Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

Ю. В. Морозова

ПРАКТИКУМ ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПРОГРАММИРОВАНИЮ

Учебное пособие

Томск
2018

Морозова Ю. В.

Практикум по объектно-ориентированному программированию : учебное пособие / Ю. В. Морозова. – Томск : ФДО, ТУСУР, 2018. – 186 с.

Пособие позволяет систематизировать знания в области объектно-ориентированного программирования на языке Java, развить навыки разработки программного кода с использованием современных кросс-платформенных инструментальных средств.

Для студентов направлений «Программная инженерия» и «Бизнес-информатика», а также всех, кто продолжает изучать основы объектно-ориентированного программирования на языке Java.

Оглавление

Введение

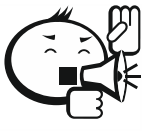
Современные технологии объектно-ориентированного программирования интенсивно развиваются, поэтому на данный момент программисту недостаточно понимать простейшие принципы ООП (инкапсуляцию, абстракцию, полиморфизм, наследование). В настоящее время появилось множество практик, соблюдение которых способствует экономии времени. Но несмотря на прогресс, эти принципы ООП являются основополагающими элементами и составляют основу языка программирования Java. Это означает, что программа, написанная на языке Java, должна строго соответствовать парадигме ООП, т. е. быть основана на использовании в программе объектов и классов.

Целью данного курса является углубленное изучение вопросов объектно-ориентированного программирования, способов создания и управления состояниями потоков, решение основных проблем при работе с потоками и синхронизацией. Рассмотрены потоки, дженерики и лямбда-выражения, которые были введены в Java 8 и сильно облегчили написание программного кода. Многие современные проекты успешно внедрили нововведения и используют их повседневно.

Для изучения данного учебного пособия необходимо знакомство с основными понятиями языка Java, такими как переменные, типы данных, массивы, методы и т. д. Пособие будет интересно студентам бакалавриата и всем интересующимся объектно-ориентированным программированием и его реализацией на языке Java.

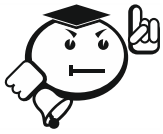
Соглашения, принятые в учебном пособии

Для улучшения восприятия материала в данном учебном пособии используются пиктограммы и специальное выделение важной информации.



Эта пиктограмма означает «Внимание!». Здесь выделена важная информация, требующая акцента на ней. Автор может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.

Эта пиктограмма означает определение или новое понятие.



В блоке «На заметку» автор может указать дополнительные сведения или другой взгляд на изучаемый предмет, чтобы помочь читателю лучше понять основные идеи.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Выводы

Эта пиктограмма означает выводы. Здесь автор подводит итоги, обобщает изложенный материал или проводит анализ.

Контрольные вопросы по главе

1 Концепции объектно-ориентированного программирования

Расцвет объектно-ориентированных технологий совпал с началом применения Интернета в качестве платформы для бизнеса и развлечений. А после того как стало очевидным, что Глобальная сеть активно проникает в жизнь людей, объектно-ориентированные технологии уже заняли удобную позицию для того, чтобы помочь в разработке новых веб-технологий. При этом мы сталкиваемся с объектами в своей повседневной жизни, вероятно, даже не осознавая этого. Они повсюду. Сегодня одной из наиболее интересных областей разработки программного обеспечения является интеграция унаследованного кода с мобильными и веб-системами [1]. Разработчики, одновременно обладающие навыками в веб-разработке как для мейнфреймов, так и для мобильных устройств, весьма востребованы. Такие объединения, по сути, превратились в соединения, основанные на объектах. Давайте вспомним основные понятия объектно-ориентированного программирования (ООП).

Объектно-ориентированное программирование *– это технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа* (*класса*)*, а классы образуют иерархию с наследованием свойств* [2].

Java используется для разработки как веб-, так и мобильных приложений. Поскольку Java является объектно-ориентированным языком, то он поддерживает объектные типы данных и содержит языковые конструкции для представления классов и объектов.

1.1 Классы и объекты

Мы живем в мире объектов. Стол, автомобиль, ручка – это объекты. Наряду с физическими существуют также абстрактные объекты, представителями которых, например, являются числа или геометрические фигуры.

Объект (*object*) *– это сущность, обладающая определенным поведением и способом представления.*

Класс (*class*) *– это шаблон или прототип, по которому создаются объекты* [3].

Диаграмма классов является ключевым элементом в объектно-ориентированном моделировании. На диаграмме (рис. 1.1) классы изображаются в рамках, содержащих три компонента:

1. В верхней части написано имя класса. Имена классов начинаются с заглавной буквы. Если класс абстрактный, то его имя пишется полужирным курсивом.
2. В средней части располагаются переменные – члены класса (поля, атрибуты), которые представляют собой статические свойства класса. Они начинаются с маленькой буквы.
3. Нижняя часть диаграммы содержит методы класса, определяющие его динамическое поведение. Они также выровнены по левому краю и пишутся с маленькой буквы.

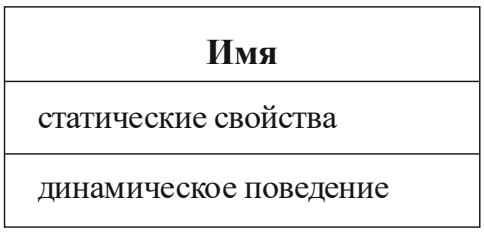


Рис. 1.1 – Компоненты диаграммы класса

На рисунке 1.2 приведен пример класса *Student* на диаграмме UML.

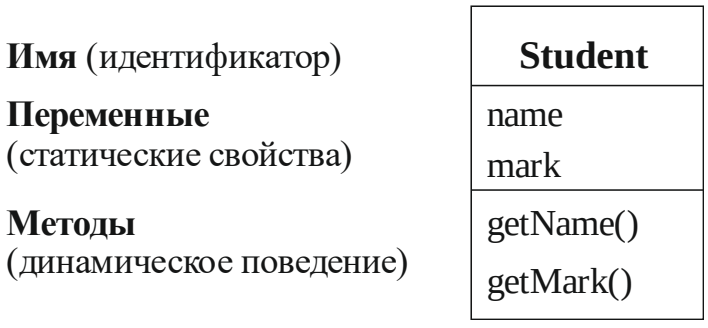


Рис. 1.2 – Диаграммы UML класса *Student*

Класс моделирует состояние и поведение объектов реального мира. Класс содержит статические свойства (их также называют полями, атрибутами, характеристиками, переменными – членами класса) и динамическое поведение, общие для всех объектов, в закрытом «запечатанном ящике» и определяет открытый интерфейс для использования таких «ящиков».

Поле* (*атрибут, переменные*) *класса *– это характеристика объекта, описывающая его свойство.*

Поскольку классы хорошо инкапсулированы, то их легко использовать повторно. Таким образом, объектно-ориентированное программирование в классе объединяет данные и инструкции для обработки данных. Объект в ООП – это экземпляр некоторого класса.

Экземпляр класса (*instance*) *– это отдельная реализация класса. Все экземпляры класса имеют одинаковые свойства, которые описаны в определении класса.*

Объект состоит из трех частей: имя объекта (*names*), состояние (*attributes*), поведение (*behaviors*) (рис. 1.3), т. е. объект обладает *состоянием*, *поведением* и *идентичностью*.

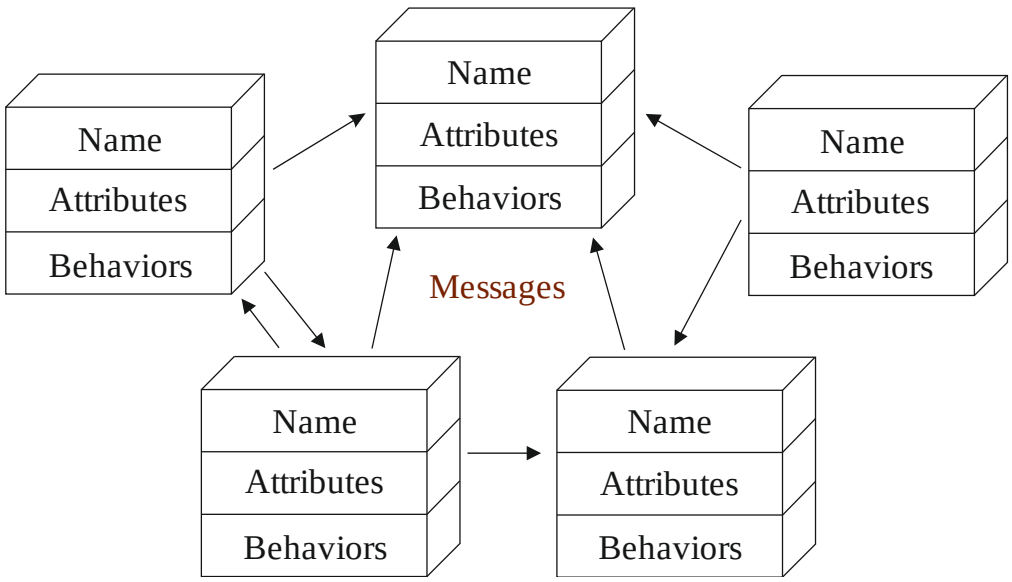


Рис. 1.3 – Представление объектов

Состояние (*attributes*) *объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.*

Поведение (*behaviors*) *– это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.*

Идентичность (*names*) *– это такое свойство объекта, которое отличает его от всех других объектов* [2].

Объекты – это отдельные, четко обозначенные экземпляры некоторого класса. Класс дает общее описание объектов, указывает, «на что они похожи». Термины «экземпляр класса» и «объект» взаимозаменяемы.

Объект также представляется диаграммой, состоящей из трех компонентов. На диаграмме объектов отображаются объекты с указанием текущих значений их полей и связей между объектами (рис. 1.4).

	ivanov:Student	petrov:Student
Имя		
Переменные	name="Иванов" mark=90	name="Петров" mark=63
Методы	getName() getMark()	getName() getMark()

Рис. 1.4 – Диаграмма объектов *ivanov* и *petrov* класса *Student*

Метод (*method*) *– это последовательность команд, которые вызываются по определенному имени.*

По большому счету метод – это то, что может делать объект соответствующего класса (или что можно делать с объектом).

Важным понятием является *сигнатура метода*.

Сигнатура (*signature*) *определяется именем метода и его аргументами* (*количеством, типом, порядком следования*).

Если для полей запрещается совпадение имен, то для *методов* в классе запрещено создание двух *методов* с одинаковыми *сигнатурами*.

В сигнатуру метода не входят возвращаемое значение, бросаемые им исключения, а также модификаторы.

Существуют два основных типа методов (кроме конструкторов): *методы экземпляра* и *статические методы*. Выполнение *метода экземпляра* зависит от состояния конкретного экземпляра класса и может выполняться после создания объекта. Для вызова метода экземпляра класса необходимо ссылаться на объект и после точки указать имя метода и параметры, которые должны быть переданы, например:

```
objectReference.someMethod();
objectReference.someOtherMethod(parameter);
```

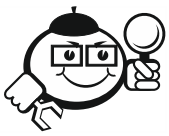
Статические методы иногда еще называют методами класса, поскольку их выполнение не зависит от состояния какого-либо конкретного объекта. Для определения метода класса используется ключевое слово *static*. Поведение статического метода определяется на уровне класса.

Наряду со статическими методами существуют *статические переменные* – переменные, определенные с ключевым словом *static*. Статические переменные определяются на уровне класса и совместно используются всеми объектами класса. Существует только одна копия статической области в классе, независимо от того, сколько экземпляров класса создано. Статические переменные называются переменными класса, т. е. переменными, относящимися ко всему классу, в отличие от переменных, относящихся к его отдельным объектам. Они инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализация требует сложных вычислений, например циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный ключевым словом *static*, который тоже будет выполнен до запуска конструктора.

Для определения класса в Java используется ключевое слово *class*.

Синтаксис определения класса следующий:

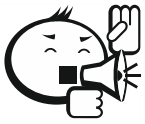
```
[модификаторы доступа] class Имя_Класса { //Тело класса, содержащее описание переменных и методов ….. }
```



Пример

Определение класса в Java

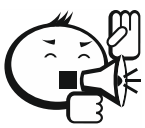
```
//Тело класса, содержащее описание полей – переменных и методов
public class Student {
//поля класса
	private String name;
	private int mark;

	//методы
		public String getName() {
			return name;
		}

			public void setMark(int mark) {
			mark=mark;
		}

	}
```



Соглашение об именах классов. Рекомендуется в качестве имени класса использовать существительные или фразы, набранные латинским шрифтом, состоящие из нескольких существительных, имеющих смысл в используемом контексте. Все слова, входящие в имена, должны начинаться с большой буквы.

1.2 Геттеры и сеттеры

Следующие понятия из ООП, которое следует рассмотреть, – это геттеры и сеттеры (*get* – получать, *set* – устанавливать). Это общепринятые способы *вводить данные* (*set*) или *получать данные* (*get*). Геттеры и сеттеры выполняют важную миссию *защиты* данных.



Сеттеры и геттеры реализуют главный принцип ООП – принцип инкапсуляции – сокрытие данных [4].

Метод *геттер* не имеет параметров (т. е. в скобках ничего не пишется) и возвращает значение одной переменной (одного поля). Метод *сеттер* всегда имеет модификатор `void` и только один параметр, для изменения значения одного поля.

Например, у нас есть класс `Student`. Зададим, используя `setName`, имя студента. А потом эти данные можно получить с помощью `getName`. Поле в нашем классе, которое мы хотим защитить, пометим модификатором `private`. Добавим метод геттер/сеттер для этого поля.

```
//геттер для получения имени, т. е. метод, который возвращает имя студента
        public String getName() {
             return name;
        }
        //сеттер для установки имени, т. е. метод для инициализации поля name
        public void setName(String name) {
             this.name=name;
             }
```

Обычно имя геттера состоит из слова `get+название_переменной`. Имя сеттера состоит из слова `set+название_переменной`.

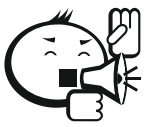
Имея соответствующую реализацию сокрытия информации, разработчик класса может осуществлять полный контроль над тем, что пользователь класса может и чего не может делать.

1.3 Перегрузка методов

Часто одно и то же слово имеет несколько разных значений, т. е. оно *перегружено*. Можно создавать методы с одинаковыми именами, но с разным набором аргументов. Java позволяет создавать несколько методов с одинаковыми именами, но разными сигнатурами.

Различные реализации методов с одинаковыми именами, но разными сигнатурами в Java называются ***перегрузкой методов****.*

Какой из перегруженных методов должен выполняться при вызове, Java определяет на основе фактических параметров.



Перегрузка методов реализует такое важное свойство в программировании, как полиморфизм. Существует два простых правила перегрузки методов, при нарушении которых компилятор выдает сообщение об ошибке:

- нельзя перегружать метод, изменив тип возвращаемого им значения;
- не должно быть двух методов с одной сигнатурой.

Пример

```
public class Main {
      public static void viewN(Integer i) { // метод 1
            System.out.printf("Integer=%d%n", i);
            }
      public static void viewN(int i) { // метод2
            System.out.printf("int=%d%n", i);
            }
       public static void viewN(Float f) { // метод 3
            System.out.printf("Float=%.4f%n", f);
            }
       public static void viewN(Number n) { // метод 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) {
             viewN(n);
             }
             viewN(new Integer(8));
             viewN(81);
             viewN(4.14f);
             viewN(8.2);

                              }
}
```

Может показаться, что в результате компиляции и выполнения данного кода будут последовательно вызваны все четыре метода, однако в консоль будет выведен иной результат (рис. 1.5).

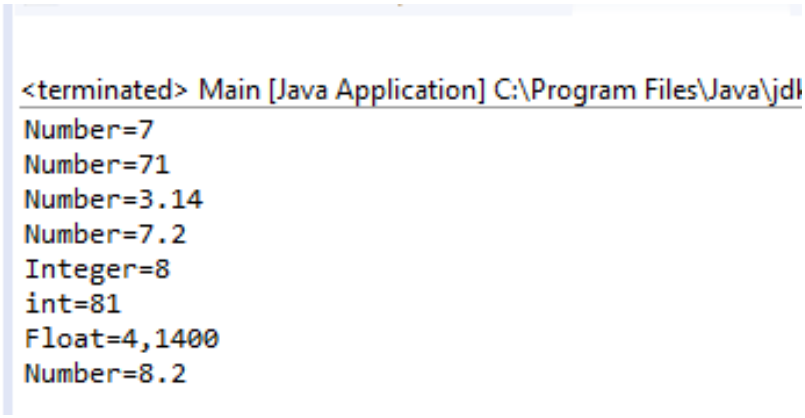


Рис. 1.5 – Перегрузка метода `viewN()`

Во всех случаях при передаче в метод *элементов массива num* был вызван четвертый метод. Это произошло вследствие того, что выбор варианта перегруженного метода происходит на этапе компиляции и зависит от типа массива. 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.

Аналогично перегрузка используется и для *конструкторов*, об этом поговорим чуть позже.

1.4 Ключевые слова *this* и *super*

`this` *и* `super` *– это специальные ключевые слова в Java, которые представляют соответственно текущий экземпляр класса и его суперкласса.*

`this` представляет текущий экземпляр класса, в то время как `super` – текущий экземпляр родительского класса.

Внутри класса для вызова своего конструктора без аргументов используется `this()`, тогда как `super()` используется для вызова конструктора без ар-

гументов, или, как его еще называют, конструктора по умолчанию родительского класса. Ключевое слово `this` используется для ссылки на данный объект внутри описания класса. В основном ключевое слово `this` используется для того, чтобы избежать двойного толкования.

При этом следует учесть, что `this` и `super` – это нестатические переменные, соответственно, их нельзя использовать в статическом контексте, а это означает, что их нельзя использовать в методе `main()`. То же самое произойдет, если в методе `main()` воспользоваться ключевым словом `super`.

Приведем пример использования ключевых слов `this` и `super`.


```
//сеттер для установки оценки
          public void setMark(int mark) {
               this.mark=mark;
          }

//геттер для получения оценки
               public int getMark() {
                    return mark;
}
```

В приведенном фрагменте кода есть два идентификатора с именем `mark` – переменная-член класса (поле) и параметр метода. Это вызывает конфликт имен. Чтобы избежать конфликта имен, можно было бы назвать параметр метода `m` вместо `mark`.

Однако имя `mark` является более содержательным и близким по контексту. Java предоставляет ключевое слово `this` для разрешения конфликта имен. Так, `this.mark` отсылает к переменной – члену класса, т. е. к полю, в то время как `mark` предполагает параметр метода.

Выводы:

- this.имяПеременной – ссылка на поле имяПеременной этого объекта; this.имяМетода(…) – вызывает метод имяМетода (…) данного объекта.
- В конструкторе можно использовать this(…) для вызова другого конструктора этого класса.
- Внутри метода можно использовать предложение “return this” для возврата этого объекта при вызове.

Основное назначение `this` и `super` – вызывать один конструктор из другого и ссылаться на переменные экземпляра, объявленные в текущем классе и его родительском классе.

1.5 Метод *toString*()

Каждый хорошо разработанный java-класс должен иметь public-метод с именем `toString()`, который возвращает текстовое описание объекта. Метод `toString()` можно явно или неявно вызвать следующим образом:

- имяОбъекта.toString();
- через `println`;
- через конкатенацию строк, т. е. оператор ‘+’.

Выполнение `println(имяОбъекта)` с объектом в качестве аргумента неявно вызывает метод `toString()` для этого объекта. Поскольку `toString()` определен в классе `java.lang.Object`, а его реализация по умолчанию не предоставляет много информации, всегда рекомендуется переопределить метод `toString`() в подклассе.

Если мы будем использовать напрямую `toString()`, то получим значения *Hashcode* для объектов.

Например,

```
Student s1 = new Student();


                //задаем, устанавливаем через сеттеры оценку
                    s1.setMark(3);
                    s1.setName("Василий");
                    //выводим на консоль его ФИО
                    System.out.println(s1.toString());
```

На консоли увидим следующий результат (рис. 1.6).

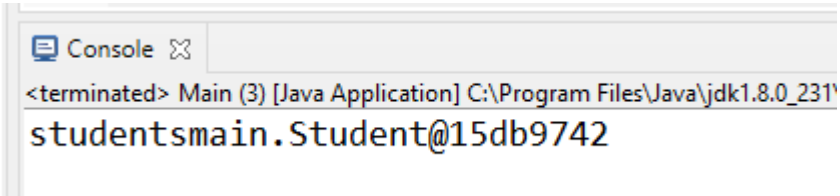


Рис. 1.6 – Вызов метода `toString()`

Поскольку java-компилятор внутренне вызывает метод `toString()`, переопределение этого метода возвращает указанные значения. Переопределим этот метода в классе `Student`.

```
//возвращает краткое описание студента
@Override
        public String toString() {
             return "Студент "+name+" получил оценку "+mark;
    }
```

Далее можно вывести информацию о конкретном студенте (рис. 1.7).

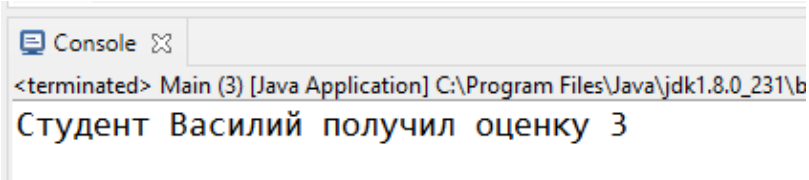


Рис. 1.7 – Вызов переопределенного метода `toString()`

Подробное описание метода `toString()` приведено в [5, гл. 10].

1.6 Конструкторы

Каждый класс может также иметь специальные методы, которые автоматически вызываются при создании и уничтожении объектов этого класса:

- конструктор (*constructor*) – выполняется при создании объектов;
- деструктор (*destructor*) – выполняется при уничтожении объектов.

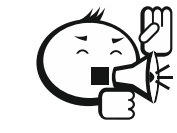
Как и в C++, в классах Java имеются конструкторы. Их назначение полностью совпадает с назначением аналогичных методов C++.

Конструктор (*constructor*) *– это особенный метод класса, который вызывается автоматически в момент создания объектов этого класса. Имя конструктора совпадает с именем класса.*

Конструкторы добавляются в класс, если в момент создания объекта нужно выполнить какие-то действия (начальную настройку) с его данными (полями).

По синтаксису конструктор похож на метод без возвращаемого значения. Также выделяют заголовок и тело конструктора. Заголовок состоит из модификаторов доступа (никакие другие модификаторы недопустимы). Тело конструктора пустым быть не может и поэтому всегда описывается в фигурных скобках

(для простейших реализаций скобки могут быть пустыми). Как и обычному методу, конструктору можно передавать аргументы. Передаются и используются они по той же схеме, что и для прочих методов класса. Однако теперь при создании объекта необходимо передать аргументы для конструктора. Аргументы передаются в круглых скобках после имени класса в команде создания объекта.



Класс обязательно должен иметь конструктор, иначе невозможно порождать объекты ни от него, ни от его наследников. Поэтому если в классе не объявлен ни один конструктор, компилятор добавляет один по умолчанию. Это public-конструктор без параметров и с телом, описанным парой пустых фигурных скобок.

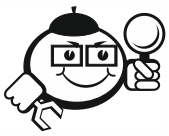
Автоматический вызов конструктора по умолчанию позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

Если в классе не определен конструктор без аргументов (но определен хотя бы один конструктор), рассчитывать на конструктор по умолчанию (конструктор без аргументов) нельзя – необходимо передавать аргументы в соответствии с описанным вариантом конструктора.

Этот конструктор не имеет параметров, он только вызывает конструктор без параметров класса-предка.

Общее правило заключается в том, что конструктор должен быть, даже если не планируете что-либо делать внутри него. Можно предусмотреть конструктор, в котором ничего нет, а затем добавить в него что-то. Хотя технически ничего плохого в использовании конструктора по умолчанию, обеспечиваемого компилятором, нет, в целях документирования и сопровождения никогда не будет лишним.

Для создания объекта некоторого класса надо использовать специальный оператор `new` после обращения к одному из конструкторов.



Пример

```
//создаем студента s1
Student s1 = new Student();
```

Здесь мы используем конструктор по умолчанию без аргументов.

Добавим к нашему классу `Student` еще конструкторы.

```
//конструкторы перегружаемые
//конструктор с двумя аргументами
	public Student(String name, int mark) {
		this.name=name;
		this.mark=mark;
	}
	//конструктор с одним аргументом
		public Student(String name) {
			this.name=name;

		}
		//конструктор с одним аргументом
		public Student(int mark) {

			this.mark=mark;
		}
//добавляем пустой конструктор, так как конструктор по умолчанию уже не можем использовать
	public Student() {
		}
```

Напомним, что перегрузка метода означает, что метод с одним и тем же именем может иметь различные реализации, что достигается различием в списке параметров (их количеством, типом или порядком). Конструктор, как и другие методы, может быть перегружаемым.

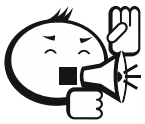
В рассмотренном классе `Student` определили три перегружаемых версии конструкторов с одинаковым именем *Student*, различающихся списком параметров:

```
Student();
Student(int mark);
Student(String name,int mark);
```

В зависимости от списка фактических параметров будет вызываться соответствующий конструктор. Если список параметров не соответствует ни одному методу, будет выдана ошибка компиляции.

```
// s1, s2, s3 – объекты класса Student, т. е. наши тестовые студенты

        Student s1 = new Student();
        Student s2 = new Student(5);
        Student s3 = new Student("Иван",5);
```

В результате переменная `s3` будет указывать на конкретного студента по имени Иван, который получил оценку 5.



Конструктор не может быть `static`, `abstract` или `final`.

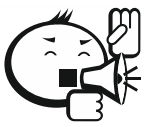
Конструкторы могут иметь любой уровень доступа, при этом применяются определенные правила видимости, которые приведены в таблице 1.1.

Таблица 1.1 – Правила доступа к конструкторам

Модификатор доступа к конструктору	**Описание**
`public`	Конструктор может быть вызван любым классом
`protected`	Конструктор может быть вызван классом из того же пакета или любым его подклассом
Без модификатора (`package-private`)	Конструктор может быть вызван любым классом из того же пакета
`private`	Конструктор может быть вызван только из того класса, в котором он определен

Существуют ситуации, когда конструкторы объявляются с доступом `protected` или `private`. Конструкторы `private` используются, когда по тем или иным причинам не допускается прямое создание объекта класса с помощью ключевого слова `new`, например, с использованием паттерна `Factory`.

В отличие от других объектно-ориентированных языков в Java процесс удаления ненужных объектов происходит автоматически, с помощью самой виртуальной машины, и называется «сборка мусора» (*garbage collection*).



В Java нет деструкторов, но существует возможность объявлять методы с именем `finalize`. Методы `finalize` аналогичны деструкторам в C++ (ключевой знак ~) и Delphi (ключевое слово `destructor`).

Несколькими предложениями этот процесс можно описать так:

У каждого объекта имеется счетчик ссылок, указывающих на него. В момент создания объекта счетчик инициализируется единицей, и впоследствии во время работы программы, если появляются новые ссылки на этот объект, счетчик увеличивается. Когда какая-то ссылка становится «ненужной», счетчик уменьшается. Виртуальная машина время от времени проверяет все объекты программы, и для тех объектов, чьи счетчики ссылок имеют нулевое значение, очищает память.

Рождение, жизнь и смерть объекта:

1. Создание: `new ...(..)`.
2. Присвоение ссылки: `object = ...`
3. Потеря ссылки.
4. Сборка мусора.

Естественно, описанный механизм весьма абстрактно описывает процесс очистки.

