1] http://www.quizful.net/post/features-of-the-application-of-modifiers-in-java

```
public class Fraction {
	private int numerator;// Числитель
	private int denominator = 1;// Знаменатель

	public Fraction(int numerator, int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another
one.");
			return;
		}
		this.numerator = numerator;
		this.denominator = denominator;
	}

	public int getNumerator() {
		return numerator;
	}

	public int getDenominator() {
		return denominator;
	}

	public void setNumerator(int numerator) {
		this.numerator = numerator;
	}

	public void setDenominator(int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another
one.");
			return;
		}
		this.denominator = denominator;
	}
}
```

Остался маленький штрих: для того, чтобы общий знак числа не мог появляться и в числителе и в знаменателе, будим хранить знак только в числителе.

```
public class Fraction {
	private int numerator;// Числитель
	private int denominator = 1;// Знаменатель

	public Fraction(int numerator, int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another
one.");
			return;
		}
		// знак храним в числителе
		this.numerator = numerator * (denominator < 0 ? -1 : 1);
		// знаменатель всегда положительный
		this.denominator = Math.abs(denominator);
	}
```

```
	public int getNumerator() {
		return numerator;
	}

	public int getDenominator() {
		return denominator;
	}

	public void setNumerator(int numerator) {
		this.numerator = numerator;
	}

	public void setDenominator(int denominator) {
		if (denominator == 0) {
			System.out.println("Denominator can't be zero. Choose another
one.");
			return;
		}
		if (denominator < 0) {
			this.numerator *= -1;
		}
		this.denominator = Math.abs(denominator);
	}
}
```

Упражнение 2.2.1

Продолжение разработки класса, описывающего рациональную дробь. Реализация конструкторов. Реализация методов экземпляра.

```
class Fraction {
	private int numerator;// Числитель
	private int denominator = 1;// Знаменатель

	public Fraction(int numerator, int denominator) {
	if (denominator == 0) {
		System.out.println("Denominator can't be zero. Choose another one.");
		return;
	}
	// знак храним в числителе
	this.numerator = numerator * (denominator < 0 ? -1 : 1);
	// знаменатель всегда положительный
	this.denominator = Math.abs(denominator);
	Normalization();
	}

	private void Normalization() {
	int n = Nod(Math.abs(numerator), Math.abs(denominator));
	numerator /= n;
	denominator /= n;
	}

	private static int Nod(int numerator, int denominator)
	{
```

```
      while (numerator != 0 && denominator != 0)
            if (numerator > denominator)
                numerator = numerator % denominator;
            else
            denominator = denominator % numerator;
      return numerator + denominator;
      }

      public boolean ProperFraction() {
      return (Math.abs(numerator) < denominator ? true : false);
      }

      public int isIntegerPart() { // выделение целой части
      return (numerator / denominator);
      }

      public Fraction FractionalPart() { // выделение дробной части
      return new Fraction(numerator % denominator, denominator);
      }

      public String ToString() { // вывод дроби на печать
      return new String(numerator + " / " + denominator);
      }

      public double ToDecimalFractions() { // результат деления в виде десятичной
дроби
      return (double) numerator / denominator;
      }

      public int getNumerator() {
      return numerator;
      }

      public int getDenominator() {
      return denominator;
      }

      public void setNumerator(int numerator) {
      this.numerator = numerator;
      }

    public void setDenominator(int denominator) {
      if (denominator == 0) {
            System.out.println("Denominator can't be zero. Choose another one.");
            return;
      }
      if (denominator < 0) {
            this.numerator *= -1;
      }
      this.denominator = Math.abs(denominator);
      }

      public Fraction SumFractionTo(Fraction obj) {

      return new Fraction(this.numerator * obj.denominator + obj.numerator *
this.denominator, this.denominator * obj.denominator);
    }
```

```
      public static Fraction SumFraction(Fraction obj1, Fraction obj2) {

      return new Fraction(obj1.numerator * obj2.denominator + obj2.numerator *
obj1.denominator, obj1.denominator * obj2.denominator);
      }
 }

      public class Main {
      public static PrintStream out = System.out;
      public static Scanner in = new Scanner(System.in);

      public static void main(String[] args) {
   // write your code here

      Fraction objectFraction1 = new Fraction(1,6);
      System.out.println(objectFraction1.ToString());

      Fraction objectFraction2 = new Fraction(1,3);
      System.out.println(objectFraction2.ToString());

      Fraction objectFraction3 = objectFraction1.SumFractionTo(objectFraction2);

      System.out.println(objectFraction3.ToString());

      Fraction                          objectFraction4                          =
Fraction.SumFraction(objectFraction1,objectFraction2);

      System.out.println(objectFraction4.ToString());

      }
}
```

2.2.5. Статические методы класса.