Выводы

Отличия конструктора от обычного метода:

- имя конструктора всегда совпадает с именем класса и по соглашению об именах начинается с большой буквы;
- у конструктора нет возвращаемого значения, следовательно, не разрешено использовать предложение `return` в теле конструктора;
- конструктор может быть вызван только через оператор `new`, при этом может быть вызван только 1 раз для создаваемого объекта;
- конструкторы не наследуются (обсудим это ниже).

1.7 Определение класса в Java

Класс `Student` не имеет главного метода `main()`, поэтому этот класс нельзя запустить на выполнение как программу. Описание класса `Student` может быть использовано как строительный блок для других программ.

Один из методов обязательно должен называться `main`, с него начинается выполнение программы. В нашей простейшей программе только один метод, а значит, имя его `main` (исключение составляют апплеты – у них метода `main()` нет). Метод `main()` иногда называют главным методом программы, поскольку во многом именно с этим методом отождествляется сама программа.

Ключевые слова `public`, `static` и `void` перед именем метода `main()` означают буквально следующее: `public` – метод доступен вне класса, `static` – метод статический и для его вызова нет необходимости создавать экземпляр класса (то есть объект), `void` – метод, который не возвращает результат. Модификаторы и уровни доступа рассмотрим немного позже.

Инструкция `String[] args` в круглых скобках после имени метода `main()` означает тип аргумента метода: формальное название аргумента `args`, и этот аргумент является текстовым массивом (тип `String`).

Добавим в проект новый класс `Main`, в котором запишем главный метод.

```
public class Main {

     public static void main(String[] args) {
// s1, s2, s3 – объекты класса Student, т. е. наши тестовые студенты

          Student s1 = new Student();
          Student s2 = new Student(5);
          Student s3 = new Student("Иван",5);

          //задаем, устанавливаем через сеттеры оценку
          s1.setMark(3);
          s1.setName("Василий");
          //выводим на консоль имя s3
          System.out.println(s3.getName());
          // выводим на консоль краткое описание студента s1
          System.out.println(s1.toString());
               }

          }
```

При создании нового объекта в первую очередь выполняется вызов конструктора. Происходит инициализация объектов, задаются начальные значения.

Инициализацию объектов можно выполнить тремя способами:

1. Инициализация по ссылке (инициализация объекта означает сохранение данных в объекте).

```
//Тело класса, содержащее описание полей – переменных и методов
class Student{
       int id;
       String name;
      }
      class TestStudent3{
       public static void main(String args[]){
        //Объявление объектов
        Student s1=new Student();
        Student s2=new Student();
        //Инициализация объектов
        s1.id=101;
        s1.name="Петров";
        s2.id=102;
        s2.name="Иванов";
        //Вывод значений полей
        System.out.println(s1.id+" "+s1.name);
        System.out.println(s2.id+" "+s2.name);
       }
      }
```

2. Инициализация с помощью метода.

```
class Student{
       int number;
       String name;
       void insertRecord(int num, String n){
        number=num;
        name=n;
       }
       void printInfo(){System.out.println(number+" "+name);}
      }
      class TestStudent{
       public static void main(String args[]){
        Student s1=new Student();
        //инициализации полей в методе
        s1.insertRecord(1,"Иванов");
```

```
    s1.printInfo();

  }
}
```

3. Инициализация объектов в конструкторе.

Далее выведем информацию о студентах на консоль (рис. 1.8).

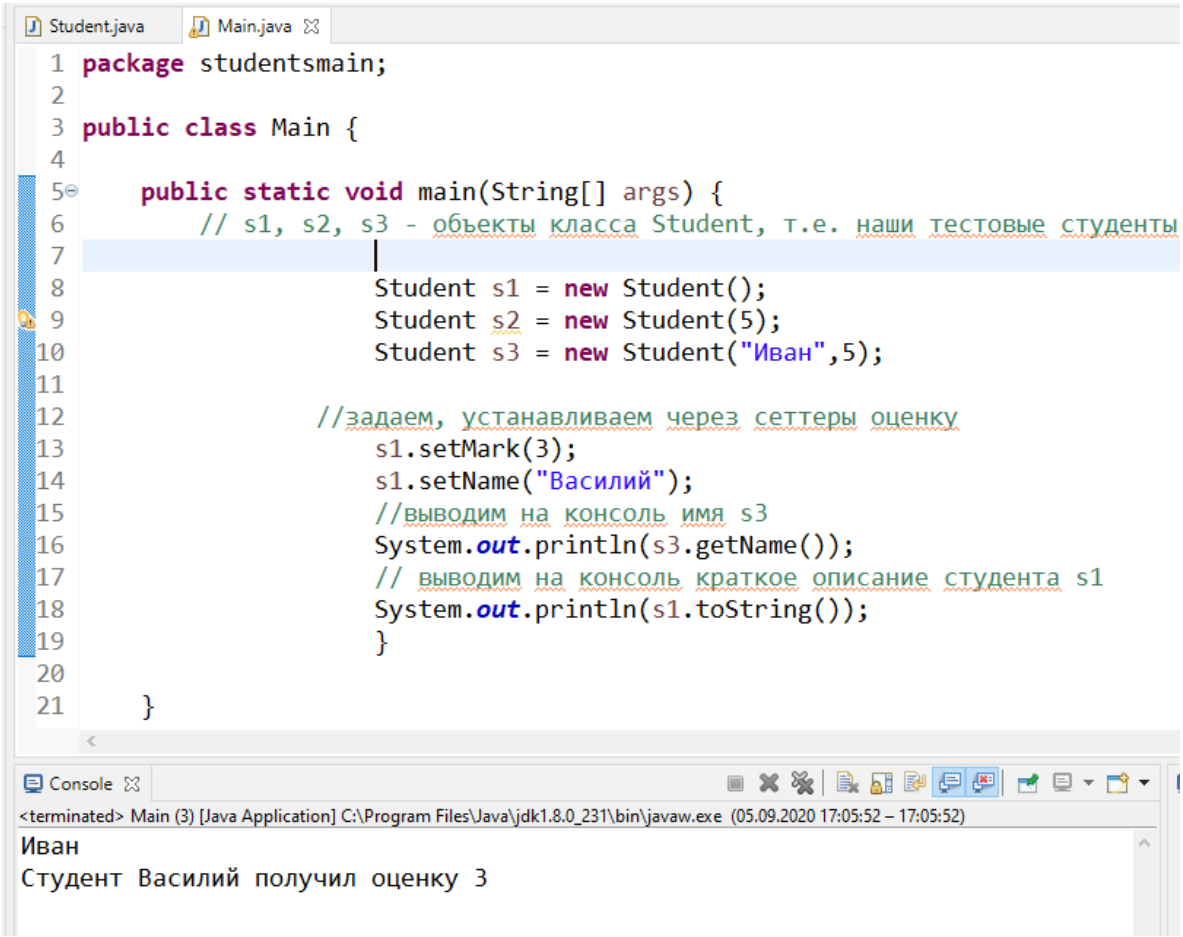
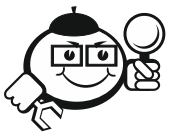


Рис. 1.8 – Результат инициализации объектов в конструкторе

Рассмотрим важное понятие *анонимный объект*. Анонимный означает безымянный.

Объект, на который нет ссылки, называется ***анонимным объектом.***

Его можно использовать только во время создания объекта. Если вам нужно использовать объект только один раз, анонимный объект – хороший подход.



Пример

Рассмотрим пример анонимного объекта.

```
class Calculation{
       void fact(int  n){
        int fact=1;
        for(int i=1;i<=n;i++){
         fact=fact*i;
        }
       System.out.println("Факториал равен "+fact);
      }
      public static void main(String args[]){
       new Calculation().fact(5);//Вызов метода через анонимный объект
      }
}
```

1.8 Принципы ООП

Исторически сложилось так, что объектно-ориентированные языки определяются следующими концепциями: инкапсуляцией, наследованием и полиморфизмом. Поэтому если тот или иной язык программирования не реализует все эти концепции, то он, как правило, не считается объектно-ориентированным. Наряду с этими тремя терминами можно выделить и абстракцию. Таким образом, основные принципы ООП:

- инкапсуляция;
- наследование;
- полиморфизм;
- абстракция.

Далее рассмотрим их.

1.8.1 Абстракция

Абстрагирование является одним из основных методов, используемых для решения сложных задач. В объектно-ориентированном подходе предметы и понятия реального мира заменяются моделями, т. е. определенными формальными конструкциями набора реальных объектов (сущностей) предметной области. Модель содержит не все признаки и свойства представляемого ею предмета или

понятия, а лишь те из них, которые существенны с точки зрения разрабатываемой программной системы. Упрощение модели (абстрагирование от ненужных деталей) по отношению к реальному предмету позволяет сделать ее формальной, благодаря чему при разработке можно четко выделить все зависимости и операции над ними в создаваемой программной системе. Это упрощает как изучение (анализ) и разработку моделей, так и их реализацию на компьютере.

Гради Буч дает следующее определение абстракции.

Абстракция (*abstraction*) *выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя* [2].

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Для этого необходимо *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями и потому непригодной.

Разница между классом и объектом такая же, как между абстрактным понятием и реальным объектом. Выделяют следующие виды абстракций (степень связи с реальным объектом уменьшается с каждым пунктом):

1) *instance* (объект) – сущность, обладающая набором характеристик, свойственных конкретному экземпляру класса. Имеет конкретные значения полей (низший уровень, без абстракции);
2) *class* (класс) – описание множества объектов, схожих по свойствам и внутренней структуре (шаблон для создания объектов);
3) *abstract class* (абстрактный класс) – абстрактное описание характеристик множества классов (выступает как шаблон для наследования другими классами). Имеет высокий уровень абстракции, в связи с чем от абстрактного класса нельзя создавать объекты напрямую (только через создание объектов от классов-наследников);

4) *interface* (интерфейс) – это конструкция языка программирования Java, в рамках которой могут описываться только абстрактные публичные методы (*abstract public*) и статические константы свойства (*final static*). То есть так же, как и на основе абстрактных классов, на основе интерфейсов нельзя порождать объекты.

Абстрактные классы и интерфейсы рассмотрим позже.

Так как *интерфейс* схож с *абстрактным классом*, но позволяет (неявно) выполнить множественное расширение, он имеет максимальный уровень абстракции.

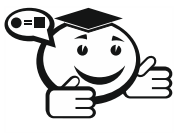
Так, для описания класса `Student` имеет смысл рассматривать такие характеристики объектов, как фамилия, имя, отчество, номер зачетной книжки, номер курса, номер группы, оценки. Не имеет смысла оценивать, например, внешние данные или характер.

1.8.2 Инкапсуляция

При использовании объектно-ориентированного подхода не принято использовать прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято использовать специальные методы этого класса для получения и изменения его свойств (*сеттеры* и *геттеры*). Внутри объекта данные и методы могут обладать различной степенью открытости (или видимости).

Область видимости *– это область программы, в пределах которой идентификатор некоторой переменной, метода или класса является связанным с этой переменной, соответственно, методом или классом. За пределами области видимости тот же самый идентификатор может быть связан с другими переменными, методами, классами.*

Инкапсуляция выступает договором для объекта, что он должен скрыть, а что открыть для доступа другим объектам.



Инкапсуляция (*encapsulation*) – *сокрытие реализации класса и отделение его внутреннего представления от внешнего* (*интерфейса*).

В Java используют модификаторы доступа, чтобы скрыть метод и ограничить доступ к переменной из внешнего мира.

Модификатор (*modifier*) – *это ключевое слово языка, которое может каким-либо образом изменить смысл некоторого определения* (*например, класса или метода*).

Java также располагает различными модификаторами доступа: `public`, `package` (по умолчанию), `protected`, `private`.

Итак, модификаторы доступа используются для управления видимостью класса или членов класса – полей и методов:

1) `public`: класс, переменная или метод доступны всем другим объектам в системе;
2) `private`: класс, переменная или метод доступны только внутри класса, в котором они объявлены. Любой другой класс из того же пакета не будет иметь доступа к этим членам класса. Классы и интерфейсы не могут быть объявлены как `private`;
3) `default` (модификатор, по умолчанию): если перед именем класса, метода или переменной отсутствует модификатор доступа, то применяется доступ по умолчанию – `default`. В этом случае члены класса видны только внутри пакета (если класс будет объявлен таким образом, то он тоже будет доступен только внутри пакета);
4) `protected`: члены класса (поля и методы) доступны только внутри пакета и в наследниках данного класса в других пакетах.

В UML нотации члены класса с различными модификаторами отображаются следующими знаками: public – «+»; protected – «#»; private – «-»; без модификатора – «~».

Если рассматривать с позиции инкапсуляции, то модификаторы доступа позволяют ограничить нежелательный доступ к членам класса извне (рис. 1.9).

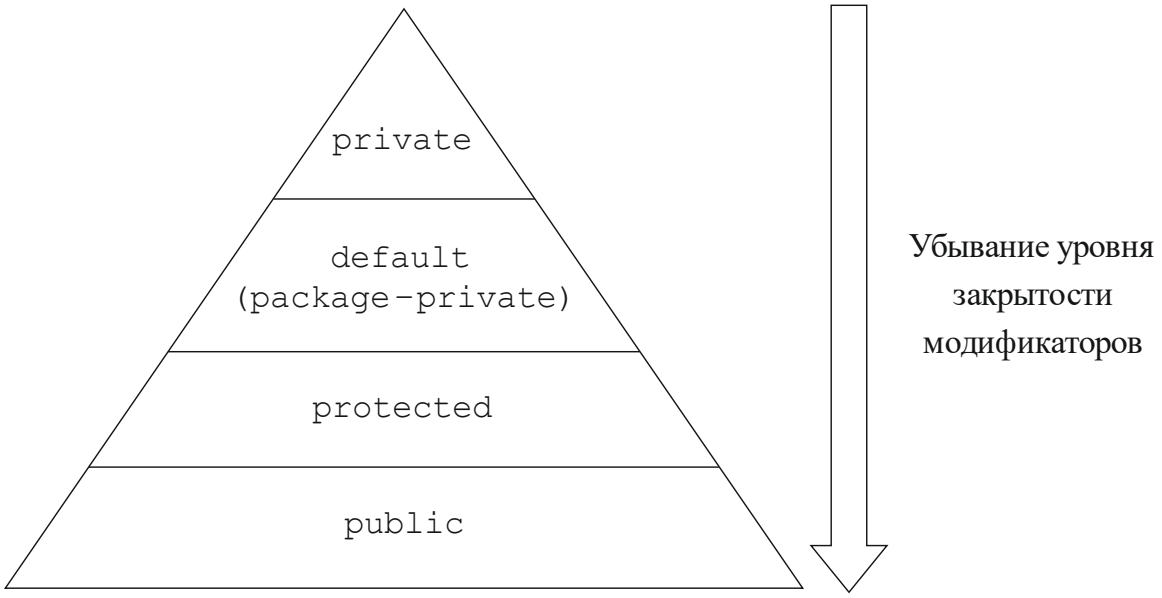
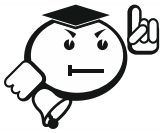


Рис. 1.9 – Уровни доступа



Концепция геттеров и сеттеров поддерживает концепцию скрытия данных. Поскольку другие объекты не должны напрямую манипулировать данными, содержащимися в одном из объектов, геттеры и сеттеры обеспечивают управляемый доступ к данным объекта. Геттеры и сеттеры иногда называют методами доступа и методами-модификаторами соответственно.

Подробнее разберем класс `Student` (рис. 1.10).

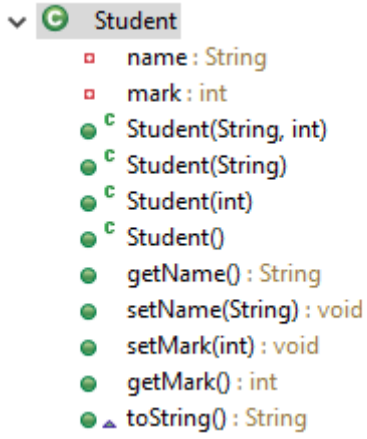


Рис. 1.10 – Структура класса `Student`

По данному классу доступной для пользователей частью являются открытые методы сеттеры и геттеры, а поля `name` и `mark` являются закрытыми. Доступ к ним осуществляется посредством сеттеров и геттеров. Мы скрыли часть реализации. Это очень важно при написании кода, который будет иметь дело с данными.

1.8.3 Наследование

Наследование – один из основополагающих принципов ООП.

Наследование *– свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется* ***базовым***, ***родительским*** *или* ***суперклассом****. Новый класс –* ***потомком***, ***наследником***, ***дочерним*** *или* ***производным классом****.*

Главное преимущество наследования в том, что оно обеспечивает формальный механизм повторного использования кода и избегает дублирования. Унаследованный класс расширяет функциональность приложения благодаря копированию поведения родительского класса и добавлению новых функций. Это делает код сильно связанным. Если вы захотите изменить суперкласс, вам придется знать все детали подклассов, чтобы не разрушить код. Но наследование обладает и недостатком – *сильная связанность*: подкласс зависит от реализации родительского класса, что делает код сильно связанным.

Целью наследования является упорядочение классов в иерархическую древовидную структуру.

Иерархия классов *представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях.*

Создадим класс-наследник `GraduateStudent` для нашего класса `Student` (рис. 1.11).

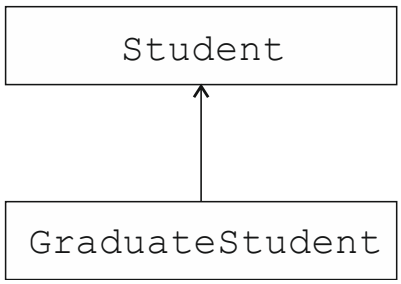
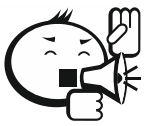


Рис. 1.11 – Пример иерархии классов

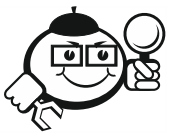
Суперкласс, или родительский класс (иногда называемый базовым), содержит все атрибуты и поведения, общие для классов, которые наследуют от него. Например, в случае с классом `Student` все классы ниже по иерархии содержат аналогичные атрибуты, такие как `name`, `mark`, а также *сеттеры* и *геттеры*, поэтому нет необходимости дублировать их. Дублирование потребует много дополнительной работы и, что вызывает, пожалуй, наибольшее беспокойство, может привести к ошибкам и несоответствиям. Подкласс, или дочерний класс (иногда называемый производным), представляет собой расширение суперкласса. Таким образом, класс `GraduateStudent` наследует все общие атрибуты и поведения от класса `Student`. Класс `Student` считается суперклассом подклассов, или дочерних классов, `GraduateStudent`. Наследование обеспечивает большое количество преимуществ в плане проектирования. Добавим в класс `GraduateStudent` только поле `thesis` (тема ВКР), и получение среднего балла для дипломника – метод `GPA()`.

Для того чтобы один класс был потомком другого, необходимо при его объявлении после имени класса указать ключевое слово `extends` и название суперкласса. Он должен быть доступным классом и не иметь модификатора `final`. Если ключевое слово `extends` не указано, считается, что класс унаследован от универсального класса `Object`.



В Java класс-наследник может иметь только одного родителя. Множественное наследование в Java реализуется только через интерфейсы.

```
class Имя_класса_наследника extends Имя_класса_родителя {
// реализация
}
```



Пример

Например, создадим класс `Graduatestudent`:

```
class GraduateStudent extends Student {
    private String thesis;

    public GraduateStudent(String name, int mark, String thesis) {

super(name, mark); //вызываем конструктор суперкласса Student
        this.thesis = thesis;
    }

    public boolean GPA() {
        boolean result = false;
        if (getMark() >= 3)
            result = true;
        return result;
    }
//Переопределим метод toString() для класса GraduateStudent
    public String toString() {
        return "ДИПЛОМНИК: " + super.toString() + " за ВКР: " + thesis;
    }
}
```

Далее давайте рассмотрим такое понятие, как переопределение методов.

1.8.4 Полиморфизм

Если мы имеем объекты, которые принадлежат одной и той же ветви иерархии (были унаследованы), то для них можно использовать единый интерфейс, который будет для каждого объекта производить однотипное действие, но результат для каждого объекта будет различным (зависящим от этого конкретного объекта).

Полиморфизм (*polymorphism*) – *положение теории типов, согласно которому имена* (*например, переменных*) *могут обозначать объекты разных* (*но имеющих общего родителя*) *классов.*

Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

Здесь действует принцип «Один интерфейс – много методов». Благодаря полиморфизму, программы становятся менее сложными, так как для определения и выполнения однотипных действий служит единый интерфейс. Такой единый интерфейс применяется пользователем или программистом к объектам разного типа, а выбор конкретного метода для реализации соответствующей команды осуществляется компьютером в соответствии с типом объекта, для которого выполняется команда. Пожалуй, полиморфизм – это лучшее, что есть в объектно-ориентированном программировании.

При правильном применении полиморфизм, инкапсуляция и наследование комбинируются так, что создают некую среду программирования, которая обеспечивает намного более устойчивые масштабируемые программы.

1. Удачно спроектированная иерархия классов является базисом для повторного используемого кода.
2. Инкапсуляция позволяет реализациям мигрировать из проекта в проект без разрушения кода, который зависит от public-интерфейса классов.
3. Полиморфизм позволяет создавать ясный, хорошо модифицируемый и читабельный код.

Хотя принципы объектно-ориентированного программирования были рассмотрены по отдельности, работают они вместе и практически не существуют отдельно друг от друга.

1.9 Переопределение методов

Кроме перегрузки существует также *переопределение методов* (*overriding*). Подкласс наследует все переменные и методы (кроме переменных и методов с модификатором доступа `private`) из суперкласса (ближайшего родителя и всех предков). Подкласс может использовать унаследованные поля и методы в соответствии с тем, как они были определены.

Изменить работу любого из методов, унаследованных от класса-предка, класс-потомок может, описав новый метод с точно

таким же именем и параметрами. Это называется ***переопределением.***

Замещение происходит, когда класс-потомок (подкласс) определяет некоторый метод, который уже есть в родительском классе (суперклассе), таким образом новый метод заменяет метод суперкласса. У нового метода подкласса должны быть те же параметры или сигнатура, тип возвращаемого результата, что и у метода родительского класса.

Начинается такой метод с аннотации `@Override`.

Аннотации не являются программными конструкциями. Они не влияют на результат работы программы. Используются они только на этапе компиляции, после компиляции уничтожаются и не используются при выполнении.

Аннотация `@Override` не обязательна, но стоит ее иметь. В этом случае компилятор получает возможность проверить, что вы переопределили метод, а не написали новый. Таким образом можно избежать некоторых ошибок из-за невнимательности. Аннотации появились только в версии Java 1.5, и более ранние версии их не поддерживают. Всегда используйте аннотацию `@Override`, когда вы пытаетесь переопределить метод суперкласса.

Пример

Например, в классе `Student` мы переопределили метод `toString()`. Переопределим его и для класса-наследника `GraduateStudent`.

```
@Override
    public String toString() {
        return "ДИПЛОМНИК: " + super.toString() + " за ВКР: " + thesis;
         }
```

`super.toString()` – это вызов переопределенного метода `toString()` в классе `Student`.

Создав объект класса `GraduateStudent`, вызовем этот метод.

```
GraduateStudent s4 = new GraduateStudent("Иванов",3,"Автоматизированное
тестирование");
```

Далее выведем информацию о студентах на консоль (рис. 1.12).

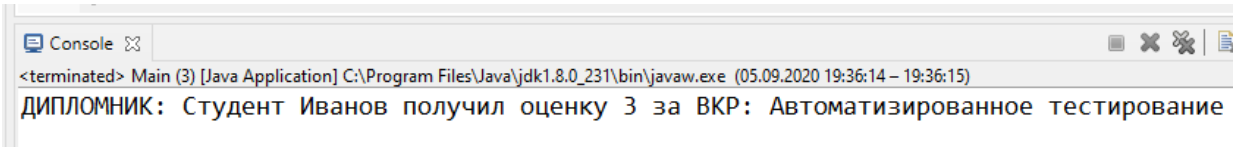


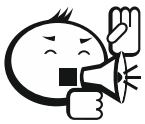
Рис. 1.12 – Вызов переопределенного метода `toString()`

Таким образом, можно выделить основные тезисы:

- Переопределение метода происходит только тогда, когда имена и сигнатуры типов этих двух методов идентичны. Если это не так, то оба метода просто перегружены.
- Поля нельзя переопределить, их можно только скрыть.
- Если пометить метод модификатором `final`, то метод не может быть переопределен.
- Переопределенные методы позволяют поддерживать полиморфизм времени выполнения.

1.10 Подстановка

Подкласс обладает всеми полями и методами своего суперкласса вследствие наследования. Это означает, что объект подкласса может делать то, что может делать объект суперкласса. В результате мы можем заменить *объектом подкласса объект суперкласса*, и все будет работать. Это называется ***замещением*** или ***подстановкой***.



Принцип подстановки Барбары Лисков – специфичное определение подтипа в объектно-ориентированном программировании. Идея Лисков о «подтипе» дает определение понятия замещения: если *S* является подтипом *T*, тогда объекты типа *T* в программе могут быть замещены объектами типа *S* без каких-либо изменений желательных свойств этой программы [3].

Так, в нашем примере классов GraduateStudent является подклассом Student. То есть можно сказать, что GraduateStudent «is-a» «является» Student (а в действительности он «является большим, чем Student»). Соотношение подкласс – суперкласс выражает так называемое соотношение «is-a» (является), которое обычно используется без перевода. С помощью подстановки можно создать объект класса GraduateStudent и присвоить его значение Student (объекту суперкласса):

```
Student sG=new GraduateStudent("Васильев", 4, "Разработка веб-приложения");
```

Теперь можно вызывать все методы, определенные в классе Student, для ссылки на sG (который все еще представляет объект GraduateStudent), например, sG.getName() и sG.getMark(). Это возможно потому, что объект подкласса обладает всеми свойствами суперкласса. Однако невозможно вызывать методы, определенные в классе GraduateStudent, для ссылки на sG, например, sG.getGPA(). Это происходит потому, что sG – это ссылка на класс Student, который не знает о методах, определенных в классе GraduateStu-dent. sG – это ссылка на класс Student, но содержит объект подкласса GraduateStudent. Ссылка на sG, однако, сохраняет внутреннюю идентичность. В нашем примере подкласс GraduateStudent переопределяет метод toString(). sG.toString() вызывает переопределенные версии из подкласса GraduateStudent вместо версий, определенных в Student (рис. 1.13):

```
System.out.println(sG.toString());
```

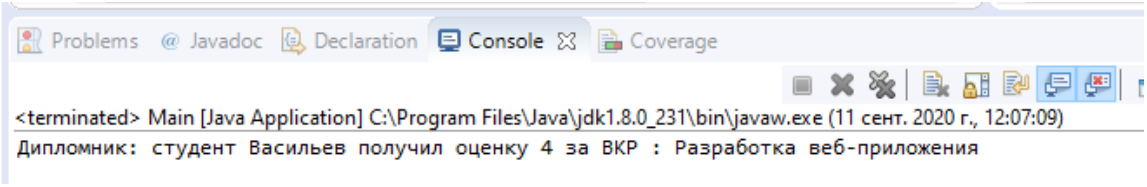


Рис. 1.13 – Вызов переопределенного метода toString()

Это происходит потому, что фактически объект sG содержит внутри объект GraduateStudent.

Выводы

1. Объекты суперкласса могут быть замещены объектами подкласса.

2. При такой замене мы можем вызывать методы, определенные в суперклассе, и не можем вызывать методы, определенные только в подклассе.
3. Однако если в подклассе переопределены унаследованные методы из суперкласса, будут вызваны переопределенные версии методов подкласса.

1.11 Апкастинг и даункастинг

Замена объекта суперкласса объектом подкласса называется ***апкастингом*** (*upcasting*) *или* ***приведением к базовому типу***.

Название действия отражает тот факт, что на UML-диаграмме подкласс изображается ниже суперкласса. Апкастинг всегда безопасен, так как объект подкласса обладает всеми свойствами суперкласса и может делать все, что может делать суперкласс. Компилятор проверяет правильность апкастинга и в противном случае выдает ошибку несовместимости типов.

Например,

```
Student s1=new GraduateStudent();// Компилятор проверяет, является ли указанное значение значением из подкласса
Student s2=new String();//Ошибка компиляции – несовместимые типы
```

Даункастинг (*downcasting*) *возвращает замещенный объект к определению через подкласс. Даункастинг – это приведение объекта суперкласса к объекту подкласса.*

Например,

```
Student s1=new GraduateStudent("Иванов");//апкастинг – безопасен
GraduateStudent gs1 = (GraduateStudent)s1;// даункастинг требует явного приведения типов
```

Даункастинг требует оператора явного приведения типов в форме префиксного оператора (новый_тип). Даункастинг не всегда безопасен и вызывает ошибку `ClassCastException` во время исполнения, если объект даункастинга не принадлежит правильному подклассу. Объект подкласса может быть заменен суперклассом, но обратное утверждение неверно.

1.12 Оператор *instanceof*

В Java имеется оператор `instanceof` типа *boolean*, который возвращает значение *true*, если объект является экземпляром данного класса. Проще говоря, оператор `instanceof` нужен, чтобы проверить, был ли объект, на который ссылается переменная `a`, создан на основе какого-либо класса `B`.

Оператор `instanceof` имеет вид:

```
a instanceof B;
```

Другими словами, оператор `instanceof` вернет значение *true*, если:

1) переменная `a` хранит ссылку на объект типа `B`;
2) переменная `a` хранит ссылку на объект, класс которого унаследован от `B`;
3) переменная `a` хранит ссылку на объект, реализующий интерфейс `B`.

Иначе оператор `instanceof` вернет значение *false*.

Простой пример из классов-оберток:

```
Object o = new Integer(3);
boolean isInt = o instanceof Integer;
```

`isInt` будет равно `true`. Объект, на который ссылается переменная `o`, является объектом класса `Integer`.

```
Integer x = new Integer(22);
  boolean isInt = x instanceof Integer;
```

Объект `x` является `Integer`, поэтому результатом будет *true*.

Вернемся к нашим студентам:

```
public class Main {
     public static void main(String[] args) {
          // s1, s2, s3 – объекты, т. е. наши тестовые студенты

               Student s1 = new Student();
          GraduateStudent s2 = new GraduateStudent();
               Student s3 = new GraduateStudent();

          //проверяем принадлежность объекта к какому-то классу
```

```
        System.out.println(s1 instanceof Student);
      System.out.println(s1 instanceof GraduateStudent);
        System.out.println(s2 instanceof Student);
      System.out.println(s2 instanceof GraduateStudent);
        System.out.println(s3 instanceof Student);
       System.out.println(s3 instanceof GraduateStudent);
            }
       }
```

Теперь рассмотрим результат на консоли (рис. 1.14).

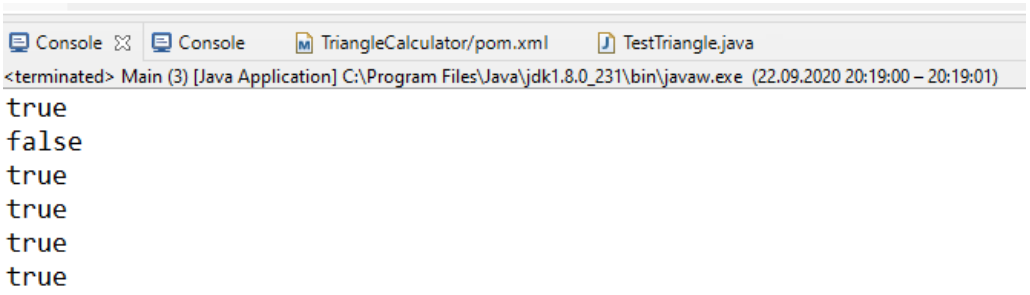
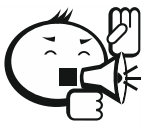


Рис. 1.14 – Результат работы метода `Instanceof`

Конструктор базового класса, если он есть, всегда вызывается первым при создании любого объекта. `Instanceof` руководствуется именно этим принципом, когда пытается определить, был ли объект `A` создан на основе класса `Б`. Если конструктор базового класса вызван, значит никаких сомнений быть не может.

Конструктор `GraduateStudent` не вызывался при создании `Student`, что логично. Ведь `GraduateStudent` – потомок `Student`, а не предок. Поэтому тут будет результат `false`.



Мы часто используем оператор `instanceof` перед понижением, чтобы проверить, принадлежит ли объект определенному типу. Оператор `instanceof` проверяет именно происхождение объекта, а не переменной.

1.13 Абстрактные классы и интерфейсы

1.13.1 Абстрактные классы

По мере изучения особенностей наследования объектов в языке Java может понадобиться создать класс, характеризующий объект обобщенно, на базе

которого впоследствии можно создать подклассы. Подклассы будут уточнять свойства объектов, формируя таким образом иерархию классов.

Самый общий класс в данном случае будет абстрактным. Он формирует «внешнюю оболочку» для подклассов, и только подклассы наполняют эту оболочку конкретным содержанием – кодом, реализующим задачи программы. Абстрактный класс, как правило, содержит один или несколько абстрактных методов.

Абстрактный метод (*abstract method*) *– это метод, реализация которого неизвестна на данный момент. Известно только то, что этот метод должен быть у всех наследников.*

Перед таким абстрактным методом указывается `abstract`, а заканчивается описание сигнатуры метода в классе традиционно – точкой с запятой:

```
abstract void method();
```

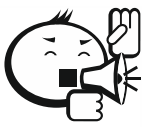
Абстрактный метод не может быть `private`, `native`, `static`, `final`.

Класс, который содержит хотя бы один абстрактный метод, называется ***абстрактным****. Иначе:* ***абстрактный класс*** *– это класс, экземпляр которого нельзя создать.*

Класс, содержащий один или несколько абстрактных методов, должен быть также объявлен как абстрактный с использованием того же спецификатора `abstract` в объявлении класса.

Важно! Абстрактный класс должен быть публичным.

Поскольку абстрактный класс не определяет реализацию полностью, у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора `new` приведет к возникновению ошибки во время компиляции. Подкласс, наследующий абстрактный класс, должен реализовать все абстрактные методы суперкласса. В противном случае он также должен быть определен как абстрактный. Таким образом, атрибут `abstract` наследуется до тех пор, пока не будет достигнута полная реализация класса.



Абстрактные классы реализуют на практике один из принципов ООП – *полиморфизм* [4].

Пример

Рассмотрим пример иерархии класса `Person` (рис. 1.15).

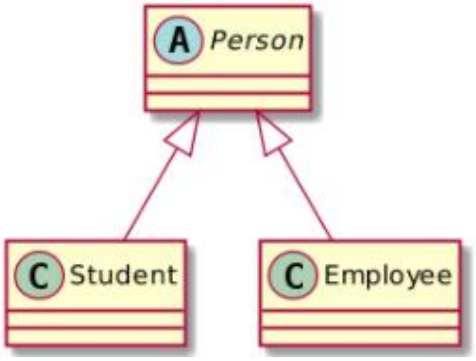


Рис. 1.15 – Иерархия класса `Person`

```
public abstract class Person
{
      private String name;
      private int mark;

  public Person(String name) {
    this.name = name;
  }
  public Person(String name,int mark) {
          this.name = name;
          this.mark=mark;
        }
  public Person(int mark) {
          this.mark = mark;
        }
  public Person () {}


  public abstract String getDescription();
}

public class Student extends Person {
```

```
    private String major;
    private int mark;
    private String name;

        public Student(String name, int mark, String major) {

        this.name=name;
        this.mark=mark;
        this.major=major;
        }

            public Student(String name) {
                this.name=name;

            }

            public Student(int mark) {
                this.mark=mark;

            }

        public Student() {
        }

          @Override
          public String getDescription() {
            return this.name + " обучается на " + this.major;
          }
}
```

В главном методе создадим объект `s3`, для которого вызовем уже переопределенный метод `getDescription()`:

```
Student s3 = new Student("Иванов",5,"ФСУ");
System.out.println(s3.getDescription());
```

При выполнении программы на консоли увидим следующий результат (рис. 1.16).

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk1.8.0_231\l
Иванов обучается на ФСУ
```

Рис. 1.16 – Вызов переопределенного метода `getDescription()`

Таким образом, абстрактный класс предоставляет шаблон для дальнейшего развития. Цель абстрактного класса – предоставить общий интерфейс (или протокол, или договор, или понимание, или соглашение об именах) для всех своих подклассов. Например, в абстрактном классе `Person` можно определить абстрактный метод `getName()`. Никакая реализация невозможна, поскольку неизвестна фамилия студента. Однако, указав сигнатуру абстрактных методов, все подклассы обязаны использовать сигнатуры этих методов. Только подклассы могут предоставлять правильные реализации. Вместе с применением полиморфизма можно проводить апкастинг объектов подкласса до `GraduateStudent` и программировать на уровне интерфейса. Разделение на интерфейс и реализацию обеспечивает лучший дизайн программного обеспечения и облегчает его расширение.

Абстрактный метод не может быть объявлен как `final`, поскольку final-метод не может быть переопределен. С другой стороны, абстрактный метод должен быть переопределен в наследнике до того, как будет использован.

Абстрактный метод не может иметь модификатор `private` (это приведет к ошибке компиляции), потому что private-метод невидим для подкласса и, таким образом, не может быть переопределен.

1.13.2 Интерфейсы

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, в Java можно наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование. Java множественное наследование не поддерживает.

Множественным наследованием *называется ситуация, когда класс наследует от двух или более классов.*

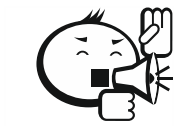
В языке Java подобную проблему позволяют решить *интерфейсы*.

Интерфейс (*interface*) *– это явно указанная спецификация набора абстрактных методов, которые должны быть представлены в классе, реализующем эту спецификацию.*

Любой *интерфейс* может иметь много реализаций. Любой класс может реализовывать несколько интерфейсов, при этом интерфейсы не входят в иерархию классов. Реализовать один и тот же интерфейс имеют право классы, никак не связанные друг с другом. Интерфейсы в Java являются ссылочными типами, как классы, но они могут содержать в себе только *константы, сигнатуры методов*.

Соглашение об именах: используйте причастие (на английском языке), состоящее из одного или нескольких слов. Каждое слово должно начинаться с заглавной буквы, например, Serializable, Movable, Clonable, Runnable и т. д.

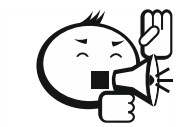
Интерфейс может быть реализован каким-либо классом либо наследоваться другим интерфейсом. Другими словами, интерфейс в Java – это 100% абстрактный суперкласс, который определяет множество методов, которые его подклассы должны поддерживать.



Интерфейсы реализуют на практике один из принципов ООП – *полиморфизм*.

Когда класс реализует интерфейс и не обеспечивает выполнения или тела метода по отношению к любому из абстрактных методов интерфейса, то класс становится *абстрактным*.

Чтобы унаследовать подклассы из интерфейса, надо использовать новое ключевое слово *implements* вместо *extends* для наследуемых подклассов, как для обычного, так и для абстрактного классов.



Важно отметить, что подкласс, наследующий интерфейс, должен переопределить все абстрактные методы, определенные в интерфейсе. В противном случае подкласс не может быть откомпилирован.

Например:

```
interface GetStudent
{
 public String getFIO();
}
//подкласс Student, наследующий интерфейс GetStudent
public class Student extends Person implements GetStudent{

	private String major;
	private int mark;
	private String name;
	private String FIO;

		public Student(String name, int mark, String major) {

		this.name=name;
		this.mark=mark;
			this.major=major;
		}



			public Student(String FIO) {
				this.FIO=FIO;

			}
			public Student(int mark) {
				this.mark=mark;

			}

		public Student() {
		}

		  @Override
		  public String getDescription() {
		    return this.name + " обучается на " + this.major;
		  }
//переопределение метода getFIO() интерфейса GetStudent
		  @Override
		 public String getFIO() {
			return FIO;
			 }
	}
```

Теперь мы можем задать и получить Ф.И.О. студента.

```
Student sF = new Student("Иванов Иван Иванович");
System.out.println(sF.getFIO());
```

При выполнении программы на консоли увидим следующий результат (рис. 1.17).

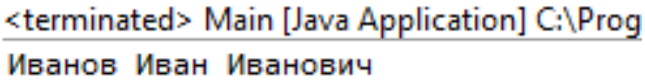


Рис. 1.17 – Вызов переопределенного метода `getFIO()`

Мы разбили наш класс на два: интерфейс и класс, унаследованный от интерфейса. Таким образом, один и тот же интерфейс могут реализовывать (наследовать) различные классы. И у каждого может быть свое собственное поведение. Это позволяет скрыть не только различные реализации, но и даже сам класс, который ее содержит (везде в коде может фигурировать только интерфейс).

У интерфейса могут быть следующие модификаторы:

- `public` (если он есть, то интерфейс доступен отовсюду, если его нет – доступен только в данном пакете);
- `abstract` (так как интерфейс всегда абстрактный, то модификатор обычно опускается);
- `strictfp` – все позже реализуемые методы должны будут работать с числами с плавающей точкой аналогично на всех машинах Java;
- интерфейсом могут расширяться многие классы;
- интерфейс может сам расширяться несколькими интерфейсами;
- интерфейс могут использовать сразу несколько классов, независимых друг от друга;
- один класс может применить множество интерфейсов.

С выпусков Java 8 интерфейсы получили новые очень интересные возможности: статические методы, методы по умолчанию и автоматическое преобразование из лямбд (функциональные интерфейсы). Начиная с JDK 8 в интерфейсах доступны статические методы, они аналогичны методам класса. С методом по умолчанию все иначе: интерфейс может отметить метод ключевым словом `default` и обеспечить реализацию для него [7]. Например:

```
public interface InterfaceWithDefaultMethods {
     void performAction();
     default void DefaulMethod() {
          // Implementation here
     }
}
```

Выводы

Зачем использовать интерфейсы? Интерфейс – это контракт (или протокол, или договор о взаимопонимании) о том, что классы могут делать. Когда класс реализует определенный интерфейс, он гарантирует реализацию всех абстрактных методов, объявленных в интерфейсе. Интерфейс определяет множество общих поведений. Классы, реализующие интерфейс, соглашаются на эти поведения и предлагают собственную реализацию этих поведений. Одним из главных применений интерфейса является предложение контракта взаимодействия для двух объектов. Как известно, класс реализует интерфейс и содержит конкретные реализации методов, объявленных в этом интерфейсе, при этом гарантируется возможность вызывать эти методы безопасно. Другими словами, два объекта могут взаимодействовать на основе контракта, определенного в интерфейсе, вместо специфических реализаций [3].

Контрольные вопросы по главе 1

1. Какие существуют способы инициализации объектов?
2. Чем отличается конструктор от простого метода?
3. Какие модификаторы может иметь класс?
4. Расскажите про модификаторы доступа. Какой принцип ООП они реализуют?
5. Можем ли мы перегрузить метод `java main()`?

2 Типы отношений между классами и объектами

Этапы разработки объектно-ориентированного представляют собой три этапа жизненного цикла:

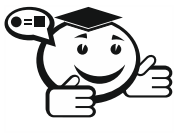
1. *Объектно-ориентированный анализ*. Определяется функциональность системы, которую приложение должно выполнять для достижения требований пользователей. Кроме того, на этом этапе определяется набор объектов, из которых будет состоять система, и описываются взаимодействие и связи между различными объектами (такое взаимодействие принимает форму сообщений между объектами); описываются процессы, которые происходят внутри каждого объекта в ответ на сообщения от других объектов, и инициация сообщений другим объектам.
2. *Объектно-ориентированное проектирование*. Формируется архитектура классов, эффективно обеспечивающая выявленную на этапе анализа функциональность приложения.
3. *Объектно-ориентированное программирование*. Определяется действительная реализация приложения средствами языка Java. Задача создания системы, обеспечивающей эффективное удовлетворение потребностей пользователей в рамках объектно-ориентированной парадигмы, является нетривиальной, и в этой работе участвуют различные специалисты, в том числе будущие пользователи системы [6].

Таким образом, необходимо смоделировать структуру и процесс функционирования программных систем до начала написания соответствующего кода. При этом непременным условием успешного завершения проекта стало построение предварительной *модели* программной системы.

Модель (*model*) – *абстракция физической системы, рассматриваемая с определенной точки зрения и представленная на некотором языке или в графической форме.*

С точки зрения общих принципов системного анализа одна и та же *физическая система* может быть представлена несколькими *моделями*. При этом назначение отдельной *модели* системы определяется характером решаемой проблемы. Основное требование к *модели* программной системы – она должна быть понятна заказчику и всем специалистам проектной группы, включая бизнес-аналитиков и

программистов. Именно для разработки такой нотации потребовались усилия группы специалистов ведущих фирм-производителей программного и аппаратного обеспечения, которые привели к появлению языка *UML*.



UML (*Unified Modeling Language – унифицированный язык моделирования*) *– язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.*

UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования в основном программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

UML позволяет разрабатывать 18 различных диаграмм, начиная от *Use Case* (диаграмма вариантов использования), заканчивая *Deployment Diagram* (диаграмма развертывания). На рисунке 2.1 представлена структура типов UML-диаграмм, которые можно использовать в процессе создания объектно-ориентированной системы.

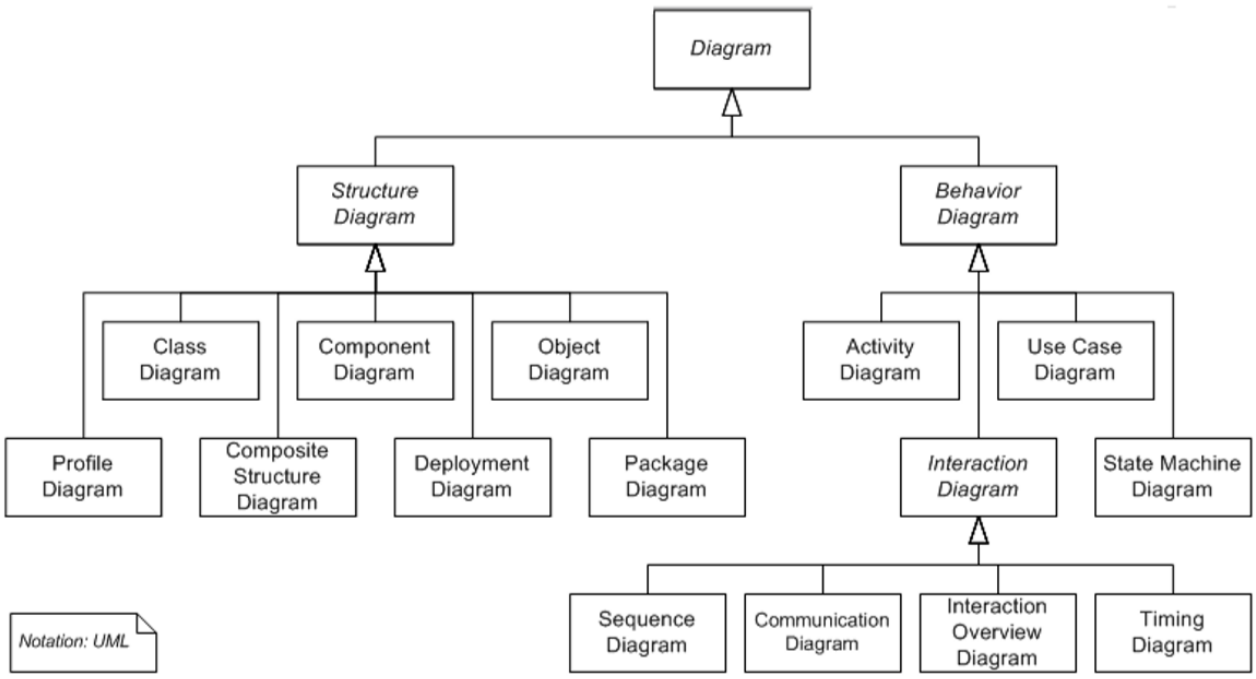


Рис. 2.1 – Типы UML-диаграмм

Более детальное рассмотрение моделей выходит за рамки настоящего пособия. В качестве примера проанализируем диаграмму классов (*Class Diagram*), состоящую из класса *Student*, которая представлена на рисунке 2.2.

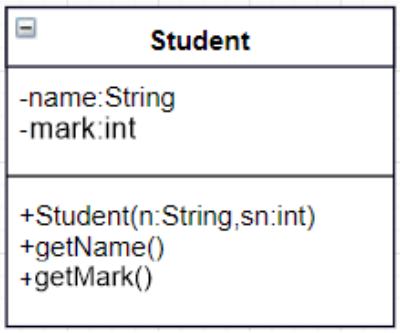


Рис. 2.2 – Диаграмма класса *Student*

На рисунке 2.2 класс представляется в виде прямоугольника, разделенного на три части. В верхней части указывается имя класса, в средней части – список переменных (полей, атрибутов) класса, в нижней части – методы класса. Символы, предшествующие переменным и методам, определяют видимость этих членов класса:

- - – private (доступны только внутри класса);
- + – public (доступны за пределами класса).

Обычно диаграмма классов состоит из многих классов и различных соединительных линий, показывающих связи между классами. Таким образом, структура программы определяется взаимодействием различных объектов и классов между собой.

Как правило, имеет место иерархия классов, а технология ООП иначе может быть названа как программирование «от класса к классу». То есть любая программа, написанная на объектно-ориентированном языке, представляет собой некоторый набор классов, связанных между собой (рис. 2.3).

Отношения *– определенная связь двух и более объектов.*

Возможны следующие связи между классами в рамках объектной модели:

1) ассоциация (*Association*);
2) агрегация (*Aggregation*);
3) композиция (*Composition*);
4) обобщение/расширение/наследование (*Inheritance*).

Рис. 2.3 – Типы отношений

2.1 Ассоциация

Базовым типом отношений является ассоциация.

Если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера «владения» или контейнеризации, то такое отношение называют ***ассоциацией*** (*association*).

Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь некоторую связь друг с другом. Эти два класса как-то связаны между собой, и мы пока не знаем точно, в чем эта связь выражена, и собираемся уточнить ее в будущем. Например, связь между объектами «Институт» и «Профессор» представлена на рисунке 2.4. Отношение ассоциации изображается линией, связывающей классы (простая, без ромбика).

Рис. 2.4 – Пример ассоциации

Мощность ассоциации (количество участников):

- «один-к-одному»;
- «один-ко-многим»;
- «многие-ко-многим».

Применительно к созданию классов на основе уже существующих используется термин «композиция» (в широком смысле), т. е. класс создается на основе существующих классов. В то же время в более узком понимании при создании таких классов используются термины «композиция» и «агрегация».

2.2 Агрегация

Агрегация являются частными случаями ассоциации. Это более конкретизированные отношения между объектами.

Отношение между классами типа «содержит» или «состоит из» называется ***агрегацией*** (*aggregation*) *или* ***включением***.

Например, «Факультет» входит в состав структуры «Института» (рис. 2.5). При агрегации объекты также имеют свой жизненный цикл, однако ограничены отношением принадлежности «*has-a*», то есть отношение «часть – целое» между двумя объектами.

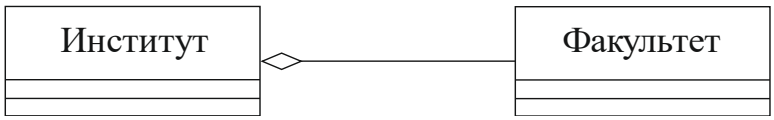


Рис. 2.5 – Пример агрегации

Агрегация изображается линией с ромбиком на стороне того класса, который выступает в качестве владельца, или контейнера. Необязательное название отношения записывается посередине линии.

Число объектов, участвующих в отношении, записывается рядом с именем роли. Запись «0..*n*» означает «от ноля до бесконечности».

Приняты следующие обозначения:

- «1..*n*» – от единицы до бесконечности;
- «0» – ноль;
- «1» – один;
- «*n*» – фиксированное количество;
- «0..1» – ноль или один.

2.3 Композиция

Композиция – еще более «жесткое» отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежат еще кому-то.

Композиция (*Composition*) *– разновидность жесткой взаимосвязи между объектами, составляющими класс. Когда объект уничтожается, объекты, составляющие его, также уничтожаются.*

Пример композиции – отношения объектов «Институт» и «Здание» (рис. 2.6).

Рис. 2.6 – Пример композиции

Разница между композицией и агрегацией заключается в том, что в случае композиции целое явно контролирует время жизни своей составной части (часть не существует без целого), а в случае агрегации целое хоть и содержит свою составную часть, время их жизни не связано (например, составная часть передается через параметры конструктора). Пример агрегации: Студент входит в Группу любителей физики. Пример композиции: Машина и Двигатель. Хотя двигатель может быть и без машины, но он вряд ли сможет быть в двух или трех машинах одновременно, в отличие от студента, который может входить и в другие группы тоже.

2.4 Наследование

Наследование (inheritance) *– это отношение между классами, при котором класс использует структуру или поведение другого* (*одиночное наследование*) *или других* (*множественное наследование*) *классов.*

Наследование вводит иерархию «общее/частное», в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение (рис. 2.7).

Использование наследования способствует уменьшению количества кода, написанного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

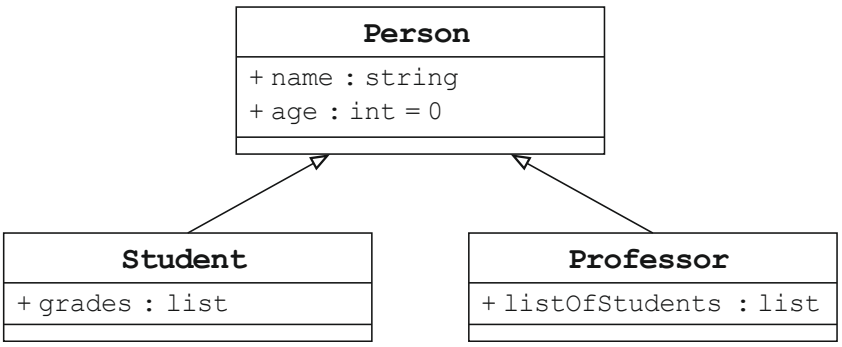
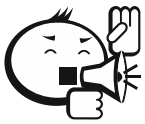


Рис. 2.7 – Пример наследования



Подкласс наследует все переменные и методы из суперкласса, включая своего ближайшего родителя, так же как и всех остальных предков, за исключением переменных и методов с модификатором `private`. Важно заметить, что подкласс не является подмножеством суперкласса. Наоборот, подкласс является «супермножеством» суперкласса. Это происходит потому, что подкласс наследует все поля и методы суперкласса и, кроме того, расширяет суперкласс, предоставляя дополнительные поля и методы.

Контрольные вопросы по главе 2

1. Чем отличается отношение композиции от ассоциации?
2. Что такое ассоциация?
3. Что такое композиция?
4. Что такое агрегация?
5. Как в языке Java поддерживается множественное наследование?

3 Введение во фреймворк «Коллекции». Обобщения

В Java имеется возможность использования массива для хранения элементов одного типа как одного из базовых типов или объектов. Однако массив не поддерживает динамическое распределение памяти, он имеет фиксированную длину, которая не может быть изменена, будучи однажды заданной. Массив является простой линейной структурой. Многие приложения могут потребовать более сложных структур данных, таких как связный список, стек, множество или деревья.

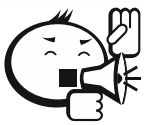
В Java динамически распределенные структуры данных, такие как *ArrayList*, *LinkedList*, *Vector*, *Stack*, *HashSet*, *HashMap*, *Hashtable*, поддерживаются единой архитектурой, которая называется фреймворк «Коллекции», определяющей общее поведение всех классов, входящих в коллекции.

3.1 Коллекции

Коллекциями (*collection*) *называют структуры, предназначенные для хранения однотипных данных.*

Фреймворк – это набор интерфейсов, который расширяет возможности для проектирования. В Java фреймворк «Коллекции» предлагает единый интерфейс для хранения, извлечения и действий с элементами коллекции независимо от лежащей в их основе фактической реализации. В Java пакет фреймворка «Коллекции» (java.util) содержит:

1) набор интерфейсов;
2) классы реализаций;
3) алгоритмы (например, сортировки или поиска).



Коллекции обладают одним важным свойством – их размер не ограничен. Выделение необходимых для коллекции ресурсов спрятано внутри соответствующего класса.

В библиотеке коллекций Java существуют два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций (рис. 3.1).

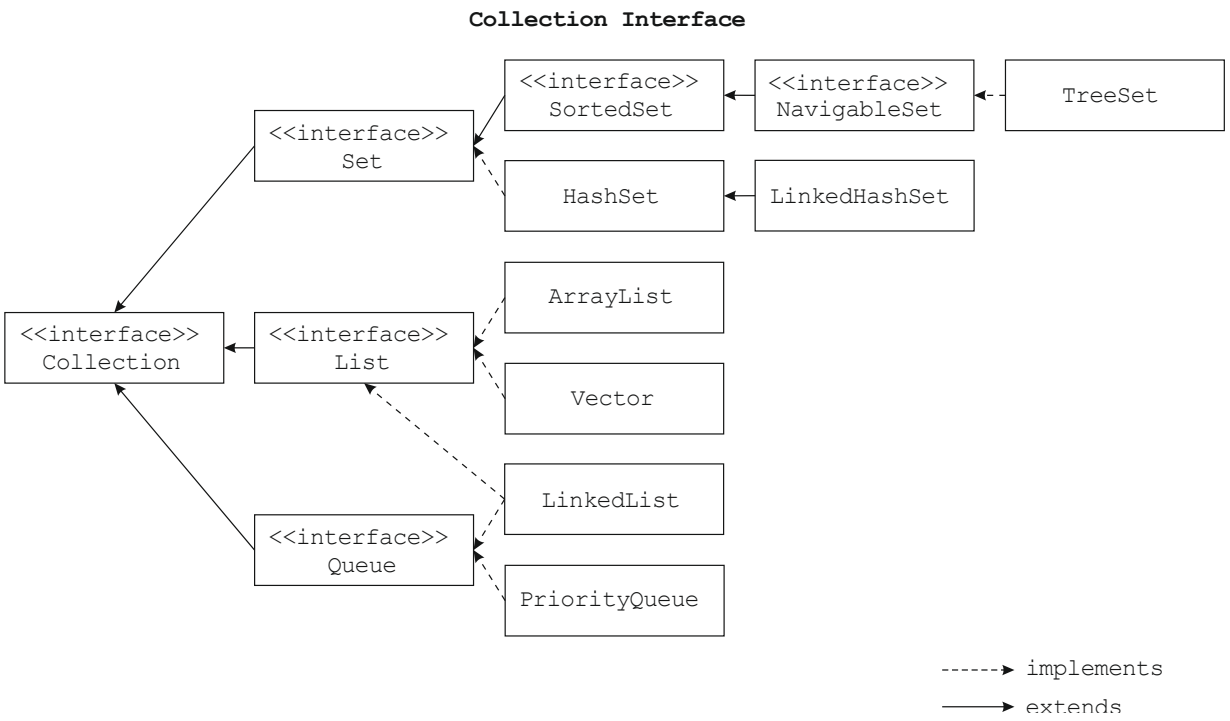


Рис. 3.1 – Иерархия `Collection`

1. `Collection` – коллекция содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (*add, addAll*), удаление (*remove, removeAll, clear*), поиск (*contains*).

Интерфейс `Collection` расширяют интерфейсы `List`, `Set` и `Queue`:

- `List` (*список*) представляет собой упорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (*sequence*). Элементы такой коллекции пронумерованы начиная от ноля, к ним можно обратиться по индексу;
- `Set` (*множество*) описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов;
- `Queue` (*очередь*) предназначен для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса `Collection` очередь предоставляет дополнительные операции вставки, получения и контроля.

2. `Map` описывает коллекцию, состоящую из пар «ключ – значение» (рис. 3.2). У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (*map*). Такую коллекцию также часто называют словарем (*dictionary*) или ассоциативным массивом (*associative array*). Словарь может содержать произвольное число элементов, никак НЕ относится к интерфейсу `Collection` и является самостоятельным.

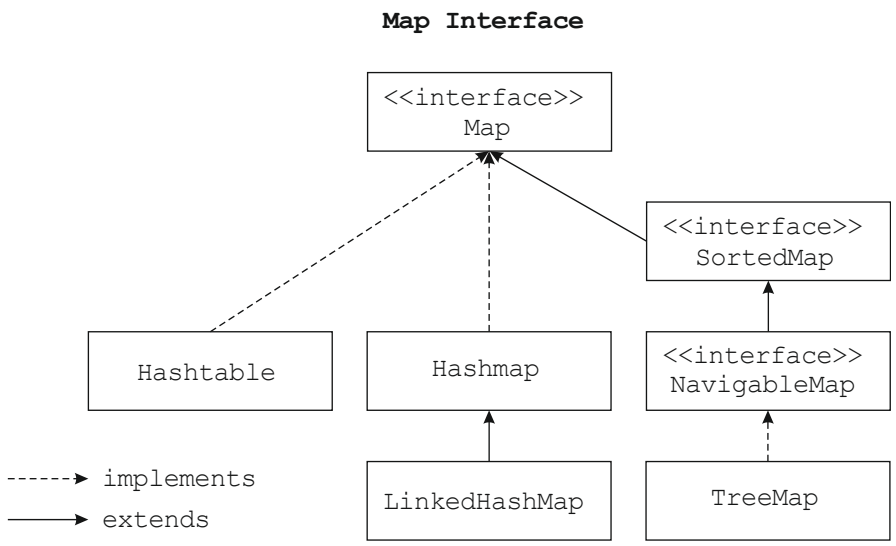


Рис. 3.2 – Иерархия `Map`

Интерфейсы `Collection` и `Map` являются базовыми, но не единственными. Их расширяют другие интерфейсы, добавляющие дополнительный функционал.

`Map` *не наследует интерфейс* `Collection`*, так как они несовместимы.* В интерфейсе `Collection` описан метод `add(Object o)`. Словари не могут содержать этот метод, потому что работают с парами ключ/значение. Также словари имеют представления `keySet`, `valueSet`, которых нет в коллекциях. Эти различия обусловили то, что интерфейс `Map` не может наследовать интерфейс `Collection` и представляет собой отдельную ветвь иерархии.

В таблице 3.1 представлены классы наборов данных [7].

Таблица 3.1 – Классы коллекций

Класс	Описание
`ArrayList`	Индексируемая последовательность, размер которой может увеличиваться и уменьшаться
`LinkedList`	Упорядоченная последовательность, обеспечивающая эффективное выполнение операций включения или удаления элемента в любой позиции
`HashSet`	Неупорядоченный набор, не допускающий дублирования элементов
`TreeSet`	Сортированное множество элементов
`EnumSet`	Набор значений нумерованного типа
`LinkedHashSet`	Множество, которое помнит порядок, в котором элементы были включены в него

PriorityQueue	Набор, обеспечивающий эффективное удаление наименьшего элемента
HashMap	Карта, которая хранит связи ключ/значение
TreeMap	Карта, в которой ключи отсортированы
EnumMap	Карта, в которой ключи принадлежат нумерованному типу
LinkedHashMap	Карта, которая помнит порядок включения элементов в нее
WeakHashMap	Карта, не используемые значения которой могут быть обработаны системой сборки мусора
IdentityHashMap	Карта, для сравнения ключей которой может быть использована операция ==

Методы интерфейсов и их описание представлены в таблице 3.2 [7].

Таблица 3.2 – Основные методы коллекций

Интерфейс	Методы
Collection	add(T e) – добавить элемент e
	clear() – очистить
	addAll(Collection col) – добавить все элементы другой коллекции, со схожим типом данных
	contains(Object o) – содержит ли коллекция элемент
	isEmpty() – возвращает, пуста ли коллекция
	remove(Object o) – удаляет элемент
	removeAll(Collection col) – удаляет все элементы, которые есть в коллекции col
	size() – возвращает количество элементов в коллекции
	containsAll(Collection col) – возвращает, содержатся ли все элементы col в коллекции
	toArray(T[] a) – возвращает массив, который содержит все элементы коллекции, на вход принимает массив, который будет заполнен ими
	retainAll(Collection col) – удаляет все элементы, не принадлежащие col
	toArray() – возвращает массив объектов, который содержит все элементы коллекции
Set	Содержит все операции интерфейса Collection, но некоторые из них имеют другой смысл. Например, операция add добавляет только уникальные элементы, зато операции поиска элемента происходят быстрее, чем в списке
List	Все методы интерфейса Collection, а также следующие: • get(int index) – получает элемент по индексу; • add(int index, T e) – вставляет элемент в позицию; • indexOf(Object obj) – возвращает первое вхождение элемента в список; • lastIndexOf(Object obj) – возвращает последнее вхождение; • set(int index, T e) – заменяет элемент в позиции index; • subList(int from, int to) – возвращает новый список, представляющий собой часть главного

Map	size() – возвращает количество элементов
	containsKey(Object key) – проверяет наличие ключа
	containsValue(Object value) – проверяет наличие значения в ассоциативном массиве
	get(Object key) – возвращает значение по ключу
	put(K key, V value) – кладет в ассоциативный массив значение value по ключу key. В случае наличия уже такого ключа происходит замена значения
	values() – возвращает значения всех элементов в виде коллекции
	remove(Object key) – удаляет элемент с ключом key, возвращая значение этого элемента (вернет null в случае отсутствия)
	clear() – удаляет все элементы из массива
	isEmpty() – возвращает, не пуст ли массив
	containsKey(Object key) – проверяет наличие ключа

3.2 Перебор элементов коллекций

1. Перебор с помощью итератора.

Интерфейс Collection расширяет интерфейс Iterable, у которого есть только один метод iterator(). Объект типа Iterator может использоваться для последовательного перебора элементов коллекции.

Интерфейс Iterator имеет следующее определение:

```
public interface Iterator <E> {
E next();
boolean hasNext();
void remove();
}
```

Реализация интерфейса предполагает, что с помощью вызова метода next() можно получить следующий элемент. С помощью метода hasNext() можно узнать, есть ли следующий элемент и не достигнут ли конец коллекции. И если элементы еще имеются, то hasNext() вернет значение true. Метод hasNext() следует вызывать перед методом next(), так как при достижении конца коллекции метод next() выбрасывает исключение NoSuchElement-Exception и метод remove() удаляет текущий элемент, который был получен последним вызовом next().

Используем итератор для перебора коллекции ArrayList. Создадим список студентов students.

```
import java.util.*;
ArrayList<Student> students = new ArrayList<Student>();
```

```
students.add(new Student("Иванов Иван Иванович"));
students.add(new Student("Петров Петр Петрович"));
students.add(new Student("Сидоров Сергей Сергеевич"));

Iterator<Student> iter = students.iterator();
     while(iter.hasNext()) {
System.out.println(iter.next().getFIO());
     }
```

В результате итератор переберет все элементы коллекции, пока не пусто. Одновременно с методом `next()` вызвали `getFIO()`, чтобы получать только Ф.И.О. (рис. 3.3). Если просто использовать `students.next()`, то получим результат вывода метода `toString()`, переопределенного в классе `Student`.

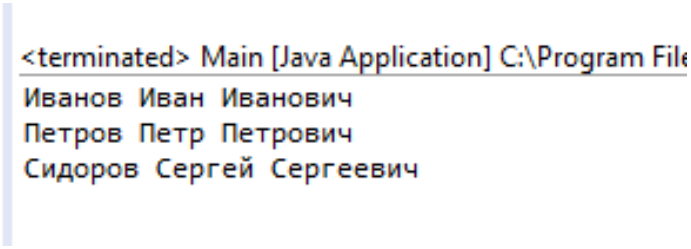


Рис. 3.3 – Вывод элементов коллекции

Интерфейс `Iterator` предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор – интерфейс `ListIterator`. Данный итератор используется классами, реализующими интерфейс `List`, то есть классами `LinkedList`, `ArrayList` и др.

Интерфейс `ListIterator` расширяет интерфейс `Iterator` и определяет ряд дополнительных методов:

- `void add(E obj)`: вставляет объект `obj` перед элементом, который должен быть возвращен следующим вызовом `next()`;
- `boolean hasNext()`: возвращает `true`, если в коллекции имеется следующий элемент, иначе возвращает `false`;
- `boolean hasPrevious()`: возвращает `true`, если в коллекции имеется предыдущий элемент, иначе возвращает `false`;
- `E next()`: возвращает следующий элемент; если такого нет, то генерируется исключение `NoSuchElementException`;
- `E previous()`: возвращает предыдущий элемент; если такого нет, то генерируется исключение `NoSuchElementException`;

- `int nextIndex()`: возвращает индекс следующего элемента; если такого нет, то возвращается размер списка;
- `int previousIndex()`: возвращает индекс предыдущего элемента; если такого нет, то возвращается число –1;
- `void remove()`: удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов `next()` или `previous()`, иначе будет сгенерировано исключение `IlligalStateException`;
- `void set(E obj)`: присваивает текущему элементу, выбранному вызовом методов `next()` или `previous()`, ссылку на объект `obj`.

Добавим в наш пример новый итератор `ListIterator`.

```
ListIterator<Student> listIter = students.listIterator();

        while(listIter.hasNext()) {
        System.out.println(listIter.next().getFIO());
                        }
        // сейчас текущий элемент - Сидоров Сергей Сергеевич
        // изменим значение этого элемента
        listIter.set(new Student("Васильев Василий Петрович"));
        // пройдемся по элементам в обратном порядке
        //новый список
        System.out.println("новый список");
        while(listIter.hasPrevious()) {
        System.out.println(listIter.previous().getFIO());
        }
```

В результате итератор переберет все элементы коллекции, пока не пусто. Затем пройдется по элементам в обратном порядке. Получится два списка (рис. 3.4).

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk1.8.0_
Иванов Иван Иванович
Петров Петр Петрович
Сидоров Сергей Сергеевич
новый список
Васильев Василий Петрович
Петров Петр Петрович
Иванов Иван Иванович
```

Рис. 3.4 – Вывод элементов коллекции с помощью итератора `ListIterator`

2. Перебор с помощью цикла `forEach`.

В Java 8 у интерфейса `Iterable` появился метод `forEach`, принимающий лямбда-выражение и применяющий это выражение на каждый элемент коллекции:

```
students.forEach(n -> System. out.println(n.getFIO()));
```

3. Перебор с помощью расширенного цикла `for`.

```
for(Student n: students) {
      System.out.println(n.getFIO());
}
```

Также для перебора можно использовать простые конструкции циклов.

Рассмотрим пример перебора на примере очереди (`Queue`), у которой есть дополнительные методы по добавлению, извлечению и проверке элементов. Чаще всего порядок выдачи элементов соответствует *FIFO* (*firstin*, *first-out*), но в общем случае определяется конкретной реализацией. Очереди не могут хранить `null`. У очереди может быть ограничен размер.

Основные методы очереди:

- `element();` – возвращает, но не удаляет головной элемент очереди;
- `offer(E o);` – добавляет в конец очереди новый элемент и возвращает `true`, если вставка удалась;
- `peek();` – возвращает первый элемент очереди, не удаляя его;
- `poll();` – возвращает первый элемент и удаляет его из очереди;
- `remove();` – возвращает и удаляет головной элемент очереди.

```
Queue<Student> queue = new LinkedList<Student>();
      queue.offer(new Student("Иванов Иван Иванович"));
      queue.offer(new Student("Петров Петр Петрович"));
      queue.offer(new Student("Сидоров Сергей Сергеевич"));
     queue.offer(new Student("Васильев Василий Петрович"));
          for (Student str : queue) {
                    System.out.println(str.getFIO());
          }
               System.out.println("head:"+queue.peek().getFIO());
//возвращает, но не удаляет первый элемент очереди
```

Результат на консоли представлен на рисунке 3.5.

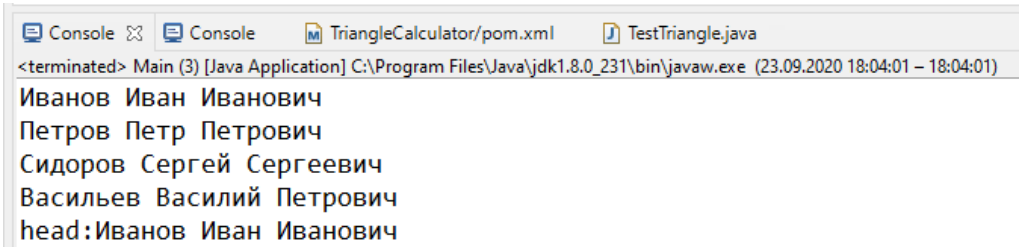


Рис. 3.5 – Вывод элементов коллекции `Queue`

Интерфейс `Deque` позволяет реализовать двунаправленная очередь, разрешающая вставку и удаление элементов в два конца очереди.

Методы `addFirst(e)`, `addLast(e)` вставляют элементы в начало и в конец очереди соответственно. Метод `add(e)` унаследован от интерфейса `Queue` и абсолютно аналогичен методу `addLast(e)` интерфейса `Deque`.

```
Deque<Student> deque = new LinkedList<Student>();
            deque.offer(new Student("Иванов Иван Иванович"));
            deque.offer(new Student("Петров Петр Петрович"));
            deque.addFirst(new Student("Сидоров Сергей Петрович"));
             //головной элемент
            deque.offer(new Student("Васильев Василий Петрович"));
      for (Student str : deque) {
            System.out.println(str.getFIO());
            }
System.out.println("head:"+deque.peek().getFIO()); //возвращает, но не удаляет первый элемент очереди
```

Результат на консоли представлен на рисунке 3.6.

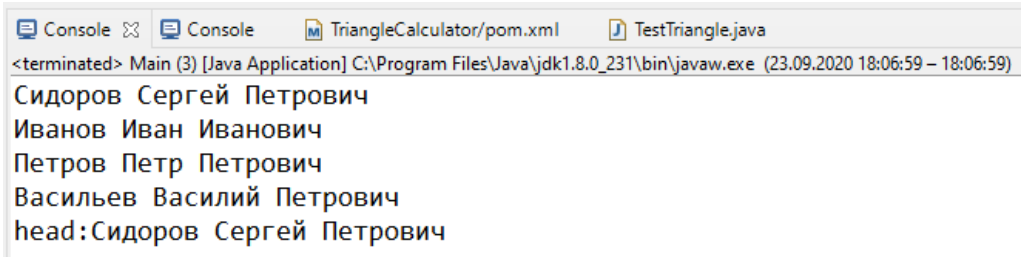


Рис. 3.6 – Вывод элементов коллекции `Deque`

Сортировку и сравнение коллекций рассмотрим в главе о лямбда-выражениях.

Выводы

Главные преимущества коллекций:

- уменьшаются затраты времени на написание кода;
- улучшается производительность благодаря использованию высокоэффективных алгоритмов и структур данных;
- коллекции являются универсальным способом хранения и передачи данных, что упрощает взаимодействие разных частей кода;
- простота в изучении, потому что необходимо выучить только самые верхние интерфейсы и поддерживаемые операции;
- реализуется поддержка многопоточного доступа.

3.3 Обобщения

Java Generics – это одно из самых значительных изменений за всю историю языка Java. «Дженерики», доступные с Java 5, сделали использование Java Collection Framework проще, удобнее и безопаснее. Ошибки, связанные с некорректным использованием типов, теперь обнаруживаются на этапе компиляции. Да и сам язык Java стал еще безопаснее [8]. Несмотря на кажущуюся простоту обобщенных типов, многие разработчики сталкиваются с трудностями при их использовании.

Дженерики **(*обобщения*)** *– это особые средства языка Java для реализации обобщенного программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.*

На сайте Oracle дженерикам посвящен отдельный раздел «Lesson: Generics» [9]. По сути, обобщения – это параметризованные типы.

С помощью параметризованных типов можно объявлять классы, интерфейсы и методы, в которых тип данных указан в виде параметра. Обобщения добавили в язык безопасность типов. Основным мотивом введения дженериков в Java было неудобство работы с нетипизированными коллекциями. Использование нетипизированных коллекций приводит к большим трудозатратам, нежели применение типизированных, причем могут порождаться ошибки. Эти ошибки обнаруживаются только во время исполнения. Можно всякий раз, когда требуется коллекция, создавать ее типизированный вариант, что избавляет от проблем, связанных с применением коллекций, но приводит к большому объему дублированию кода. Дженерики позволяют избежать подобных проблем за счет введения

специальных параметров типов и обобщения реализации классов и методов. Они позволяют не создавать отдельную копию типизированной коллекции для измененного типа элемента, а создавать единую обобщенную реализацию, в которой тип элемента заменен параметром типа, и впоследствии возложить работу по созданию специализированных коллекций на компилятор.

Дженерики позволяют передавать тип информации компилятору в форме <*тип*>. Таким образом, компилятор может выполнить все необходимые действия по проверке типов во время компиляции, обеспечивая безопасность по приведению типов во время выполнения [3].

Обобщенные типы *– это механизм компилятора, посредством которого можно некоторым стандартным образом создавать* (*и использовать*) *типы* (*классов, интерфейсов и т. п.*)*, получая единый код и параметризуя* (*или обобщая*) *все остальное.*

Например:

```
List list = new ArrayList();
list.add("Hello");
String text = list.get(0) + ", world!";
System.out.print(text);
```

Этот код выполнится, но если удалить из кода конкатенацию со строкой ", world!", то получим ОШИБКУ КОМПИЛЯЦИИ:

```
error: incompatible types: Object cannot be converted to String
```

Все дело в том, что `List` хранит список объектов типа `Object`. Так как `String` – наследник для класса `Object`, то требует явного приведения, чего мы не сделали. А при конкатенации для объекта будет вызван статический метод `String.valueOf(obj)`, который в итоге вызовет метод `toString()` для `Object`. То есть `List` у нас содержит `Object`. Выходит, там, где нам нужен конкретный тип, а не `Object`, нам придется самим делать приведение типов:

```
List list = new ArrayList();
list.add("Hello!");
list.add(123);
for (Object str : list) {
    System.out.println((String)str);
}
```

Однако в данном случае `List` принимает список объектов и хранит не только `String`, но и `Integer`. Но самое страшное, в этом случае компилятор не увидит ничего плохого, и мы получим ошибку уже ВО ВРЕМЯ ВЫПОЛНЕНИЯ (Runtime), т. е. ошибку выполнения, а не компиляции. Это уже гораздо хуже! Ошибка будет следующей:

```
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

Чтобы указать компилятору, какие типы мы собираемся использовать, в Java SE 5 и ввели дженерики. Исправим наш пример:

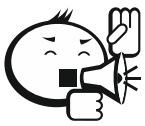
```
List<String> list = new ArrayList<>();
list.add("Hello!");
list.add(123);
for (Object str : list) {
    System.out.println(str);
}
```

Нам уже больше не нужно выполнять приведение к `String`.

В дженериках вместо передачи аргументов компилятору передается тип информации, заключив его в угловые скобки <>.

Теперь компилятор не даст скомпилировать класс, пока мы не удалим добавление `123` в список, т. к. это `Integer`. Будет показано следующее сообщение:

```
The method add(int, String) in the type List<String> is not applicable for the arguments (int)
```



Что такое *безопасность типов*? Это просто гарантия компилятора, что если правильные типы используются в правильных местах, то исключения *ClassCastException* во время выполнения не должно быть.

Согласно спецификации языка Java:

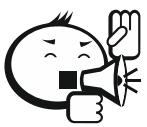
- Тип переменной является безусловным идентификатором. Переменные типа вводятся объявлениями универсального класса, объявлениями универсального интерфейса, объявлениями универсального метода и объявлениями универсального конструктора.
- Класс является *общим*, если он объявляет одну или несколько переменных типа. Эти переменные типа известны как параметры типа класса.

Он определяет одну или несколько переменных типа, которые действуют как параметры. Объявление универсального класса определяет набор параметризованных типов, по одному для каждого возможного вызова раздела параметров типа. Все эти параметризованные типы используют один и тот же класс во время выполнения.

- Интерфейс является *общим*, если он объявляет одну или несколько переменных типа. Эти переменные типа известны как параметры типа интерфейса. Интерфейс определяет одну или несколько переменных типа, которые действуют как параметры. Объявление универсального интерфейса определяет набор типов, по одному для каждого возможного вызова раздела параметров типа. Все параметризованные типы используют один и тот же интерфейс во время выполнения.
- Метод является *общим*, если он объявляет одну или несколько переменных типа. Эти переменные типа известны как параметры формального типа метода. Форма списка параметров формального типа идентична списку параметров типа класса или интерфейса.
- Конструктор может быть объявлен как универсальный, независимо от того, является ли класс, в котором объявлен конструктор, универсальным. Конструктор является универсальным, если он объявляет одну или несколько переменных типа. Эти переменные типа известны как параметры формального типа конструктора. Форма списка параметров формального типа идентична списку параметров типа универсального класса или интерфейса [10].

Наиболее часто используются следующие имена параметров типа:

- `E` – элемент (широко используется в Java Collections Framework, например, `ArrayList`, `Set` и т. д.);
- `K` – ключ (используется в `Map`);
- `T` – тип;
- `B` – значение (используется в `Map`);
- `S`, `U`, `V` и т. д. – 2-й, 3-й, 4-й… тип.



Обобщения действуют только со ссылочными типами!

Еще один важный термин в дженериках Java – «стирание типа». Стирание типов гарантирует, что для параметризованных типов не будут созданы новые

классы; следовательно, универсальные шаблоны не несут дополнительных затрат времени выполнения.

Обобщения в Java были добавлены для обеспечения проверки типов во время компиляции и не используются во время выполнения, поэтому компилятор Java использует функцию стирания типов, чтобы удалить весь код проверки типов универсальных типов в байтовом коде и при необходимости вставить приведение типов.

Разберемся на примере.