Ключевое слово static

В языке Java модификатор **static** применяется к внутренним классам, методам, полям и логическим блокам. И тогда мы получаем соответственно статические классы, статические методы, статические поля и статические блоки инициализации.

Статические поля

Иногда бывает нужно, чтобы поле класса имело одинаковое значение для всех объектов данного класса и при изменении значения поля все методы видели новое значение. В таком случае это поле нужно объявить с ключевым словом **static** (статический) и оно станет общим для всех экземпляров класса.

В языке Java, в отличие, например, от Паскаля или C, отсутствуют глобальные переменные и константы в привычном смысле. Но так как это остается необходимым, применяют статические поля.

Наиболее часто статические поля используют как константы. Например, библиотечный класс Math имеет статическую константу PI:

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

С одной стороны модификатор доступа public позволяет использовать константу везде, а с другой стороны, добавив слово *final*, мы гарантируем, что это поле невозможно изменить. Применять общедоступные статические поля без final очень опасно, так как любой объект сможет изменить их значения.

Статические члены класса создаются в памяти сразу при загрузке класса, т.е. не нужно создавать экземпляр класса, чтобы их использовать. Обращение происходит через *ИмяКласса.поле* или *ИмяКласса.метод*().

В предыдущем примере в любом месте программы (стоит модификатор public) можно получить значение пи, написав: *Math.PI*.

Необходимо с осторожностью подходить к использованию статических полей по нескольким причинам:

- *они создаются в момент загрузки класса в память и живут до момента завершения работы приложения, тем самым занимая ресурсы;*
- *доступность публичных статических полей усложняет отслеживание изменения их содержимого, что особенно опасно в случае многопоточных программ, а также это нарушение принципа ООП - инкапсуляции.*

Несколько рекомендаций по использованию статических полей со словом public:

- можно использовать в качестве констант (со словом final);
- по возможности не использовать статические поля в качестве глобальных переменных: нарушение принципа инкапсуляции, а также не безопасно в случае многопоточных программ.

Блоки статической инициализации

Класс также может содержать блоки статической инициализации, которые присваивают значения статическим полям или выполняют иную необходимую логику. Статический инициализатор используют, как правило, когда простой инициализации в объявлении статического поля недостаточно. Например, создание статического массива часто должно выполняться одновременно с его инициализацией в операторах программы. Приведем пример инициализации небольшого массива с степенями 10:

```
public class Number{
	static int[] dischargeArr = new int[4]; //статический массив степеней 10

	static {	//статический блок
		dischargeArr[0] = 1;
		for (int i = 1; i < dischargeArr.length; i++)
			dischargeArr[i] = dischargeArr[i - 1] * 10;
	}
```

```
    public static void main(String[] args) {
        for (int i = 0; i < dischargeArr.length; i++)
            System.out.println(dischargeArr[i]);

    }
}
```

Статическая инициализация внутри класса выполняется в порядке слева направо и сверху вниз. В данном примере можно гарантировать, что массив `dischargeArr` будет создан до выполнения статического блока.

Статические методы

Как уже было сказано статический метод можно вызвать, используя тип класса, в котором эти методы описаны, т.е. не нужно каждый раз создавать новый объект для доступа к таким методам. Класс *java.lang.Math* — замечательный пример, в котором почти все методы статичны.

Когда использовать статические методы?

1. *Когда вы создаете класс-утилиту с часто используемыми простыми методами. Например, вы пишете программу, работающую с деньгами и вам часто необходимо в одном и том же формате выводить стоимость. Или вы пишете программу, работающую с файлами и часто приходится показывать размер файла, то можно вынести такой метод в отдельный вспомогательный класс.*
2. *Когда пишите метод, который не зависит от полей своего класса, т.е. работает только с входными параметрами. Например, в классе String есть метод valueOf, который создает строку на основе переданного аргумента.*
3. *Очень часто статические методы используются при логировании. Например, в Андроиде есть клас Log, имеющий статические методы d, w, e и т.д. (см. тему 2.3). Если бы они были не статическими, то каждый раз при необходимости вывода сообщения в лог, пришлось бы создавать (или получать) объект класса Log, что было бы неудобно.*

При использовании статических методов есть ограничения. Вы НЕ можете получить доступ к НЕ статическим членам класса, внутри статического метода. Результатом компиляции приведенного ниже кода будет ошибка:

```
public class Counter{
    private int count;
    public static void main(String args[]){
        System.out.println(count); //compile time error
    }
}
```

Приведенный пример - это одна из наиболее распространённых ошибок новичков в Java. Так как метод main статический, а переменная *count* нет, в данном примере метод *println*, внутри метода main выдаст “Compile time error”. Еще одна распространённая ошибка, когда пытаются сделать локальную переменную статической.