```
List<Integer> list = new ArrayList<Integer>();
      list.add(1000);      //строка без ошибки
      list.add("lokesh"); //ошибка компиляции, т. к. добавляем String в
Integer;
```

Когда вы напишете приведенный выше код и скомпилируете его, вы получите следующую ошибку:

```
      The method add(Integer) in the type List<Integer> is not ap-
plicable for the arguments (String)
```

Компилятор предупредил. Это и есть цель дженериков, т. е. *безопасность типа*.

Еще одна польза от дженериков – это получение байтового кода после удаления второй строки из приведенного выше примера. Если вы сравните байт-код приведенного выше примера с дженериками и без них, то никакой разницы не будет. Ясно, что компилятор удалил всю информацию о дженериках. Итак, приведенный выше код очень похож на код ниже без универсальных шаблонов.

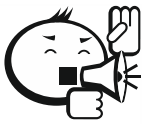
```
      List list = new ArrayList();
      list.add(1000);
```

Вся информация о типах стирается компилятором с помощью функции «стирания типов». Стирание типов и дженерики сделаны так, чтобы обеспечить обратную совместимость со старыми версиями JDK, но при этом дать возможность помогать компилятору с определением типа в новых версиях Java.

Еще одно интересное понятие о дженериках – «сырой тип» – *Raw type*.

Сырой тип (*raw type*) *– это имя обобщенного класса или интерфейса без аргументов типа.*

Можно часто увидеть использование сырых типов в старом коде, поскольку многие классы (например, коллекции) до Java 5 были необобщенными, например, с использованием `List` вместо `List<String>`. При использовании сырых типов получается то же самое поведение, которое было до введения обобщений в Java, т. е. «сырые типы» используются для обратной совместимости. Их использование в новом коде не рекомендуется, поскольку использование универсального класса с аргументом типа обеспечивает более строгую типизацию, что, в свою очередь, может улучшить понятность кода и привести к более раннему выявлению потенциальных проблем.



Новые версии языка программирования Java запрещают использование сырых типов.

Теперь давайте разберемся, как можно более подробно применить универсальные шаблоны.

Отдельно рассмотрим:

- универсальные классы (*generic class*) и интерфейсы;
- общие методы и универсальные конструкторы;
- подстановочные символы (*wildcard*).

3.3.1 Универсальные классы (*generic class*) и интерфейсы

Типизировать можно не только методы, но и сами классы.

*Класс называется **универсальным**, если он объявляет одну или несколько переменных типа T.*

Эти типы переменных известны как параметры `T` типа класса Java.

Пример

Давайте разберемся в этом на примере о студентах и преподавателей.

Есть класс `Student`:

```
class Student {
    private String name;
    private Integer age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
    }

    public void setAge(int age) {
        this.age=age;
    }


        public int getAge() {
            return age;
        }
        @Override
        public String toString() {
            return "студент "+name+",  "+age;
        }
     }
```

Есть класс `Teacher`:

```
class Teacher {
    private String name;
    private Integer age;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
    }

    public void setAge(int age) {
        this.age=age;
    }

        public int getAge() {
            return age;
```

```
    }
    @Override
    public String toString() {
        return "преподаватель "+name+",  "+age;
    }
}
```

Создадим класс `School` и укажем его как обобщенный (универсальный).

С помощью буквы `T` в определении класса class `School <T>` мы указываем, что данный тип `T` будет использоваться этим классом. Параметр `T` в угловых скобках называется *универсальным параметром*, так как вместо него можно подставить любой тип. При этом пока мы не знаем, какой именно это будет тип: `String`, `int` или какой-то другой. Причем буква `T` выбрана условно, это может быть и любая другая буква или набор символов.

```
class School<T> {
        private T person;

        public T getPerson() {
            return person;
        }

        public void setPerson(T person) {
            this.person = person;
        }
}
```

Сначала давайте протестируем класс `Student`.

```
public class Main {

    public static void main(String[] args) {
        //создадим тестового студента
        Student student = new Student();
        student.setName("Сидоров Василий");
        student.setAge(19);
        //Создадим объект School и поместим в него тип Student
  //Во время вызова, School<Student>, конкретный тип <Student> или параметр фактического типа, заменил параметр формального типа <T>.

        School<Student> school_stu = new School<>();
        // поместим объект Student в School
        school_stu.setPerson(student);
        // Здесь мы вызываем метод get() для объекта Student в соответствии с типом обобщения.
```

```
        Student person_stu = school_stu.getPerson();
        // вызываем переопределенный метод toString() в классе Student
             System.out.println(person_stu.toString());
    }

      }
```

При определении переменной и создании объекта после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо универсального параметра. При этом надо учитывать, что они работают только с объектами, но не работают с примитивными типами. То есть мы можем написать `School<Student>`, но не можем использовать тип `int` или `double`, например, `School <int>`.

Вместо примитивных типов надо использовать классы-оберт-ки: `Integer` вместо `int`, `Double` вместо `double` и т. д.

На консоли будет следующий результат (рис. 3.7).

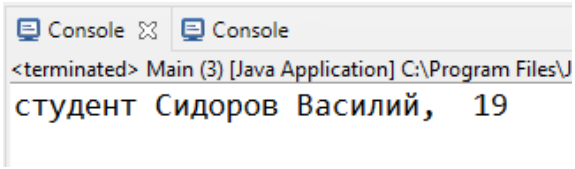


Рис. 3.7 – Пример использования универсального типа в классе `Student`

Повторим то же самое с классом `Teacher`.

```
// Создаем тестового преподавателя
        Teacher teacher = new Teacher();
        teacher.setName("Морозова Юлия");
        teacher.setAge(39);
        //Создадим объект School и помещаем в него тип Teacher
        School<Teacher> school_tea = new School<>();
        //  поместим объект Teacher в School
        school_tea.setPerson(teacher);
        Teacher person_tea = school_tea.getPerson();
         System.out.println(person_tea.toString());
```

На консоли будет следующий результат (рис. 3.8).

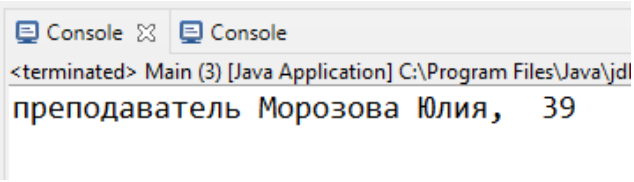


Рис. 3.8 – Пример использования универсального типа в классе `School`

Таким образом, нам не нужно создавать много классов `School`. Мы просто добавляем разные параметры в создание объектов `School`.

В приведенном выше примере мы создаем объекты `School`. Мы передаем разные параметры, почему тогда полученные объекты одинаковы? Ответ в том, что мы на самом деле не создаем разные типы, то есть мы передали разные аргументы параметров, но в памяти есть только один, который является исходным базовым типом (`School`).

```
System.out.println("school_stu Class:" + school_stu.getClass());
System.out.println("school_tea Class:" + school_tea.getClass());
System.out.println(school_stu.getClass() == school_tea.getClass());
```

На консоли будет следующий результат (рис. 3.9).

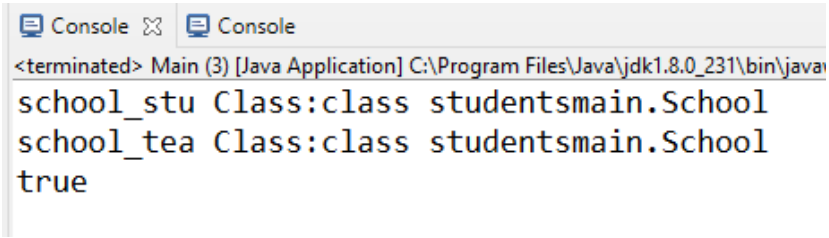


Рис. 3.9 – Пример реализации одного типа объекта,
но с разными параметрами

Логически универсальные типы можно рассматривать как несколько типов, и все они являются одними и теми же базовыми типами во время выполнения.

Интерфейсы, как и классы, также могут быть *обобщенными*. Создадим обобщенный интерфейс `Accountable` и используем его в классе `Student`:

```
interface Accountable<T>{
    T getId();
    int getSum();
```

```
    void setSum(int sum);
}

 class Student implements Accountable<String> {
    private String name;
    private int age;
    //добавили новые поля
    private String id;
    private int sum;

    Student(String id, int sum){
        this.id = id;
        this.sum = sum;
    }


    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
    }

    public void setAge(int age) {
        this.age=age;
    }


        public int getAge() {
            return age;
        }
        @Override
        public String toString() {
            return "студент "+name+",  "+age;
        }
        //реализуем методы интерфейса
        @Override    public String getId() { return id; }
        @Override    public int getSum() { return sum; }
        @Override    public void setSum(int sum) { this.sum = sum; }

     }

public class Main {
     public static void main(String[] args) {
          //создадим тестовых студентов

     Accountable<String> acc1 = new Student("1235rwr", 5000);
     Student acc2 = new Student("2373", 4300);
```

```
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());
//проверим принадлежность к классу Student
        System.out.println("acc1 Class:" + acc1.getClass());
        System.out.println("acc2 Class:" + acc2.getClass());
        System.out.println(acc1.getClass() == acc2.getClass());
    }

        }
```

На консоли будет следующий результат (рис. 3.10).

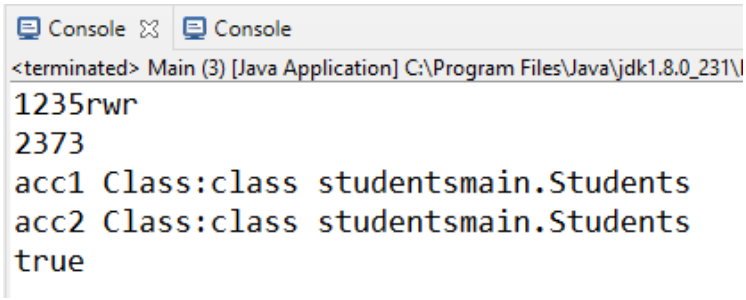


Рис. 3.10 – Пример реализации обобщенного интерфейса

При реализации подобного интерфейса есть две стратегии. В данном случае реализована первая стратегия, когда при реализации для универсального параметра интерфейса задается конкретный тип, например, в данном случае это тип `String`. Тогда класс, реализующий интерфейс, жестко привязан к этому типу.

Вторая стратегия представляет определение обобщенного класса, который также использует тот же универсальный параметр. Сделаем класс `Student` также обобщенным:

```
interface Accountable<T>{
    T getId();
    int getSum();
    void setSum(int sum);
}


 class Student<T> implements Accountable<T> {
    private String name;
    private int age;
    //добавили новые поля
    private T id;
    private int sum;

    Students(T id, int sum){
        this.id = id;
```

```
        this.sum = sum;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
    }

    public void setAge(int age) {
        this.age=age;
    }


        public int getAge() {
            return age;
        }
        @Override
        public String toString() {
            return "студент "+name+",  "+age;
        }
        //реализуем методы интерфейса
        public T getId() { return id; }
        public int getSum() { return sum; }
        public void setSum(int sum) { this.sum = sum; }

     }

public class Main {
     public static void main(String[] args) {
          //создадим тестовых студентов

          Students<String> acc1 = new Students<String>("1235rwr", 5000);
          Students<String> acc2 = new Students<String>("2373", 4300);
        System.out.println(acc1.getId());
        System.out.println(acc2.getId());

//проверим принадлежность к классу Student
      System.out.println("acc1 Class:" + acc1.getClass());
      System.out.println("acc2 Class:" + acc2.getClass());
      System.out.println(acc1.getClass() == acc2.getClass());
   }
     }
```

На консоли будет следующий результат (рис. 3.11).

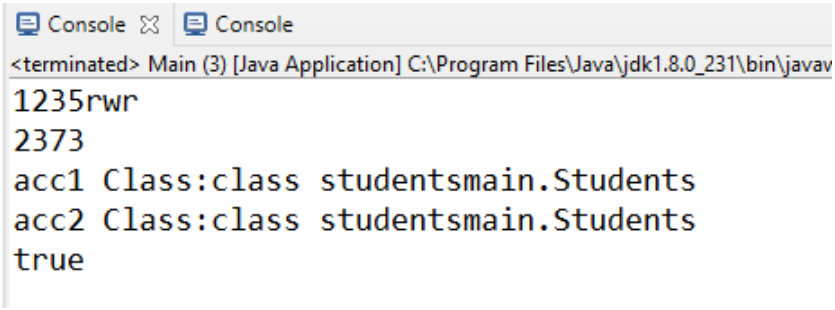


Рис. 3.11 – Пример реализации обобщенного класса

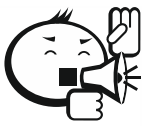
Также интерфейс `Comparable` – отличный пример *Generics* в интерфейсах. Он записан так:

```
package java.lang;
      import java.util.*;

      public interface Comparable<T> {
          public int compareTo(T o);
      }
```

3.3.2 Дженерик-методы и универсальные конструкторы

Иногда мы не хотим, чтобы весь класс был параметризован, в этом случае мы можем создать дженерик-метод. Поскольку конструктор – это особый вид метода, мы также можем использовать тип обобщения в конструкторах.

Универсальные методы очень похожи на универсальные классы. Они отличаются друг от друга только в одном аспекте: информация об области или типе находится только внутри метода.



Универсальные методы вводят свои собственные параметры типа.

Ниже приведен пример универсального метода, который можно использовать для поиска всех вхождений параметров типа в списке переменных.

```
public class Main {

       public static <T> int countAll(T[] list, T element) {
      int count = 0;
      if (element == null) {
      for ( T listElement : list )
      if (listElement == null)
      count++;
      }
```

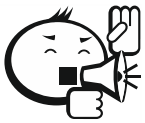
```
    else {
    for ( T listElement : list )
    if (element.equals(listElement))
    count++;
    }
    return count;
    }

    public static void main(String[] args) {

        //создаем массив целых чисел
        Integer [] test = {1,4,3,4};
        //получаем кол-во вхождений числа 4
        System.out.println(countAll(test,4));

}

    }
```

Если вы передадите список `String` для поиска в этом методе, он будет работать нормально.

```
    //создаем массив  чисел
            String [] test = {"1","4","3","4"};
        //получаем кол-во вхождений числа 4
        System.out.println(countAll(test,"4"));
```

Ответ тоже будет 2.

Конструктор – это блок кода, который инициализирует вновь созданный объект.



Конструктор напоминает метод, но это не метод, поскольку он не имеет типа возвращаемого значения.

Конструктор имеет то же имя, что и класс, и выглядит так в коде Java. Теперь давайте вернемся к нашим студентам и разберемся, как это работает.

```
class Students <T> {
    private T name;
    private T age;

    // Generic constructor
    public Students(T name, T age) {
```

```
        this.age=age;
        this.name=name;

    }


    public T getName() {
        return name;
    }

    public T getAge() {
            return age;
        }

        @Override
        public String toString() {
            return "студент "+name+",  "+age;
        }

     }
```

В приведенном выше примере конструктор класса Students имеет информацию о типе. Таким образом, у вас может быть экземпляр измерения со всеми атрибутами только одного типа.

```
public class Main {

          public static void main(String[] args) {
          //создадим массив тестовых студентов
          Students student1 = new Students("Иванов", 18);
        Students student2 = new Students("Петров", "25");

        System.out.println(student1.getName());
        System.out.println(student1.getAge());
        System.out.println(student1.toString());

        System.out.println(student2.getName());
        System.out.println(student2.getAge());
        System.out.println(student2.toString());
}

     }
```

На консоли будет следующий результат (рис. 3.12).

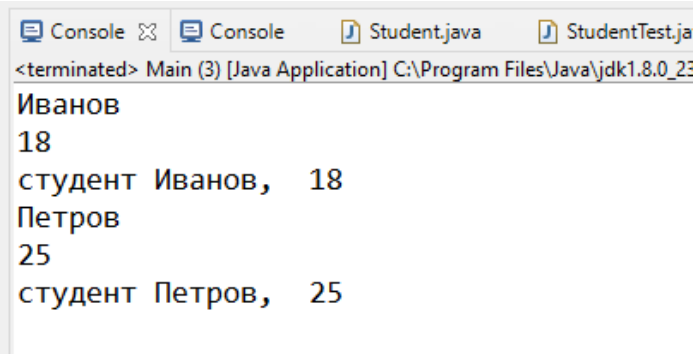


Рис. 3.12 – Пример реализации универсального конструктора

3.3.3 Подстановочные символы (*wildcard*)

Знак вопроса (*?*) – *это подстановочный знак в универсальных шаблонах, представляющий неизвестный тип.*

Подстановочный знак можно использовать как тип параметра, поля или локальной переменной, а иногда и как возвращаемый тип. Мы не можем использовать подстановочные знаки при вызове универсального метода или создании экземпляра универсального класса.

Запись вида <? extends ...> или <? super ...> *называется wildcard или символом подстановки, с верхней границей* (*extends*) *или с нижней границей* (*super*).

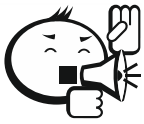
Например, `List<? extends Number>` из иерархии классов на рисунке 3.13 может содержать объекты, класс которых является `Number` или наследуется от `Number`. `List<? super Number>` может содержать объекты, класс которых `Number` или у которых `Number` является наследником (супертип от `Number`).

«? extends Number» обозначает «любой тип, унаследованный от Number».

`extends B` – символ подстановки с указанием верхней границы;

`super B` – символ подстановки с указанием нижней границы;

где `B` – представляет собой границу.



Если необходимо читать из контейнера, то используйте *wildcard с верхней границей* `<? extends>`. Если необходимо писать

в контейнер, то используйте *wildcard с нижней границей* `<? super>`. Не используйте *wildcard*, если нужно производить и запись, и чтение.

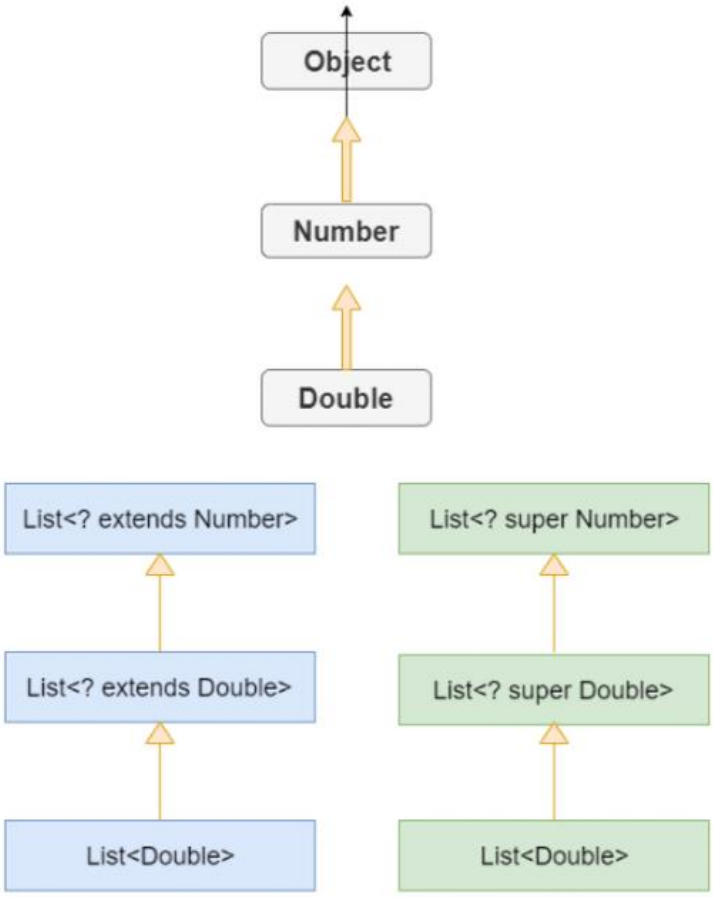
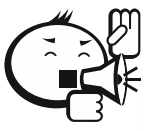


Рис. 3.13 – Иерархия классов

Иногда возникает ситуация, когда мы хотим, чтобы наш общий метод работал со всеми типами, в этом случае можно использовать неограниченный *подстановочный знак*. Это то же самое, что использовать `<? extends Object>`. Чтобы в метод можно было передать любой список `List` с любым параметром, существуют две записи:

```
List<? extends Object>
List<?>
```

Они эквивалентны по смыслу, поэтому обычно используют вторую.



Существует универсальный тип, в котором все аргументы типа являются неограниченными подстановочными знаками `<?>` без каких-либо ограничений на переменные типа. Например

```
ArrayList<?>  list = new ArrayList<Long>();
//or
ArrayList<?>  list = new ArrayList<String>();
//or
ArrayList<?>  list = new ArrayList<Employee>();
.....................................................
```

Давайте посмотрим пример. Добавим к классам `Students` и `Teacher` родительский класс (`Person`). Модифицируем код.

```
//родительский класс - super
 class Person {
    protected String type;
    protected Integer id;

    public Person(int id, String type) {
        this.id = id;
        this.type= type;
    }

    public String getType() {
        return this.type;
    }

    @Override
    public String toString() {
        return "человек "+id+",  "+type;
         }
     }
//класс наследник
 class Students extends  Person {

      public Students(int id) {
             super(id, "Студент");
         }

    @Override
    public String toString() {
        return "студент "+id+",  "+type;
    }

}
//класс наследник
 class Teacher extends  Person{
      public Teacher(int id) {
```

```
            super(id, "Преподаватель");
        }

        @Override
        public String toString() {
            return "преподаватель "+id+",  "+type;
        }
}
```

Теперь давайте создадим три списка, используя эти классы, как показано ниже.

```
public class Main {

        public static void main(String[] args) {
            //создадим списки тестовых объектов
            //1
             List<Person> pList = new ArrayList<>();
              pList.add(new Person(1, "Студент"));
              pList.add(new Students(4));
              System.out.println(pList);

              //2
              List<Students> cList = new ArrayList<>();
              cList.add(new Students(2));
              System.out.println(cList);

              //3
              List<Teacher> dList = new ArrayList<>();
              dList.add(new Teacher(3));
              System.out.println(dList);

        }
    }
```

После выполнения программы получим следующий результат (рис. 3.14)

```
Console  Console
<terminated> Main (3) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe
[человек 1,  Студент, студент 4,  Студент]
[студент 2,  Студент]
[преподаватель 3,  Преподаватель]
```

Рис. 3.14 – Пример использования обобщения при наследовании

Мы можем добавить только объект `Person` и объекты его подклассов `Students` и `Teacher` в `pList`. Но если мы сделаем следующее:

```
pList = cList;
```

это вызовет ошибку времени компиляции, несоответствие типов, т. к. невозможно преобразовать из списка в список (рис. 3.15).

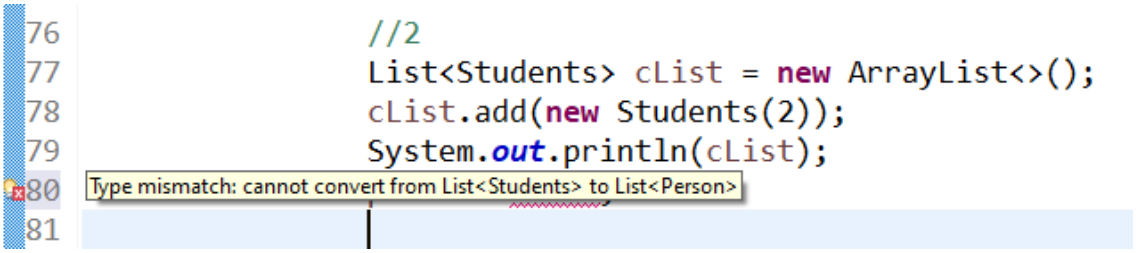


Рис. 3.15 – Ошибка несоответствия типов

Если все же нам необходимо записать список `cList` в список `pList`, т. е. вставить экземпляры `Person` и `Teacher` в список `CList`, можно сделать это через ссылку `pList`, которая объявлена как `List`. Таким образом, мы сможем вставить объекты, отличные от `Students`, в список, объявленный для хранения экземпляров `Students` (или подкласса `Students`).

Теперь предположим, что мы определили метод для печати типа `Person` из коллекции:

```
public static void printType(List<Person> persons) {
        System.out.println("printing type");
      for(Person p : persons) {
          System.out.println(p.getType());
     }
       }
```

Метод `printType()` не будет работать, если мы передадим ему список `Students` или список `Teacher`.

Решить эту проблему поможет подстановочный символ «?».

```
public static void printType(List<? extends Person> persons) {
     System.out.println("printing type");
      for(Person p : persons) {
          System.out.println(p.getType());
     }
       }
```

Теперь вызовем этот метод для всех списков:

```
Person.printType(pList);
Person.printType(cList);
Person.printType(dList);
```

Получим результат выполнения программы (рис. 3.16).

```
printing type
Студент
Студент
printing type
Студент
printing type
Преподаватель
```

Рис. 3.16 – Пример использования `wildcard`

Использовать `<? extends wildcard>` можно, если надо получить объект из коллекции, т. к. доступ только для чтения. Мы не можем добавлять элементы в коллекцию.

Чтобы добавить объекты в коллекцию, используем `<? super wildcard>`, он позволит добавить объект Person или его подклассы. Чтобы добавить элементы в эту коллекцию, мы можем создать метод `addPerson()`, как показано ниже.

```
public static void addPerson(Person p, List<? super Person> person) {
      person.add(p);
      }
```

Добавим нового студента в список:

```
//выведем список dList
                  Person.printType(dList);
                  //добавим еще один элемент в список
                  dList.add(new Teacher(8));
                  //выведем список dList
                        Person.printType(dList);
```

Как видим, обозначение типа дает нам мало информации (рис. 3.17).

```
printing type
Преподаватель
printing type
Преподаватель
Преподаватель
```

Рис. 3.17 – Пример использования *wildcard с нижней границей*

Добавим еще номер:

```
public static void printType(List<? extends Person> persons) {

    System.out.println("printing type");
     for(Person p : persons) {
         System.out.println(p.toString());
    }
        }
```

Теперь видно, что мы добавили преподавателя с номером 8 (рис. 3.18).

```
printing type
преподаватель 3,  Преподаватель
printing type
преподаватель 3,  Преподаватель
преподаватель 8,  Преподаватель
```

Рис. 3.18 – Пример использования *wildcard с верхней границей*

Выводы

Преимущества дженериков в Java:

1. Возможность повторного использования кода.
2. Приведение отдельных типов не требуется.
3. Одним из преимуществ использования дженериков является избежание затрат и обеспечение безопасности типов. Это особенно полезно при работе с коллекциями.
4. Generics предоставляет способ сообщить тип коллекции для компилятора, чтобы можно было проверить, что ограничения типа не нарушаются во время выполнения.

Контрольные вопросы по главе 3

1. Почему мы используем Generics в Java?
2. Как Generics работают в Java?
3. Для чего необходимо использовать стирание типов?
4. В чем различие между `List<? extends T>` и `List <? super T>`?
5. В чем различие между `List<?>` и `List<Object>` в Java?

4 Потоки ввода-вывода и потоки выполнения. Многопоточное программирование

4.1 Потоки

Ввод-вывод относится к различным способам получения и передачи информации: к чтению и записи дискового файла, символов с клавиатуры либо получению данных из сети. Эти абстракции дают удобную возможность для работы с вводом-выводом (I/O), не требуя при этом, чтобы каждая часть вашего кода понимала разницу между, скажем, клавиатурой и сетью. В Java эта абстракция называется потоком (*stream*) и реализована в нескольких классах пакета *java.io*.

Поток данных (*stream*) *представляет собой абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.*

Поток прячет детали того, что случается с данными внутри реального устройства ввода-вывода. При этом потоки делятся на *байтовые* и *символьные*. Единицей обмена для байтовых потоков является байт, для символьных – символ Unicode.

На вершине иерархии байтовых потоков находятся два абстрактных класса: `InputStream` и `OutputStream` (рис. 4.1). В этих классах определены методы `read()` и `write()`, предназначенные для чтения данных из потока и записи данных в поток соответственно.

Иерархия классов для символьных потоков ввода-вывода начинается с абстрактных классов `Reader` и `Writer` (рис. 4.2). В этих классах определены методы `read()` для считывания символьных данных из потока и `write()` для записи символьных данных в поток.

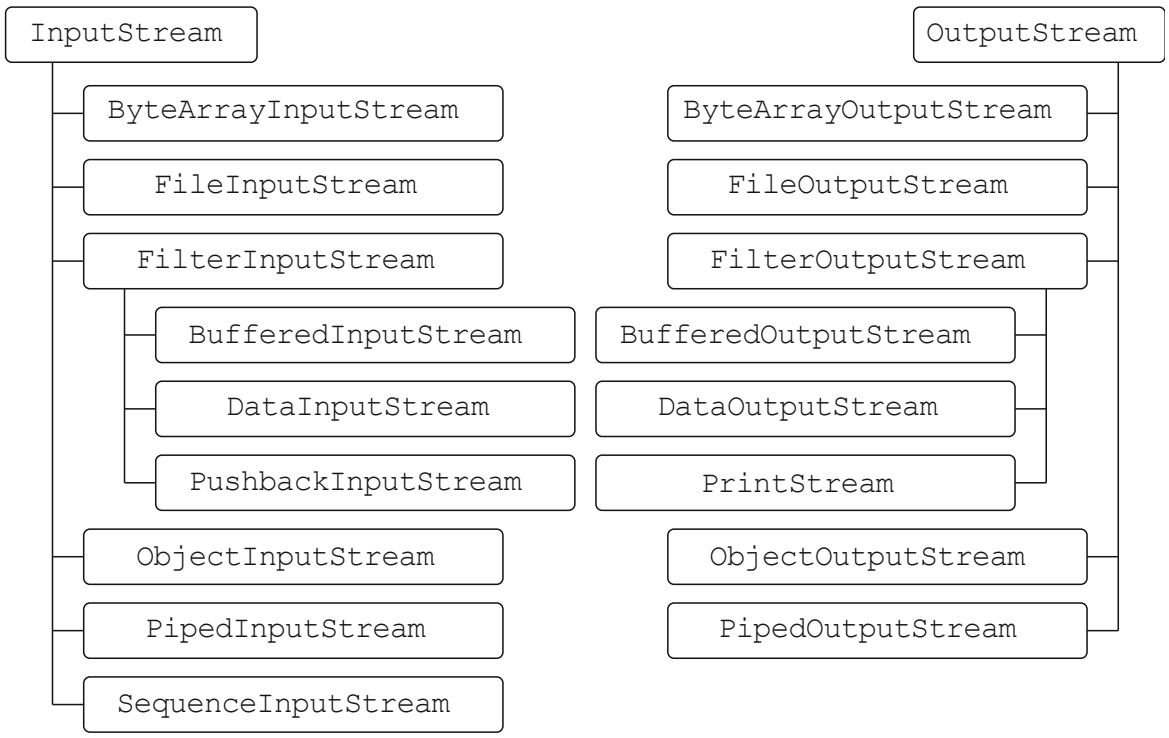


Рис. 4.1 – Иерархия байтовых потоков

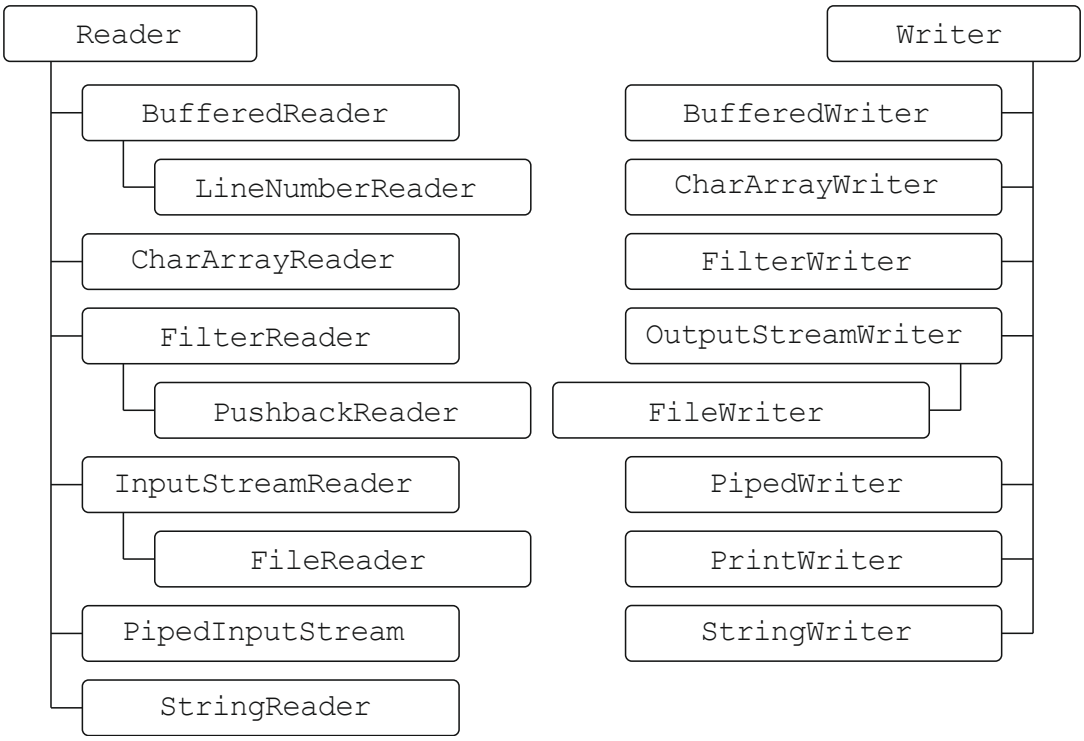


Рис. 4.2 – Иерархия символьных потоков

Бывают ситуации, когда требуется совместно использовать байтовые и символьные потоки данных, например, если данные должны поступить из источника в виде набора битов, быть каким-либо образом обработаны и переданы в другое место также в виде битов, при этом известно, что данные являются тек-

стовой информацией. В таких ситуациях более удобно на этапе обработки данных перейти от байтовых потоков к символьным, так как последние предоставляют более приспособленный к обработке текста инструментарий. Для преобразования между байтовыми и символьными потоками используются *классы-«мосты»*: `InputStreamReader` – для преобразования от байтового потока к символьному для чтения данных – и `OutputStreamWrite` – от символьного потока к байтовому для записи данных.

InputStream

Работа `InputStream` состоит в представлении классов, которые производят ввод от различных источников. Каждый из них имеет ассоциированный подкласс `InputStream`.

Все методы этого класса при возникновении ошибки возбуждают исключение `IOException`. Ниже приведен краткий обзор методов класса `InputStream`:

- `read()` возвращает представление очередного доступного символа во входном потоке в виде целого;
- `read(byte b[])` пытается прочесть максимум `b.length` байтов из входного потока в массив `b`. Возвращает количество байтов, в действительности прочитанных из потока;
- `read(byte b[], int off, int len)` пытается прочесть максимум `len` байтов, расположив их в массиве `b`, начиная с элемента `off`. Возвращает количество реально прочитанных байтов;
- `skip(long n)` пытается пропустить во входном потоке `n` байтов. Возвращает количество пропущенных байтов;
- `available()` возвращает количество байтов, доступных для чтения в настоящий момент;
- `close()` закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению `IOException`;
- `mark(int readlimit)` ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано `readlimit` байтов;
- `reset()` возвращает указатель потока на установленную ранее метку;
- `markSupported()` возвращает `true`, если данный поток поддерживает операции `mark/reset`.

OutputStream

Эта категория включает классы, которые решают, куда будет производиться вывод: в массив байтов, в файл. Все методы этого класса имеют тип `void` и возбуждают исключение `IOException` в случае ошибки. Ниже приведен список методов этого класса:

- `write(int b)` записывает один байт в выходной поток. Обратите внимание: аргумент этого метода имеет тип `int`, что позволяет вызывать `write`, передавая ему выражение, при этом не нужно выполнять приведение его типа к `byte`;
- `write(byte b[])` записывает в выходной поток весь указанный массив байтов;
- `write(byte b[], int off, int len)` записывает в поток часть массива – `len` байтов, начиная с элемента `b[off];`
- `flush()` очищает любые выходные буферы, завершая операцию вывода;
- `close()` закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать `IOException`.

Символьные потоки

Для работы с символьными потоками в Java существуют два базовых класса – `Reader` и `Writer`.

Абстрактный класс `Reader` предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

- `absract void close()`: закрывает поток ввода;
- `int read()`: возвращает целочисленное представление следующего символа в потоке. Если таких символов нет и достигнут конец файла, то возвращается число –1;
- `int read(char[] buffer)`: считывает в массив `buffer` из потока символы, количество которых равно длине массива `buffer`. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;
- `int read(CharBuffer buffer)`: считывает в объект `CharBuffer` из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;

- `absract int read(char[] buffer, int offset, int count)`: считывает в массив `buffer`, начиная со смещения `offset`, из потока символы, количество которых равно `count`;
- `long skip(long count)`: пропускает количество символов, равное `count`. Возвращает число успешно пропущенных символов.

Класс `Writer` определяет функционал для всех символьных потоков вывода. Его основные методы:

- `Writer append(char c)`: добавляет в конец выходного потока символ `c`. Возвращает объект `Writer`;
- `Writer append(CharSequence chars)`: добавляет в конец выходного потока набор символов `chars`. Возвращает объект `Writer`;
- `abstract void close()`: закрывает поток;
- `abstract void flush()`: очищает буферы потока;
- `void write(int c)`: записывает в поток один символ, который имеет целочисленное представление;
- `void write(char[] buffer)`: записывает в поток массив символов;
- `absract void write(char[] buffer, int off, int len)`: записывает в поток только несколько символов из массива `buffer`. Причем количество символов равно `len`, а отбор символов из массива начинается с индекса `off`;
- `void write(String str)`: записывает в поток строку;
- `void write(String str, int off, int len)`: записывает в поток из строки некоторое количество символов, которое равно `len`, причем отбор символов из строки начинается с индекса `off`.

Специализированные потоки

В пакет `java.io` входят потоки для работы с основными типами источников и приемников данных (табл. 4.1).

Таблица 4.1 – Специализированные потоки

Тип данных	Байтовые потоки		Символьные потоки	
	Входной	**Выходной**	**Входной**	**Выходной**
Файл	`FileInput-Stream`	`FileOutput-Stream`	`FileReader`	`FileWriter`

Массив	ByteArray-InputStream	ByteArray-OutputStream	CharArray-Reader	CharArray-Writer
Строка	_	_	String-Reader	String-Writer
Конвейер	PipedInput-Stream	PipedOutput-Stream	Piped-Reader	Piped-Writer

Конструкторы этих потоков в качестве аргумента принимают ссылку на источник или приемник данных – файл, массив, строку.

Файловые потоки

Java обеспечивает ряд классов и методов, которые позволяют читать и записывать файлы. Для Java все файлы имеют байтовую структуру, а Java обеспечивает методы для чтения и записи байтов в файл. Кроме того, Java позволяет упаковывать байтовый файловый поток в символьно-ориентированный объект.

Для создания байтовых потоков, связанных с файлами, чаще всего используются два поточных класса – `FilelnputStream` и `FileOutput-Stream`. Для открытия файла создается объект одного из этих классов с указанием имени файла как аргумента конструктора.

Стандартные потоки ввода-вывода

Термин *стандартный ввод-вывод* относится к концепции Unix (которая в некоторой форме была воспроизведена в Windows и многих других операционных системах) единого потока информации, который используется программой. Весь ввод программы может вестись через *стандартный ввод*, весь вывод может идти в *стандартный вывод*, а все сообщения об ошибках могут посылаться в *стандартный поток ошибок*. Значение стандартного ввода-вывода в том, что программы вместе могут представлять цепочку и стандартный вывод одной программы может стать стандартным вводом для другой.

Чтение из стандартного ввода

Часть возможностей ввода-вывода может быть реализована посредством класса `System`.

Класс `java.lang.System` является `final`, все поля и методы являются статическими (`static`), поэтому нельзя создать подкласс и переопределить его методы, используя наследование. Класс

Java System не предоставляет каких-либо публичных конструкторов, поэтому мы не можем создать экземпляр этого класса.

Класс System содержит три переменных потока: in, out и err. Эти поля имеют атрибуты public и static:

- поле System.out – поток стандартного вывода. По умолчанию он связан с консолью. Поле System.out является объектом класса PrintStream;
- поле System.in – это поток стандартного ввода. По умолчанию он связан с клавиатурой. Поле является объектом класса InputStream;
- поле System.err – это стандартный поток ошибок. По умолчанию поток связан с консолью. Поле является объектом класса Print-Stream.

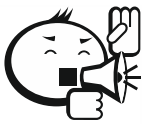
Для вывода на консоль ранее использовался метод println() класса PrintStream без определения экземпляров этого класса. Можно использовать статическое поле out класса System, которое является объектом класса PrintStream. Исполняющая система Java связывает это поле с консолью.

Можно писать System.out.println(), определяя новую ссылку на System.out, например:

```
PrintStream pr = System.out;
```

и писать просто

```
pr.println();
```



Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin 1 с кодами '\u0000'–'\u00FF' при этом просто откидывается нулевой старший байт и выводятся байты '0x00'–'0xFF'. Для кодов кириллицы, которые лежат в диапазоне '\u0400'–'\u04FF' кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали.

Пожалуй, наиболее широко используемой возможностью, предоставляемой System, является стандартный вывод, доступный через переменную

System.out. Ее тип – PrintStream. Стандартный вывод можно перенаправить в другой поток (файл, массив байт и т. д., главное, чтобы это был объект PrintStream).

Пример

```
System.out.println("Наши студенты сохранены в файл");
            try {
            PrintStream print = new PrintStream(new FileOut-
putStream("c:\\file.txt"));
            System.setOut(print);
         System.out.println("Список студентов сохранен.");
            for(MyClass mc : list) {
            System.out.println("Item:" + mc);
            }
            } catch(FileNotFoundException e) {
            e.printStackTrace();
        }
```

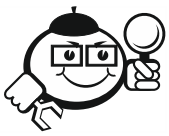
Ввод с консоли

Для ввода данных используется класс Scanner из библиотеки пакетов Java.

Этот класс надо импортировать в той программе, где он будет использоваться. Это делается до начала открытого класса в коде программы.

В классе есть методы для чтения очередного символа заданного типа со стандартного потока ввода, а также для проверки существования такого символа.

Для работы с потоком ввода необходимо создать объект класса Scanner, при создании указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом System.in. А стандартный поток вывода (дисплей) – уже знакомым объектом System.out. Есть еще стандартный поток для вывода ошибок – System.err, но работа с ним выходит за рамки нашего курса.



Пример

```
        import java.util.Scanner; // импортируем класс Scanner
        public class Main {
        public static void main(String[] args) {
        Scanner in = new Scanner(System.in); /* создаем объект класса
Scanner */
        int i;
       System.out.print("Введите возраст студента: ");
         if(in.hasNextInt()) { /* возвращает истину, если с потока ввода
можно считать целое число */
         i = in.nextInt(); /* считывает целое число с потока ввода и со-
храняет в переменную */
         System.out.println(i);
         } else {
       System.out.println("Вы ввели не целое число");
         }
    }
 }
```

Метод `hasNextDouble()`, примененный к объекту класса `Scanner`, проверяет, можно ли считать с потока ввода вещественное число типа `double`, а метод `nextDouble()` считывает его. Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит).

Метод `nextLine()` позволяет считывать целую последовательность символов, т. е. строку, а значит, полученное через этот метод значение нужно сохранять в объекте класса `String`. В следующем примере создается два таких объекта, потом в них поочередно записывается ввод пользователя, а далее на экран выводится одна строка, полученная объединением введенных последовательностей символов.

Пример

```
  import java.util.Scanner;
        public class Main {
        public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        String s;
        System.out.print("Введите имя студента: ");
```

```
        s = in.nextLine();
        System.out.println(s);
        }
        }
```

Существует и метод `hasNext()`, проверяющий, остались ли в потоке ввода какие-то символы.

Как видно, связь с консолью средствами классов-потоков весьма сложна. Начиная с Java SE 6 в пакет `java.io` добавлен класс `Console`, облегчающий эту задачу.

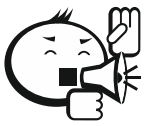
Поскольку программа связывается с той консолью, в которой запущена виртуальная машина Java, единственный объект класса `Console` создается статическим методом `console()` класса `System`, например:

```
Console cons = System.console();
```

Метод возвращает `null`, если виртуальная машина Java запущена не из консоли, а каким-нибудь приложением.

Форматированный вывод

На технологию Java традиционно переходит очень много программистов, прежде писавших программы на языке C. Им очень не хватает функции `printf()`, позволяющей самому программисту как-то оформить вывод информации: задать количество цифр при выводе вещественных чисел, точно указать количество пробелов между данными.



Начиная с JDK 1.5 методы `printf()`, очень похожие на одноименные функции языка C, появились в классах `PrintStream` и `PrintWriter`. Кроме них в эти классы введены методы `format()`, выполняющие те же действия. Последние методы заимствованы из класса `Formatter`, находящегося в пакете java.util и специально предназначенного для форматирования.

Заголовки методов форматированного ввода-вывода класса `PrintStream` выглядят так:

- `PrintStream format(Local l, String format, Object ... args);`
- `PrintStream format(String format, Object ... args);`

- `PrintStream printf(Local l, String format, Object ... args);`
- `PrintStream printf(String format, Object ... args).`

В классе `PrintWriter` такие же методы возвращают ссылку на свой экземпляр класса `PrintWriter`.

Строка символов `format` описывает шаблон для вывода данных, перечисленных в следующих аргументах метода, а также содержит надписи, которые должны появиться на консоли. Например, тот же самый вывод на консоль, который мы до сих пор делали методом

```
System.out.println("x = " + x + ", y = " + y);
```

можно сделать методом

```
System.out.printf("x = %d, y = %d\n", x, y);
```

В строке формата мы пишем поясняющий текст `"x = , y = \n"`, который будет просто выводиться на консоль. В текст вставляем *спецификации формата* `"%d"`. На место этих спецификаций во время вывода будут подставлены значения данных, перечисленных в следующих аргументах метода.

Класс File

В отличие от большинства классов ввода-вывода класс `File` работает не с потоками, а непосредственно с файлами. Данный класс позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

Для создания объектов класса `File` можно использовать один из следующих конструкторов:

- `File(File dir, String name)` – указывается объекта класса `File` (каталог) и имя файла;
- `File(String path)` – указывается путь к файлу без указания имени файла;
- `File(String dirPath, Sring name)` – указывается путь к файлу и имя файла;
- `File(URI uri)` – указывается объекта URI, описывающий файл.

Методы класса File

Класс `File` может использоваться для создания каталога или дерева каталогов. Также можно узнать свойства файлов (размер, дату последнего измене-

ния, режим чтения/записи), определить, к какому типу (файл или каталог) относится объект `File`, удалить файл. У класса очень много методов, перечислим некоторые:

- `getAbsolutePath()` – абсолютный путь файла начиная с корня системы. В Android корневым элементом является символ «слеш» (/);
- `canRead()` – доступно для чтения;
- `canWrite()` – доступно для записи;
- `exists()` – файл существует или нет;
- `getName()` – возвращает имя файла;
- `getParent()` – возвращает имя родительского каталога;
- `getPath()` – путь;
- `lastModified()` – дата последнего изменения;
- `isFile()` – объект является файлом, а не каталогом;
- `isDirectory` – объект является каталогом;
- `isAbsolute()` – возвращает `true`, если файл имеет абсолютный путь;
- `renameTo(File newPath)` – переименовывает файл. В параметре указывается имя нового имени файла. Если переименование прошло неудачно, то возвращается `false`;
- `delete()` – удаляет файл. Также можно удалить пустой каталог;
- `length()` – получить длину в байтах;
- `setReadable()`, `setWritable()`, `setExecutable()` – позволяют установить их для всех пользователей или только для владельца файла или каталога.

Пример

```
import java.io.File;
          import java.io.IOException;

public class File_App {

   public static void main(String[] args) {
            // определяем объект для каталога
     File myFile = new File("C://work//oop.txt");
     System.out.println("Имя файла: " + myFile.getName());
     System.out.println("Родительский каталог: " + myFile.getParent());
            if(myFile.exists())
```

```
            System.out.println("Файл существует");
            else
            System.out.println("Файл еще не создан");

            System.out.println("Размер файла: " + myFile.length());
            if(myFile.canRead())
        System.out.println("Файл доступен для чтения");
            else
        System.out.println("Файл не доступен для чтения");

            if(myFile.canWrite())
        System.out.println("Файл доступен для записи");
            else
        System.out.println("Файл не доступен для записи");

            // создадим новый файл
            File newFile = new File("C://work//MyFile");
            try {
            boolean created = newFile.createNewFile();
            if(created)
            System.out.println("Файл создан");
            }
            catch(IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

4.2 Сериализация и десериализация объектов

Кроме данных базовых типов в поток можно отправлять объекты произвольных классов.

Сериализация (*serializable*) *– это процесс, когда состояние объекта и его метаданные* (*например, имя класса объекта и имена его атрибутов*) *сохраняются в последовательность байтов, по которой затем его можно полностью восстановить.*

Проще говоря, сериализация – процесс преобразования объектов в потоки байтов для хранения. Преобразование объекта в этот формат – сериализация –

позволяет сохранить всю информацию, необходимую для последующего восстановления (десериализации) объекта в нужное время (рис. 4.3).

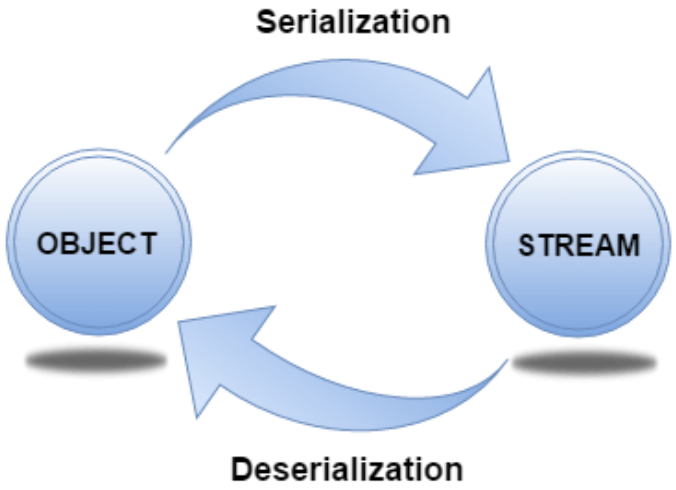
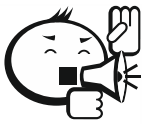


Рис. 4.3 – Процессы сериализации и десериализации

Процесс извлечения объекта из потока байтов называется ***десериализацией*** (*deserialization*).

Существует две основных причины сериализации объекта: сохранение и передача. Сохранение объекта – это запись состояния объекта в постоянном устройстве хранения, например, в базе данных. Передача объекта означает передачу объекта в другой компьютер или систему.

Для того чтобы объекты класса могли выполнять процесс сериализации, этот класс должен расширять интерфейс `Serializable`.



Интерфейс `Serializable` является «маркерным» интерфейсом. Это означает, что у него нет методов или полей, он просто «маркирует» класс как возможность сериализации. Когда виртуальная машина Java (JVM) встречает класс, который «помечен» как `Serializable` во время процесса сериализации, виртуальная машина предположит, что безопасно писать в поток.

Все подклассы класса, реализующего интерфейс `Serializable`, также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том

случае, если это поле не имеет спецификатора `static` или `transient`, поскольку такие поля не могут быть предметом сериализации, но существует различие в десериализации. Так, поле с модификатором `transient` после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением `null`), а поле с модификатором `static` получает значение по умолчанию в случае отсутствия в области видимости объектов данного типа, а при их наличии получает значение, которое определено для существующего объекта.

Класс `String` и все классы-оболочки по умолчанию реализуют интерфейс `java.io.Serializable`.

4.2.1 Сериализация

Первым шагом к выполнению работы по сериализации является предоставление объектам возможности использовать этот механизм.

Чтобы сериализовать объект, необходимо, чтобы класс объекта реализовал интерфейс `Serializable`. Затем следует выполнить следующие действия:

1) создать поток `ObjectOutputStream (o)`, который записывает объект в переданный `OutputStream`;
2) записать в поток: `o.writeObject(Object)`;
3) сделать `o.flush()` и `o.close()`.

Для сериализации объектов в поток используется класс `ObjectOutputStream`. Он записывает данные в поток. Для создания объекта `ObjectOutputStream` в конструктор передается поток, в который производится запись:

```
ObjectOutputStream(OutputStream out);
```

Для записи данных `ObjectOutputStream` использует ряд методов, среди которых можно выделить следующие:

- `void close()`: закрывает поток;
- `void flush()`: очищает буфер и сбрасывает его содержимое в выходной поток;
- `void write(byte[] buf)`: записывает в поток массив байтов;
- `void write(int val)`: записывает в поток один младший байт из `val`;

- `void writeBoolean(boolean val)`: записывает в поток значение `boolean`;
- `void writeByte(int val)`: записывает в поток один младший байт из `val`;
- `void writeChar(int val)`: записывает в поток значение типа `char`, представленное целочисленным значением;
- `void writeDouble(double val)`: записывает в поток значение типа `double`;
- `void writeFloat(float val)`: записывает в поток значение типа `float`;
- `void writeInt(int val)`: записывает целочисленное значение `int`;
- `void writeLong(long val)`: записывает значение типа `long`;
- `void writeShort(int val)`: записывает значение типа `short`;
- `void writeUTF(String str)`: записывает в поток строку в кодировке UTF-8;
- `void writeObject(Object obj)`: записывает в поток отдельный объект.

4.2.2 Десериализация

Десериализация – это процесс восстановления объекта из сериализованного состояния, операция, обратная сериализации.

Чтобы десериализовать объект, необходимо, чтобы класс объекта реализовал интерфейс `Serializable`. Затем следует выполнить следующие действия:

1) создать поток `ObjectInputStream (o)`, в который передать `InputStream (instream)` (из файла, из памяти...);
2) получить объект и привести к нужному типу: `MyObject my = (MyObject) o.readObject(instream).`

Класс `ObjectInputStream` отвечает за обратный процесс – чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода:

```
ObjectInputStream(InputStream in);
```

Функционал `ObjectInputStream` сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

- `void close()`: закрывает поток;
- `int skipBytes(int len)`: пропускает при чтении несколько байтов, количество которых равно `len`;
- `int available()`: возвращает количество байтов, доступных для чтения;
- `int read()`: считывает из потока один байт и возвращает его целочисленное представление;
- `boolean readBoolean()`: считывает из потока одно значение `boolean`;
- `byte readByte()`: считывает из потока один байт;
- `char readChar()`: считывает из потока один символ `char`;
- `double readDouble()`: считывает значение типа `double`;
- `float readFloat()`: считывает из потока значение типа `float`;
- `int readInt()`: считывает целочисленное значение `int`;
- `long readLong()`: считывает значение типа `long`;
- `short readShort()`: считывает значение типа `short`;
- `String readUTF()`: считывает строку в кодировке UTF-8;
- `Object readObject()`: считывает из потока объект.

Эти методы охватывают весь спектр данных, которые можно сериализовать.

При чтении объектов `ObjectInputStream` напрямую пытается сопоставить все атрибуты с классом, в который мы пытаемся привести читаемый объект.

Давайте посмотрим на пример, приведенный ниже:

```
import java.io.Serializable;
//сериализуем класс
 class Students implements Serializable{
 int age;
 String name;

 //конструкторы
 Students() {
 };

 public Students(String name,int age) {
```

```
  this.age= age;
  this.name = name;
 }

 @Override
 public String toString() {
     return "ФИО:" + name + "\nВозраст: " + age;
 }
}
```

В приведенном выше примере класс `Students` реализует интерфейс `Serializable`. Теперь его объекты можно преобразовать в поток. Если он не может точно отобразить соответствующий объект, он генерирует исключение *ClassNotFound*.