2.2.6. Инициализация различных типов данных

Java гарантирует правильную инициализацию переменных перед использованием.

Если примитивный тип является членом данного класса, происходящее немного отличается. Так как любой метод может инициализировать или использовать данные, становится не практичным заставлять пользователя инициализировать их соответствующим значением перед использованием. Однако также не безопасно оставлять их заполненными всяким мусором, так как каждая переменная-член класса примитивного типа гарантированно получает инициализирующее значение.

```
class Measurement {
      boolean t;
      char c;
      byte b;
      short s;
      int i;
      long l;
      float f;
      double d;

      void print() {
            System.out.println("Data type      Initial value\n" + "boolean
      "
                  + t + "\n" + "char         [" + c + "] " + (int) c + "\n"
                  + "byte           " + b + "\n" + "short       " + s + "\n"
                  + "int            " + i + "\n" + "long        " + l + "\n"
                  + "float          " + f + "\n" + "double      " + d);
      }
}

public class InitialValues {
      public static void main(String[] args) {
            Measurement d = new Measurement();
            d.print();
      }
}
```

Вот что программа печатает на выходе:

```
Data type   Initial value
boolean     false
char        [ ] 0 // Значение char - это ноль, который печатается как пробел.
byte          0
short       0
int           0
long        0
float       0.0
double      0.0
```

Что произойдет, если присвоить переменной начальное значение? Прямой способ - просто присвоить значение в точке определения переменной в классе. Вот определение полей в классе Measurement, который изменен для обеспечения начальных значений:

```
class Measurement {
      boolean b = true;
      char c = 'x';
      byte B = 47;
      short s = 0xff;
      int i = 999;
      long l = 1;
      float f = 3.14f;
      double d = 3.14159;
      // . . .
}
```

Можно инициализировать объекты таким же способом. Если Depth - это класс, то можно вставить переменную и инициализировать ее следующим образом:

```
class Measurement {
      Depth o = new Depth();
      boolean b = true;
      // . . .
}
```

Для обеспечения начального значения можно вызвать метод:

```
class CInit {
      int i = f();
      // ...
}
```

Конечно, этот метод может иметь аргументы, но эти аргументы не могут быть другими членами класса, которые еще не инициализированы. Таким образом, это можно сделать так:

```
class CInit {
      int i = f();
      int j = g(i);
}
```

Мы помним, что в языке Java для создания массива, как и для создания любого другого объекта, используется оператор *new*. Тип элементов массива может быть либо примитивным (*int, double* и т.д.), либо ссылочным (объекты).

Элементы массивов примитивных типов по умолчанию инициализируются 0 (*false* для *boolean*).

Элементы ссылочного массива содержат ссылку *null* до тех пор, пока не будут явно проинициализированы. Поэтому при попытке обращения к не проинициализированному элементу массива произойдет ошибка NullPointerException.

Инициализацию массива можно проводить отдельно после объявления:

```
Integer[] arr = new Integer[3];
arr[0] = new Integer(1);
arr[1] = new Integer(5);
arr[2] = new Integer(-5);
```

Также возможна явная инициализация массива при объявлении:

```
Integer[] arr = new Integer[]{new Integer(1), new Integer(5), new Integer(-5)};
```

И даже без использования оператора *new*:

```
Integer[] arr = {new Integer(1), new Integer(5), new Integer(-5)};
```

Например, для числового массива явная инициализация записывается следующим образом:

```
int i[]={1, 3, 5};
int j[]={}; // эквивалентно new int[0]
```