Давайте теперь рассмотрим сам процесс. Сохраним в файл объекты класса `Students` и считаем их обратно на консоль:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Main {

          public static void main(String[] args) {
               //создание объектов p1 и p2
               Students p1 = new Students("Иванов", 18);
               Students p2 = new Students("Петров", 22);

             try {
               //запись объектов в файл с помощью ObjectOutputStream
                FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                ObjectOutputStream o = new ObjectOutputStream(f);

               //Запись объектов в файл myObjects.txt
                o.writeObject(p1);
                o.writeObject(p2);

                o.close();
                f.close();
```

```
                //чтение объектов с помощью ObjectInputStream
                FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
                ObjectInputStream oi = new ObjectInputStream(fi);

               //Чтение объектов из файла myObjects.txt
                Students pr1 = (Students) oi.readObject();
                Students pr2 = (Students) oi.readObject();

                System.out.println(pr1.toString());
                System.out.println(pr2.toString());

                oi.close();
                fi.close();

            } catch (FileNotFoundException e) {
                System.out.println("File not found");
            } catch (IOException e) {
                System.out.println("Error initializing stream");
            } catch (ClassNotFoundException e) {

                e.printStackTrace();
            }

        }

            }
```

В результате работы программы, которая сериализирует объекты, мы получим примерно следующий результат на консоли и в файле *myObjects.txt* (рис. 4.4).

Посмотреть содержимое файла с сериализационными объектами простыми текстовыми редакторами не получится, т. к. там хранятся бинарные данные.

Как видно из результатов работы программы, мы получили из файла ровно те же объекты, что и сохранили.

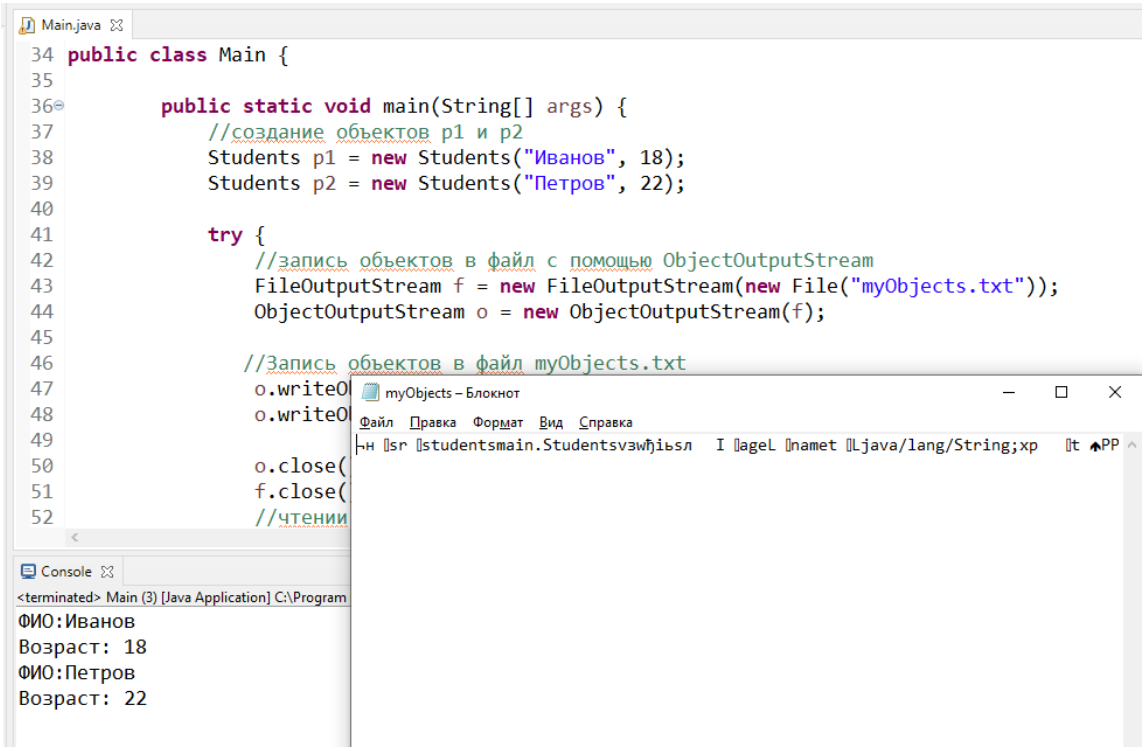
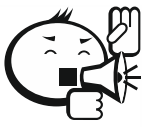


Рис. 4.4 – Результат сериализации и десериализации объектов

4.2.3 Исключение данных из сериализации

По умолчанию сериализуются все переменные объекта. Однако, возможно, мы хотим, чтобы некоторые поля были исключены из сериализации. Для этого они должны быть объявлены с модификатором `transient`.



`transient` («временный, переходный») – это модификатор для поля класса. При восстановлении объекта у него будет это поле, но со значением `null (0, false)`.

Например, исключим поле `age` из нашего примера:

```
import java.io.Serializable;
//сериализуем класс
 class Students implements Serializable{
transient int age;
 String name;

 //конструкторы
 Students() {
 };

 public Students(String name,int age) {
```

```
 this.age= age;
 this.name = name;
}

@Override
public String toString() {
    return "ФИО:" + name + "\nВозраст: " + age;
}
    }
```

В результате, если мы десериализуем объекты, мы получили значения для поля `age` по умолчанию для `transient`-переменной, поскольку значение возраста не было сериализовано (рис. 4.5). Оно всегда будет возвращать значение по умолчанию. В таком случае он вернет 0, потому что тип данных `age` является целым числом.

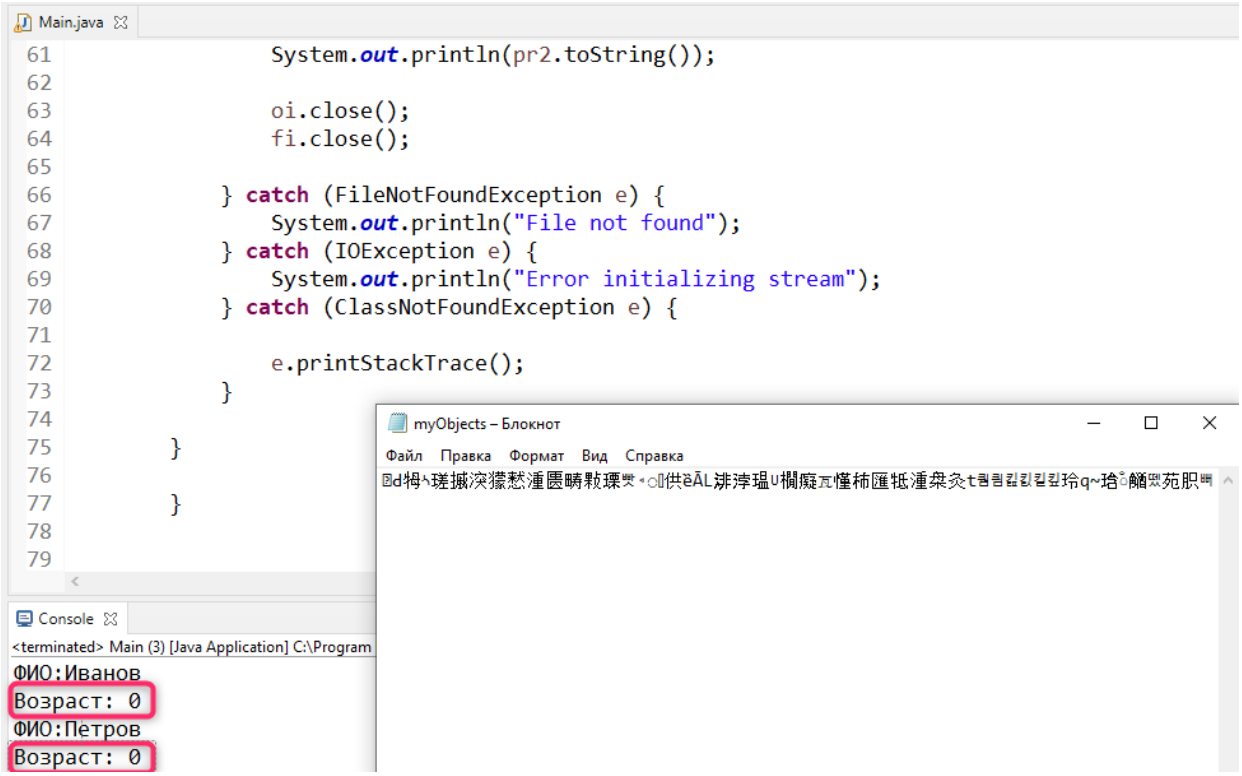


Рис. 4.5 – Десериализация поля `transient`-переменной

4.2.4 Сериализация статических полей

Если в классе есть какой-либо статический член данных, он не будет сериализован, потому что статический является частью класса, а не объекта.

Укажем в нашем примере поле `faculty` как статическое.

```
import java.io.Serializable;
//сериализуем класс
```

```
 class Students implements Serializable{
int age;
String name;
static String faculty;

 //конструкторы
Students() {
};

public Students(String nameOfFaculty, String name,int age) {
faculty = nameOfFaculty;
 this.age= age;
 this.name = name;
}


@Override
public String toString() {
    return "\nФакультет " + faculty +"\nФИО:" + name + "\nВозраст: " +
age;
}

}
```

Далее:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class Main {

          public static void main(String[] args)  {
               //создание объекта p
          Students p = new Students("ФСУ","Иванов", 18);

             try {
                //запись объектов в файл с помощью ObjectOut-putStream
                 FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                 ObjectOutputStream o = new ObjectOutputStream(f);
                 //Запись объектов в файл myObjects.txt
                 o.writeObject(p);

                 o.close();
```

```
                f.close();
                //Чтение объектов с помощью ObjectInputStream
                FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
                ObjectInputStream oi = new ObjectInputStream(fi);


      Students.faculty = "ФДО";//изменение значения static-поля
               //Чтение объектов из файла myObjects.txt
                Students pr = (Students) oi.readObject();

                System.out.println(pr.toString());

                oi.close();
                fi.close();

            } catch (FileNotFoundException e) {
                System.out.println("File not found");
            } catch (IOException e) {
                System.out.println("Error initializing stream");
            } catch (ClassNotFoundException e) {

                e.printStackTrace();
            }

        }

            }
```

Вывод этой программы будет следующий (рис. 4.6).

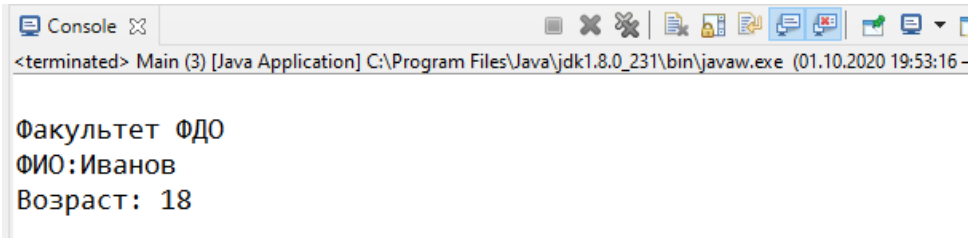
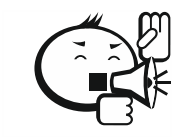


Рис. 4.6 – Сериализация статического поля

Поле `faculty`, помеченное как статическое, получает то значение, которое имеет это поле на текущий момент, т. е. при создании объекта `p` поле получило значение «ФСУ», а затем значение статического поля было изменено на «ФДО». Если же объекта данного типа нет в области видимости, то статическое поле также получает значение по умолчанию. Поэтому после десериализации на консоли мы увидели значение «ФДО».



Во время десериализации статическое значение переменной будет загружено из класса, т. е. получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта. Но любая статическая переменная, которая предоставляется при инициализации класса, сериализуется.

Если нужно сериализовать статические поля, мы должны делать это вручную. Добавим два метода для сериализации/десериализации статического поля `faculty`:

```
public static void serializeStatic(ObjectOutputStream oos) throws IOException{
     oos.writeUTF(faculty);
 }
public static void deserializeStatic(ObjectInputStream ois) throws IOException{
       faculty=ois.readUTF();
}
```

Далее вызовем эти методы:

```
public class Main {

           public static void main(String[] args)  {
                //создание объекта p
                Students p = new Students("ФСУ","Иванов", 18);

              try {
                //запись объектов в файл с помощью ObjectOut-putStream
                 FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                 ObjectOutputStream o = new ObjectOutputStream(f);

                //Запись объектов в файл myObjects.txt
                 Students.serializeStatic(o);
                 o.writeObject(p);
                 o.close();
                 f.close();

                 //Чтение объектов с помощью ObjectInputStream
                 FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
       ObjectInputStream oi = new ObjectInputStream(fi);
       Students.faculty = "ФДО";//изменение значения static-поля
```

```
            //Чтение объектов из файла myObjects.txt
             Students.deserializeStatic(oi);
             Students pr = (Students) oi.readObject();
             System.out.println(pr.toString());

             oi.close();
             fi.close();

         } catch (FileNotFoundException e) {
             System.out.println("File not found");
         } catch (IOException e) {
             System.out.println("Error initializing stream");
         } catch (ClassNotFoundException e) {

             e.printStackTrace();
         }

     }

}
```

Вывод этой программы будет следующий (рис. 4.7).

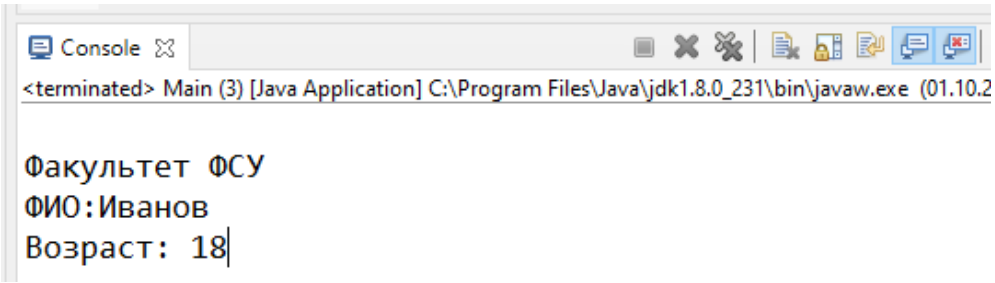


Рис. 4.7 – Сериализация статического поля

Как видите, теперь все в порядке. Мы просто добавили и вызвали два статических метода: для сериализации и десериализации статической переменной. Также можно управлять сериализацией в ручном режиме, где это требуется.

4.2.5 Сериализация с массивом или коллекцией

В случае массива или коллекции все объекты массива или коллекции должны быть сериализуемыми. Если какой-либо объект не сериализуем, сериализация не удастся.

Создадим список наших студентов:

```
import java.io.File;
import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
public class Main {

           public static void main(String[] args)  {
                //создание списка объектов

                ArrayList<Students> people = new ArrayList<Students>();
                   people.add(new Students("ФСУ","Иванов", 18));
                   people.add(new Students("ФДО","Петров", 22));
                   people.add(new Students("ФСУ","Сидоров", 20));

              try {
                 //запись объектов в файл с помощью ObjectOut-putStream
                  FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                  ObjectOutputStream o = new ObjectOutputStream(f);

                 //Запись объектов в файл myObjects.txt
                  o.writeObject(people);
                  o.close();
                  f.close();

                  //Чтение объектов с помощью ObjectInputStream
                  FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
                  ObjectInputStream oi = new ObjectInputStream(fi);


                 //Чтение объектов из файла myObjects.txt
                // десериализация в новый список
                  ArrayList<Students> newPeople= new ArrayList<Stu-
dents>();

                  newPeople=((ArrayList<Students>)oi.readObject());

                  for(Students p : newPeople)
                      System.out.printf(p.toString());

                  oi.close();
                  fi.close();

              } catch (FileNotFoundException e) {
```

```
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        } catch (ClassNotFoundException e) {

            e.printStackTrace();
        }

    }

     }
```

Вывод этой программы будет следующий (рис. 4.8).

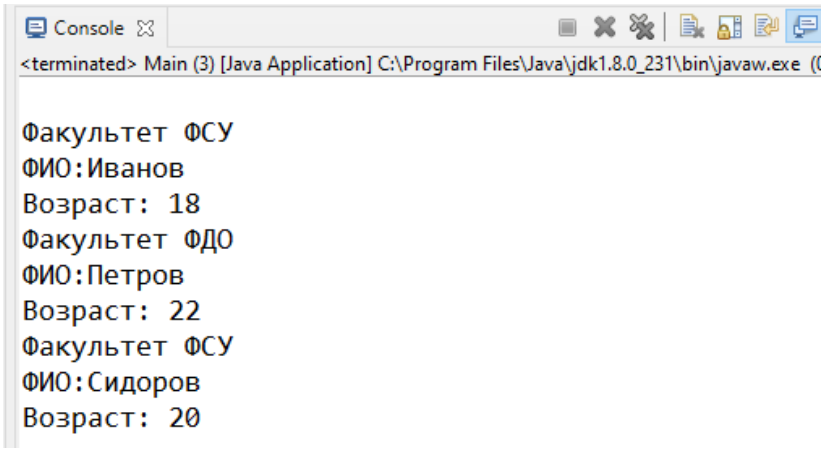


Рис. 4.8 – Сериализация статического поля

4.2.6 Сериализация Java с наследованием

Если класс реализует сериализуемый интерфейс, тогда все его подклассы также будут сериализуемыми. Давайте посмотрим на пример, приведенный ниже:

```
import java.io.Serializable;
//сериализуем класс
class Person implements Serializable{
 int age;
 String name;

  Person(String name, int age) {
        this.age = age;
        this.name = name;
       }
      }
//класс-наследник
 class Students extends Person{
```

```
String faculty;

  //конструктор

 public Students(String nameOfFaculty, String name, int age) {
  super(name, age);
  this.faculty = nameOfFaculty;
 }

 @Override
 public String toString() {
  return "\nФакультет " + faculty +"\nФИО:" + name + "\nВозраст: " + age;
 }
     }
```

Теперь вы можете сериализовать объект класса `Students`, который расширяет класс `Person`, который является сериализуемым. Свойства родительского класса наследуются подклассами, поэтому, если родительский класс является `Serializable`, подкласс также будет сериализуемым.

```
public class Main {

           public static void main(String[] args)  {
                //создание объектов p1 и p2
                Students p1 = new Students("ФДО","Иванов", 18);
                Students p2 = new Students("ФСУ","Петров", 22);

              try {
                //запись объектов в файл с помощью ObjectOut-putStream
                 FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                 ObjectOutputStream o = new ObjectOutputStream(f);

                //Запись объектов в файл myObjects.txt
                 o.writeObject(p1);
                 o.writeObject(p2);

                 o.close();
                 f.close();
                 //Чтение объектов с помощью ObjectInputStream
                 FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
                 ObjectInputStream oi = new ObjectInputStream(fi);
                //Чтение объектов из файла myObjects.txt
                 Students pr1 = (Students) oi.readObject();
```

```
            Students pr2 = (Students) oi.readObject();

            System.out.println(pr1.toString());
            System.out.println(pr2.toString());

            oi.close();
            fi.close();

        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        } catch (ClassNotFoundException e) {

            e.printStackTrace();
        }

    }

     }
```

Результат выполнения данного примера показан на рисунке 4.9.

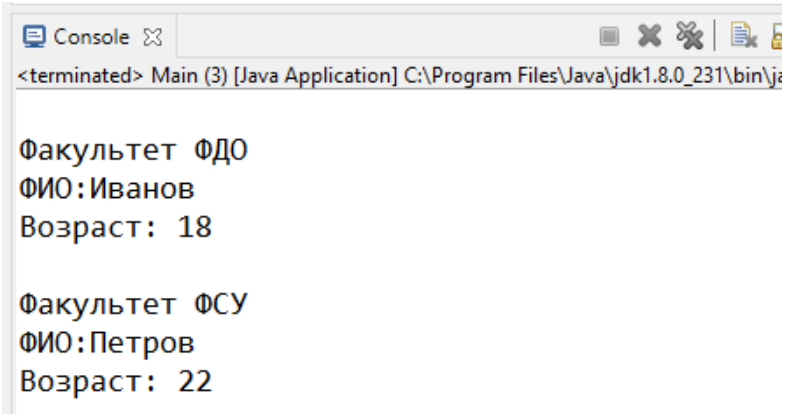


Рис. 4.9 – Пример сериализации при наследовании

4.2.7 Сериализация Java с агрегированием

Если у класса есть ссылка на другой класс, все ссылки должны быть сериализуемыми, иначе процесс сериализации не будет выполнен. В таком случае во время выполнения выбрасывается *NotSerializableException*.

```
import java.io.Serializable;
//простой класс
class Address{
 String city;
```

```
 public Address(String city) {
       this.city=city;
      }
     }

//сериализуемый класс
class Students implements Serializable{
Address address;//HAS-A
String faculty;
int age;
String name;
  //конструктор

 public Students(String city,String nameOfFaculty, String name, int age)
{
      this.address=address;
      this.age = age;
       this.name = name;
  this.faculty = nameOfFaculty;
 }

 @Override
 public String toString() {
     return "\nГород " + address +"\nФакультет " + faculty +"\nФИО:" +
name + "\nВозраст: " + age;
 }
     }
```

Далее сериализуем/десериализуем объекты класса `Students`:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class Main {

          public static void main(String[] args)  {
               //создание объектов p1 и p2
               Students p1 = new Students("Томск","ФДО","Иванов", 18);
               Students p2 = new Students("Северск","ФСУ","Петров", 22);

             try {
               //запись объектов в файл с помощью ObjectOut-putStream
```

```
                FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
                ObjectOutputStream o = new ObjectOutputStream(f);

               //Запись объектов в файл myObjects.txt
                o.writeObject(p1);
                o.writeObject(p2);

                o.close();
                f.close();
                //Чтение объектов с помощью ObjectInputStream
                FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
                ObjectInputStream oi = new ObjectInputStream(fi);

               //Чтение объектов из файла myObjects.txt
                Students pr1 = (Students) oi.readObject();
                Students pr2 = (Students) oi.readObject();

                System.out.println(pr1.toString());
                System.out.println(pr2.toString());

                oi.close();
                fi.close();

            } catch (FileNotFoundException e) {
                System.out.println("File not found");
            } catch (IOException e) {
                System.out.println("Error initializing stream");
            } catch (ClassNotFoundException e) {

                e.printStackTrace();
            }

        }

            }
```

Как видим, поле `address`, несмотря на инициализацию, осталось со значением `null` (рис. 4.10).

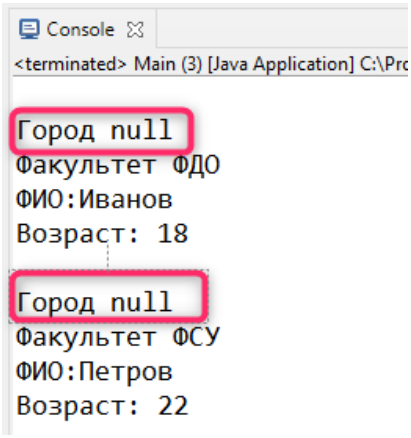


Рис. 4.10 – Сериализация при агрегации объектов

Поскольку `Address` не сериализуемый, мы не можем сериализовать объекты класса `Students`.

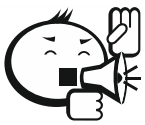
Все объекты внутри объекта должны быть сериализуемыми.

4.2.8 SerialVersionUID

Процесс сериализации во время выполнения связывает идентификатор с каждым классом `Serializable`, который известен как `SerialVersionUID`. Он используется для проверки отправителя и получателя сериализованного объекта. Отправитель и получатель должны быть одинаковыми. Отправитель и получатель должны иметь одинаковый `SerialVersionUID`, в противном случае при десериализации объекта будет создано исключение *InvalidClassException*.

`SerialVersionUID` – *элемент управления версиями в классе* `Serializable`.

Если вы явно не объявите `serialVersionUID`, JVM сделает это за вас автоматически, основываясь на различных аспектах вашего класса `Serializable`, как описано в «Спецификация сериализации объектов Java» [11].



IDE сообщит об отсутствии `SerialVersionUID`, например: *The Serializable class Customer does not declare a static final SerialVer-*

sionUID field of type long (Serializable-класс `Customer` не объявил статическое финальное поле `SerialVersionUID` типа `long`).

. . . . . .

`SerialVersionUID` должен быть типа `long` с модификаторами `static` и `final`. Предлагается явно объявить поле `serialVersionUID` в классе, а также сделать его закрытым. Например, добавим в класс `Students`:

```
private static final long serialVersionUID=1L;
```

Далее сериализуем/десериализуем объекты:

```
public class Main {

        public static void main(String[] args)  {
             //создание объектов p1 и p2
             Students p1 = new Students("ФДО","Иванов", 18);
             Students p2 = new Students("ФСУ","Петров", 22);

           try {
             //запись объектов в файл с помощью ObjectOut-putStream
              FileOutputStream f = new FileOutputStream(new File("my-
Objects.txt"));
              ObjectOutputStream o = new ObjectOutputStream(f);

             //Запись объектов в файл myObjects.txt
              o.writeObject(p1);
              o.writeObject(p2);

              o.close();
              f.close();
              //Чтение объектов с помощью ObjectInputStream
              FileInputStream fi = new FileInputStream(new File("my-
Objects.txt"));
              ObjectInputStream oi = new ObjectInputStream(fi);

             //Чтение объектов из файла myObjects.txt
              Students pr1 = (Students) oi.readObject();
              Students pr2 = (Students) oi.readObject();

              System.out.println(pr1.toString());
              System.out.println(pr2.toString());

              oi.close();
              fi.close();

           } catch (FileNotFoundException e) {
               System.out.println("File not found");
           } catch (IOException e) {
```

```
            System.out.println("Error initializing stream");
        } catch (ClassNotFoundException e) {

            e.printStackTrace();
        }

    }

     }
```

Теперь давайте изменим `serialVersionUID` константу в классе `Students`:

```
private static final long serialVersionUID = 2L;
```

Повторим попытку десериализации объекта из той же строки, полученной ранее, только закомментируем запись объектов:

```
//Запись объектов в файл myObjects.txt
//    o.writeObject(p1);
//    o.writeObject(p2);
```

Повторный запуск чтения объектов должен сгенерировать вывод (рис. 4.11).

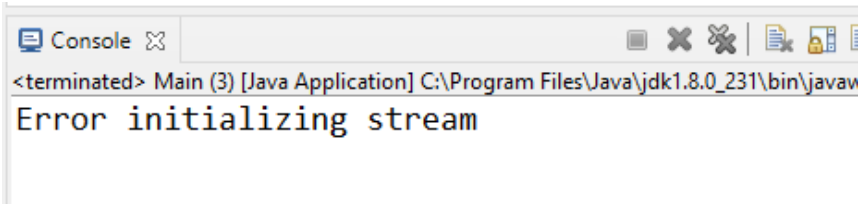


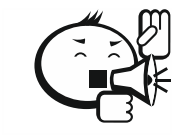
Рис. 4.11 – Пример проверки отправителя
и получателя сериализованного объекта с разными `serialVersionUID`

Таким образом, при каждом добавлении или удалении элементов класса (то есть атрибутов и методов) следует изменять его `serialVersionUID`. Изменив `serialVersionUID` класса, мы изменили его версию/состояние. В результате во время десериализации не было найдено совместимых классов и было создано исключение *InvalidClassException*.

4.3 Потоки выполнения

К большинству современных распределенных приложений и веб-приложений выдвигаются требования одновременной поддержки многих пользователей,

каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов.



Потоки – средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке.

Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существуют два способа создания и запуска потока: на основе расширения класса `Thread` или реализации интерфейса `Runnable`.

1. На основе расширения класса `Thread`.

Для управления потоком класс `Thread` предоставляет ряд методов. Наиболее используемые из них:

- `getName()`: возвращает имя потока;
- `setName(String name)`: устанавливает имя потока;
- `getPriority()`: возвращает приоритет потока;
- `setPriority(int proirity)`: устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из всех потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета – от 1 до 10. По умолчанию главному потоку выставляется средний приоритет – 5;
- `isAlive()`: возвращает `true`, если поток активен;
- `isInterrupted()`: возвращает `true`, если поток был прерван;
- `join()`: ожидает завершение потока;
- `run()`: определяет точку входа в поток;
- `sleep()`: приостанавливает поток на заданное количество миллисекунд;
- `start()`: запускает поток, вызывая его метод `run()`.

Создать собственный поток не сложно. Достаточно наследоваться от класса `Thread`. Создадим класс `TestThread`, наследник класса `Thread`, и переопределим в нем метод `run()`:

```
class TestThread extends Thread {
@Override
public void run() {
```

```
for (int i = 0; i < 10; i++) {
System.out.println("Thread"+i);
try {
Thread.sleep(7); // остановка на 7 миллисекунд
 } catch (InterruptedException e) {
System.err.print(e);    }
            }
      }
}
```

2. При реализации интерфейса `Runnable` необходимо определить его единственный абстрактный метод `run()`.

Интерфейс `Runnable` содержит только один метод `run()`.

Метод `run()` выполняется при запуске потока. После определения объекта `Runnable` он передается в один из конструкторов класса `Thread`.

Например:

```
class TestRunnable implements Runnable {
       @Override public void run() {
       for (int i = 0; i < 10; i++) {
       System.out.println("Runnable"+i);
       try {
       Thread.sleep(7);    }
       catch (InterruptedException e) {
       System.err.println(e);    }  } }
   }
```

Мы можем вывести всю информацию о потоке:

```
public class Main {

      public static void main(String[] args)  {

            // новые объекты потоков
            TestThread t = new TestThread();
            Thread w = new Thread(new TestRunnable());
           // запуск потоков
           t.run();
           w.run();
      }
}
```

Консольный вывод (рис. 4.12).

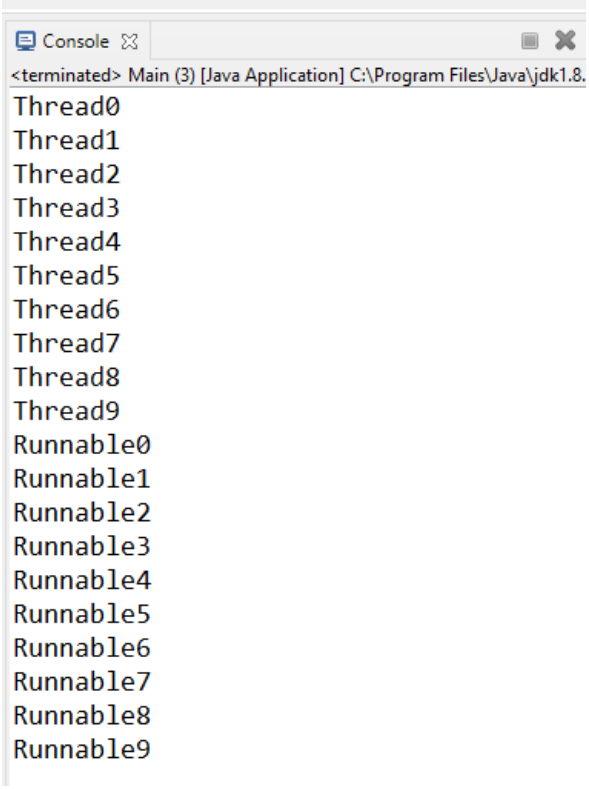


Рис. 4.12 – Прямой вызов метода `run()` в главном потоке

Интерфейс `Runnable` не имеет метода `start()`, а только единственный метод `run()`. Поэтому для запуска такого потока, как `TestRunnable`, следует создать экземпляр класса `Thread` с передачей экземпляра `TestRunnable` его конструктору:

```
Thread w = new Thread(new TestRunnable());
```

При этом видим, что при прямом вызове метода `run()` поток не запустится, выполнится только тело самого метода.