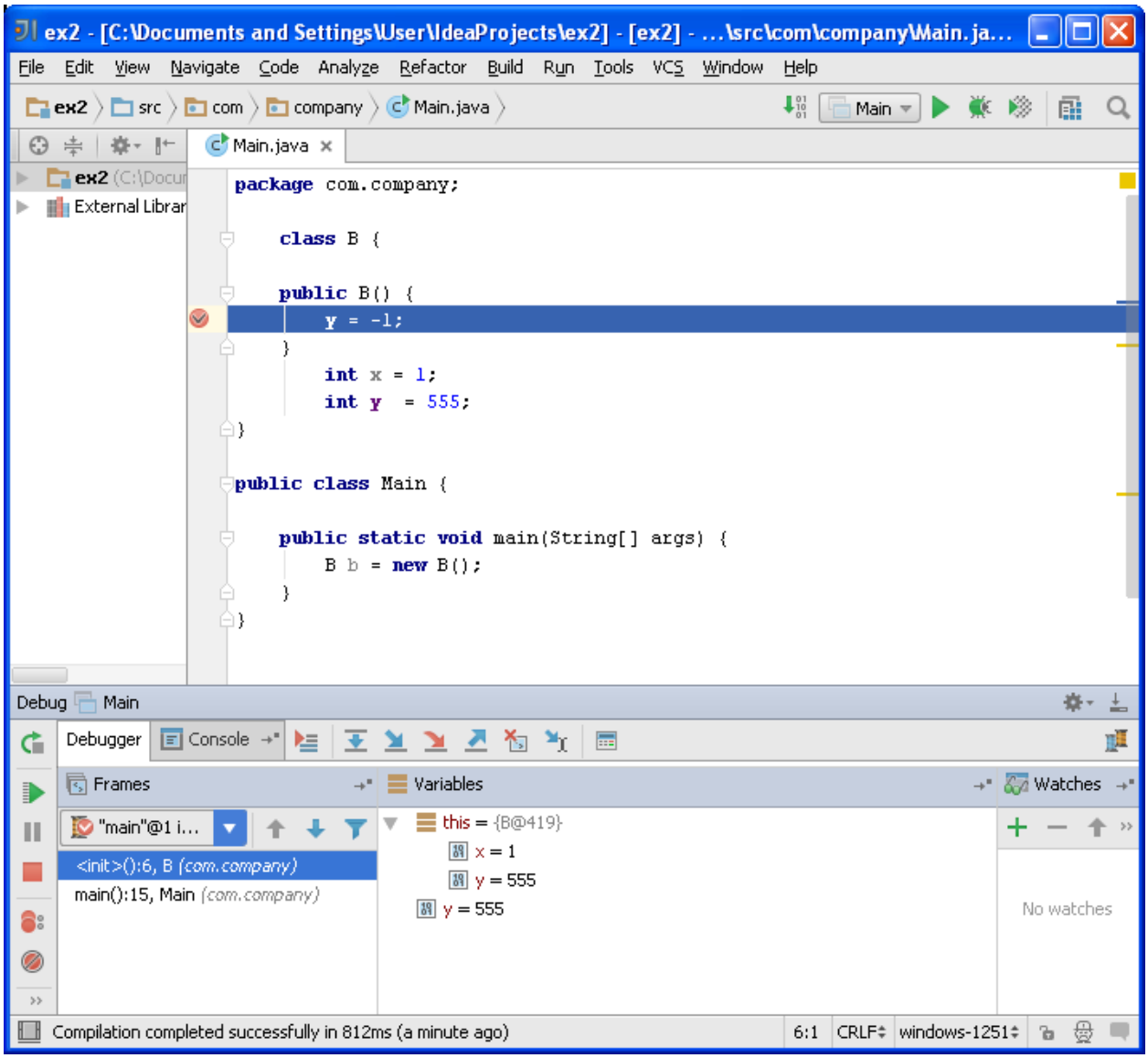
Для создания многомерных массивов можно использовать инициализаторы. В этом случае необходимо обратить внимание на количество необходимых вложенных фигурных скобок:

```
int i[][] = {{1,2}, null, {3}, {}};
```

В приведенном примере порождается четыре объекта. Это, во-первых, массив массивов длиной 4, а во-вторых, три массива второго уровня с длинами 2, 1, 0, соответственно.

Упражнение 2.2.2

Поля класса инициализируются до вызова конструктора, в каком бы порядке они не стояли в классе. Пример.



Разбор примера демонстрации создания и инициализации многомерных массивов

```
package com.company;

      class B {

    public B() {
      y = -1;
      }
      int x = 1;
      int y  = 555;
}

public class Main {

      public static void main(String[] args) {
      B b = new B();
      }
}
```

Задание 2.2.1

Написать функцию run(), тестирующую класс «Рациональная дробь». Функция должна создавать экземпляры класса, выполнять реализованные в классе методы и выводить результат. Реализация статических методов класса: сложение дробей; вычитание дробей; умножение дробей; деление дробей. Модифицируйте функцию *print*, чтобы вывод при необходимости был в виде смешанной дроби, убедитесь в корректности работы с отрицательными числами.

* Модифицируйте конструктор дроби, чтобы все хранимые дроби были несократимы

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателям IT ШКОЛЫ SAMSUNG Мансурову Руслану Маратовичу, Доброхотовой Людмиле Александровне и Куренкову Владимиру Вячеславовичу.