```
t.run();
w.run();
```

Чтобы использовать эти классы как потоки, мы должны вызвать метод `start()`. При этом метод `run()` выполнится в отдельном потоке.

```
public class Main {
        public static void main(String[] args)  {
              // новые объекты потоков
              TestThread t = new TestThread();
              Thread w = new Thread(new TestRunnable());
```

```
            // запуск потоков
            t.start();
            w.start();
            //t.run();
            //w.run();

        }
    }
```

Порядок вывода, как правило, различен при нескольких запусках приложения (рис. 4.13).

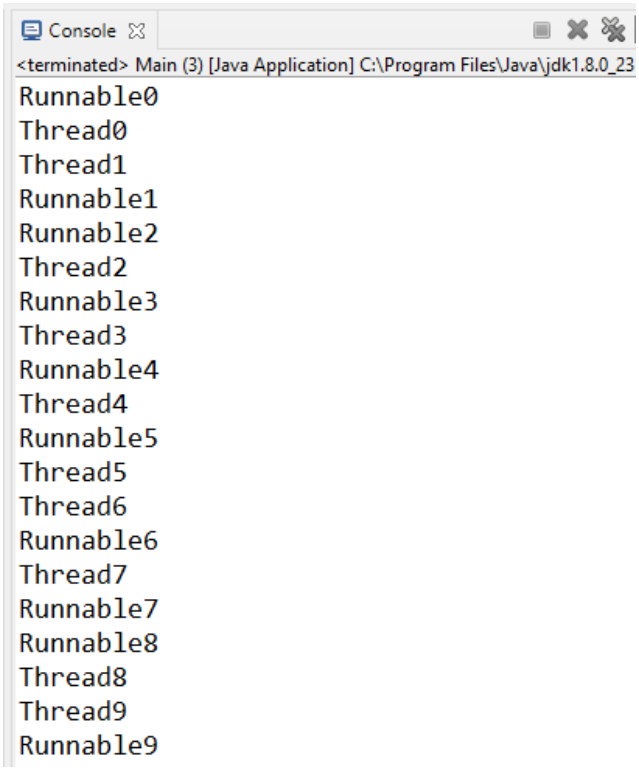


Рис. 4.13 – Реализация многопоточности с помощью метода `start()`, который, по сути, выполняет вызов метода `run()`

Если класс предоставляет больше возможностей, чем просто запускаться в виде `Thread`, то лучше реализовать интерфейс `Runnable`. Если же там просто нужно запустить в отдельном потоке, то можно наследоваться от класса `Thread`.

Реализация `Runnable` является более предпочтительной, поскольку Java поддерживает реализацию нескольких интерфейсов. Если класс наследует `Thread`, то уже нельзя наследовать другие классы.

4.4. Жизненный цикл потока

При выполнении программы объект класса `Thread` может быть в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный» (рис. 4.14).

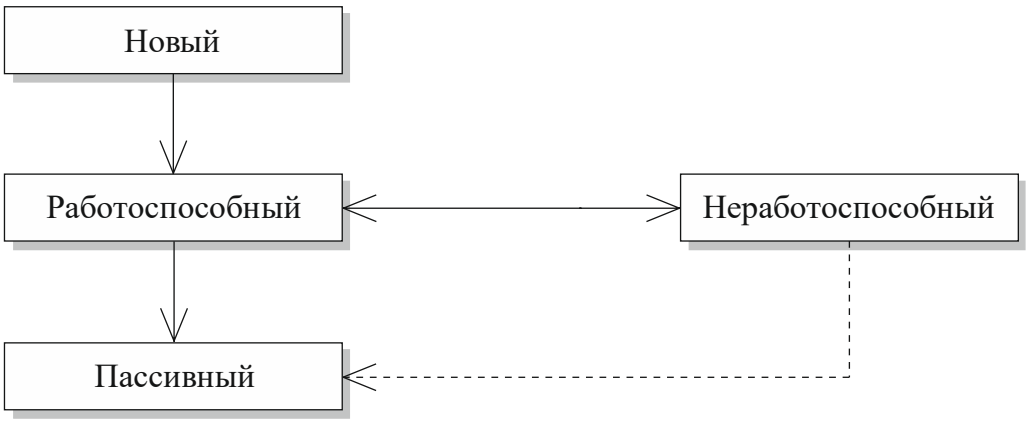


Рис. 4.14 – Состояния потока

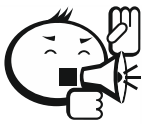
Ниже показаны различные состояния потока в Java:

- NEW – поток создан, но еще не запущен;
- RUNNABLE – поток выполняется;
- BLOCKED – поток блокирован;
- WAITING – поток ждет окончания работы другого потока;
- TIMED_WAITING – поток некоторое время ждет окончания другого потока;
- TERMINATED или DEAD – поток завершен.

При создании потока он получает состояние «новый» (NEW) и не выполняется. Для перевода потока из состояния «новый» в состояние «работоспособный» (RUNNABLE) следует выполнить метод `start()`, который вызывает метод `run()` – основной метод потока.

Получить текущее значение состояния потока можно вызовом метода `getState()`. Поток переходит в состояние «неработоспособный» в режиме ожидания (WAITING) вызовом методов `join()`, `wait()`, `suspend()` или методов ввода-вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания по времени (TIMED_WAITING) с помощью методов `yield()`, `sleep(long millis)`, `join(long timeout)` и `wait(long timeout)`, при выполнении которых может генерироваться прерывание *InterruptedException*. Вернуть потоку работо-

способность после вызова метода `suspend()` можно методом `resume()`, а после вызова метода `wait()` – методами `notify()` или `notifyAll()`. Поток переходит в «пассивное» состояние (TERMINATED), если вызваны методы `interrupt()`, `stop()` или метод `run()` завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока.



Методы `suspend()`, `resume()` и `stop()` запрещены к использованию, так как они не являются в полной мере «потокобезопасными».

Метод `interrupt()` успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток неработоспособен, например, находится в состоянии TIMED_WAITING, то метод инициирует исключение *InterruptedException*. Чтобы этого не происходило, следует предварительно вызвать метод `isInterrupted()`, который проверит возможность завершения работы потока.

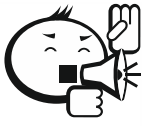
После того как поток завершает выполнение, его состояние изменяется на DEAD, то есть он отработал свое и уже не нужен.

4.5 Многопоточность

Большинство языков программирования поддерживают такую важную функциональность, как многопоточность, и Java в этом плане не исключение.

Многопоточность в Java *– это выполнение двух или более потоков одновременно для максимального использования центрального процесса.*

При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно – параллельно. Большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности.



Многопоточность позволяет писать эффективные программы, максимально использующие доступные вычислительные мощности

в системе. Еще одним преимуществом многопоточности является сведение к минимуму времени ожидания. Это особенно важно для интерактивных сетевых сред, где работают программы на java, поскольку простои в таких средах – обычное явление.

В Java есть встроенная система классов для реализации многопоточной модели программирования. Исполняющая система Java во многом зависит от потоков исполнения, и все библиотеки классов разработаны с учетом многопоточности. Многопоточная система в Java построена на основе класса `Thread`, его методах и дополняющем его интерфейсе `Runnable`. Класс `Thread` инкапсулирует поток исполнения. Чтобы создать новый поток исполнения, следует расширить класс `Thread` или же реализовать интерфейс `Runnable`.

4.5.1 Главный поток

Когда запускается любое приложение, начинает выполняться поток, называемый главным потоком (main).

Главный поток важен по нескольким причинам:

1. Он создается первым.
2. От этого потока исполнения порождаются все дочерние потоки.
3. Зачастую он должен быть последним потоком, завершающим выполнение программы, поскольку в нем производятся различные завершающие действия.

Несмотря на то что главный поток создается автоматически, им можно управлять через объект класса `Thread`. Для этого нужно вызвать метод `currentThread()`, после чего можно управлять потоком.

Метод `currentThread()` в качестве результата возвращает ссылку на поток, из которого вызывался метод. Поэтому если метод вызвать в главном методе программы (инструкция вида `Thread.currentThread()`), получим ссылку на главный поток.

Пример

```
public class Main {
          public static void main(String[] args)  {
               Thread t = Thread. currentThread();
               System.out.println("Текущий поток исполнения: "+ t);
               //изменить имя потока исполнения
```

```
            t.setName ("MyThread");
            System.out.println("После изменения имени потока: "+ t);
            try { for(int n = 0; n < 5; n++)
            System.out.println(n);
            Thread.sleep (1000);
            } catch (InterruptedException e) {
            System.out.println( "Главный поток исполнения прерван");
            }

        }
    }
```

В этом примере программы ссылка на текущий поток исполнения (в данном случае главный поток) получается в результате вызова метода `currentThread()` и сохраняется в локальной переменной `t`. Затем выводятся сведения о потоке исполнения. Далее вызывается метод `setName()` для изменения внутреннего имени потока исполнения. После этого сведения о потоке исполнения выводятся заново, а в следующем далее цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется с помощью метода `sleep()`. Аргумент метода `sleep()` задает время задержки в миллисекундах. Метод `sleep()` из класса `Thread` может сгенерировать исключение типа *InterruptException*, если в каком-нибудь другом потоке исполнения потребуется прервать ожидающий поток.

Ниже приведен результат, выводимый данной программой (рис. 4.15).

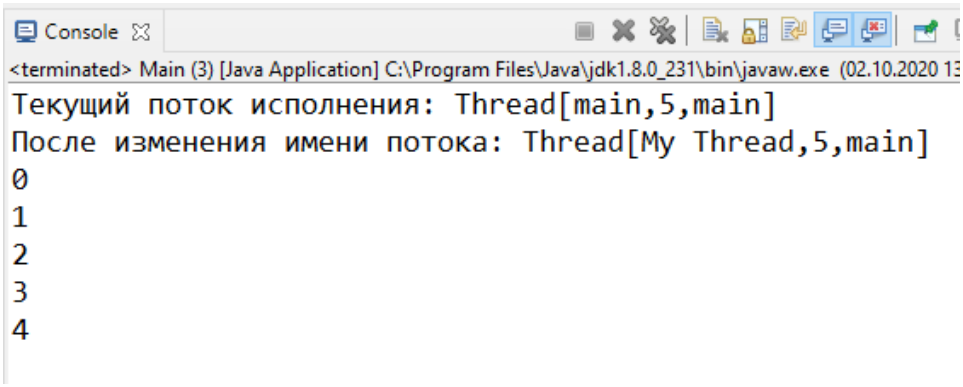


Рис. 4.15 – Пример приостановки выполнения текущего потока

. . . . . .

4.5.2 Создание и завершение потоков

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса `Thread`, либо реализуя в классе интерфейс `Runnable`.

Изменим наш класс `TestThread`. Предполагается, что в конструктор класса `TestThread` передается имя потока, которое затем передается в конструктор родительского класса. Также здесь изменим метод `run()`, код которого собственно и будет представлять весь тот код, который выполняется в потоке.

```
class TestThread extends Thread {
TestThread(String name){
super(name);
}
@Override
public     void run(){
System.out.printf("Поток %s начал работу...\n",
 Thread.currentThread().getName());
try{
      Thread.sleep(500);
}
catch(InterruptedException e){

System.out.println("Поток прерван");
      }
System.out.printf("Поток %s завершил работу...\n", Thread.cur-
rentThread().getName());
 }
      }
```

Теперь применим этот класс в главном классе программы:

```
public class Main {
      public static void main(String[] args)  {

            System.out.println("Главный      поток начал работу...");
            new   TestThread("TestThread").start();
            System.out.println("Главный      поток завершил работу...");

      }
}
```

Консольный вывод представлен на рисунке 4.16.

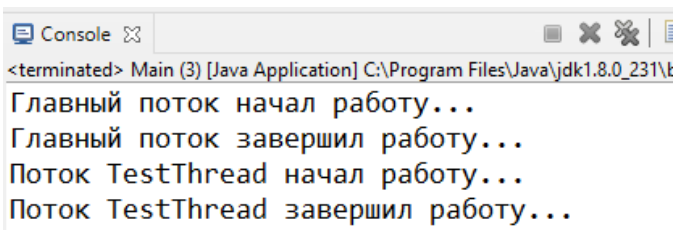


Рис. 4.16 – Пример запуска и завершения текущего и главного потоков

Здесь в методе `main` в конструктор `Thread` передается произвольное название потока, и затем вызывается метод `start()`. По сути этот метод как раз и вызывает переопределенный метод `run()` класса `Thread`. Обратите внимание, что главный поток завершает работу раньше, чем порожденный им дочерний поток `Thread`.

Как правило, более распространенной ситуацией является случай, когда главный поток завершается самым последним. Для этого надо применить метод `join()`:

```
public class Main {
          public static void main(String[] args)  {

                System.out.println("Главный поток начал работу...");
                TestThread t= new TestThread("TestThread");
                t.start();
                try{
                t.join();
                }
                catch(InterruptedException e){
                System.out.printf("Поток %s прерван", t.getName());
                }
                System.out.println("Главный поток завершил работу...");
                }

     }
```

Метод `join()` заставляет вызвавший поток (в данном случае главный поток) ожидать завершения вызываемого потока, для которого и применяется метод `join` (в данном случае поток `TestThread`).

Консольный вывод представлен на рисунке 4.17.

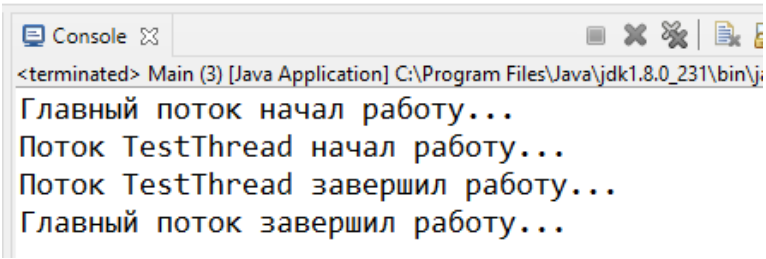


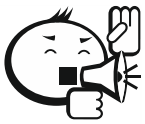
Рис. 4.17 – Пример управления последовательности выполнения потоков

Если в программе используется несколько дочерних потоков и надо, чтобы главный поток завершался после дочерних, то для каждого дочернего потока надо вызвать метод `join()`.

Реализация интерфейса `Runnable` во многом аналогична переопределению класса `Thread`. Мы его уже рассмотрели ранее.

4.5.3 Завершение потока

Особо следует остановиться на механизме завершения потока. Во всех примерах выше потоки завершались после выполнения последней операции. Однако нередко имеет место и другая организация потока, и тогда мы также должны предусмотреть механизм завершения потока.



В Java существуют средства для принудительного завершения потока. В частности, метод `Thread.stop()` завершает поток незамедлительно после своего выполнения. Однако этот метод, а также `Thread.suspend()`, приостанавливающий поток, и `Thread.resume()`, продолжающий выполнение потока, были объявлены устаревшими, и их использование отныне крайне нежелательно. Дело в том, что поток может быть «убит» во время выполнения операции, обрыв которой на полуслове оставит некоторый объект в неправильном состоянии, что приведет к появлению трудноотлавливаемой и случайным образом возникающей ошибки.

Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за свое завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода `run()` (`main()` – для главного потока), либо по сообщению из другого потока.

Как правило, это делается с помощью опроса логической переменной. И если она равна, например, `false`, то поток завершает бесконечный цикл и заканчивает свое выполнение. Изменим наш `TestThread`:

```
class TestRunnable implements Runnable {
      private boolean isActive;
      void disable(){
      isActive=false;
      }
      TestRunnable(){
```

```
        isActive = true;
        }
        public void run(){
        System.out.printf("Поток %s начал работу... \n", Thread.cur-
rentThread().getName());
         int counter=1; // счетчик циклов
         while(isActive){
          System.out.println("Цикл " + counter++);

        try{
        Thread.sleep(500);
             }

        catch(InterruptedException     e){

        System.out.println("Поток прерван");
             }                                          }

        System.out.printf("Поток %s завершил работу...\n", Thread.cur-
rentThread().getName());
        }
       }
```

Переменная `isActive` указывает на активность потока. С помощью метода `disable()` мы можем сбросить состояние этой переменной. Теперь используем этот класс:

```
public class Main {
           public static void main(String[] args)  {
System.out.println("Главный      поток начал работу...");
TestRunnable tThread = new TestRunnable();
new   Thread(tThread,"TestThread").start();
try{
Thread.sleep(1100);
tThread.disable();
Thread.sleep(1000);
} catch(InterruptedException     e){
System.out.println("Поток  прерван");
}
System.out.println("Главный поток     завершил работу...");
      }
      }
```

Вначале запускается дочерний поток:

```
new Thread(tThread,"TestThread").start();
```

Затем на 1100 миллисекунд останавливаем главный поток и потом вызываем метод `tThread.disable()`, который переключает в потоке флаг `isActive`. И дочерний поток завершается.

Консольный вывод представлен на рисунке 4.18.

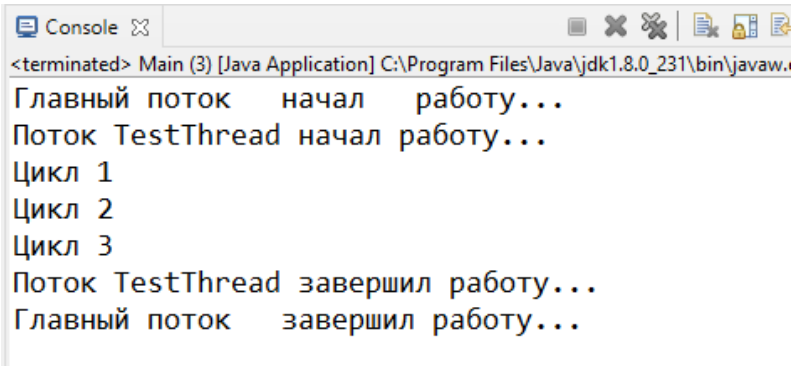


Рис. 4.18 – Пример прерывания потока

4.5.4 Управление приоритетами

Каждый поток в системе имеет свой приоритет.

Приоритет потоков *– это некоторое число в объекте потока, более высокое значение которого означает больший приоритет.*

Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда более привилегированные потоки простаивают.

Потоку можно назначить приоритет от 1 (MIN_PRIORITY) до 10 (MAX_PRIORITY) с помощью метода `setPriority(int)`. Получить значение приоритета можно с помощью метода `getPriority()`.

Пример

```
class TestThread extends Thread {
TestThread(String name){
super(name);
}
```

```
@Override
public     void run(){
for (int i = 0; i < 10; i++){
System.out.println(getName() + " " + i);
try {
Thread.sleep(0);//попробовать sleep(1000);
}
catch (InterruptedException e) {
System.err.print("Error" + e); }
}

}
}
```

Далее выставим приоритеты потокам:

```
public class Main {
          public static void main(String[] args)  {
TestThread min = new TestThread("Min");//1
TestThread max = new TestThread("Max");//10
TestThread norm = new TestThread("Norm");//5
min.setPriority(Thread.MIN_PRIORITY);
max.setPriority(Thread.MAX_PRIORITY);
norm.setPriority(Thread.NORM_PRIORITY);
min.start();

norm.start(); max.start();
}
}
```

. . . . . .

Поток с более высоким приоритетом в данном случае, как правило, монополизирует вывод на консоль (рис. 4.19).

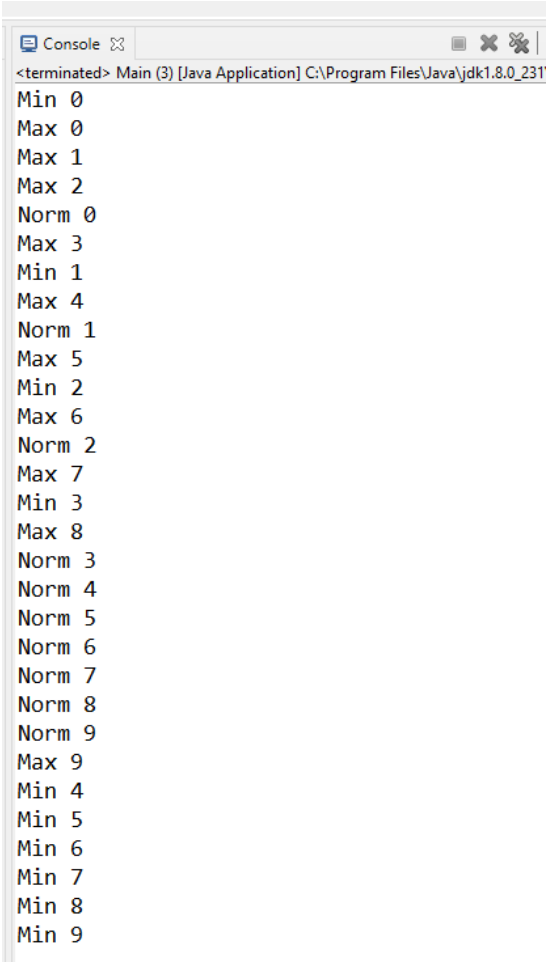


Рис. 4.19 – Пример управления потоками с помощью приоритетов

4.5.5 Синхронизация потоков

В многопоточности Java присутствует *асинхронное* поведение. Если один поток записывает некоторые данные, а другой в это время их считывает, в приложении может возникнуть ошибка. Поэтому при одновременном доступе к общему ресурсу несколькими потоками используется *синхронизация*.

В Java есть свои методы для обеспечения синхронизации. Простейший способ синхронизации – концепция *монитора* и *synchronized*.

Монитор (**monitor**) – *это специальный объект, который следит за «состоянием» метода или объекта. Он смотрит, «занят» ресурс или «свободен» в данный момент.*

Каждый объект в Java имеет ассоциированный с ним монитор. Монитор представляет своего рода инструмент для управления доступа к объекту.

Концепция «монитор» внутри себя содержит 4 поля:

1. `locked` типа `boolean`, которое показывает, захвачен монитор или нет;
2. `owner` типа `Thread` – в это поле записывается поток, который захватил данный монитор;
3. `blocked set` – в это множество попадают потоки, которые не смогли захватить блокировку, или поток, который выходит из состояния `wait`;
4. `wait set` – в это множество попадают потоки, для которых был вызван метод `wait`.

`Synchronized` – *это ключевое слово, которое позволяет заблокировать доступ к методу или части кода, если его уже использует другой поток.*

Когда выполнение кода доходит до оператора `synchronized`, монитор объекта блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток, который и произвел блокировку. После окончания работы блока кода монитор объекта освобождается и становится доступным для других потоков. После освобождения монитора его захватывает другой поток, а все остальные потоки продолжают ожидать его освобождения.

Использование ключевого слова `synchronized` гарантирует, что блоки кода будут выполняться только одним потоком в каждую конкретную единицу времени.

Существует два применения `synchronized` – для метода и для блока кода.

Первый способ: для блока кода.

Если Вам не нужно синхронизировать весь метод, а только его часть, например объект, можно использовать следующую форму `synchronized`:

Пример

```
class CommonResource{

    int x=0;
```

```
}


class TestThread extends Thread{
CommonResource t;

TestThread(CommonResource t){
this.t=t;
}

public void run(){
      t.x=1;
      for(int i=1;i<=5;i++){
            System.out.printf("%s %d \n", Thread.currentThread().get-
Name(), t.x);
         t.x++;
           try{
            Thread.sleep(0);
           }catch(Exception e){System.out.println(e);}
        }
}

      }
```

Здесь определен класс `CommonResource`, который представляет общий ресурс и в котором определено одно целочисленное поле `x`.

Этот ресурс используется классом `TestThread`. В методе он просто увеличивает в цикле значение `x` на единицу. Причем при входе в поток значение `x=1`.

Организуем 5 таких потоков.

```
public class Main {
      public static void main(String[] args)  {
            CommonResource obj= new CommonResource();//only one object
                  for (int i = 1; i < 6; i++){
                  Thread t = new Thread(new TestThread(obj));
             t.setName("Thread "+ i);
             t.start();
}
            }

      }
```

То есть мы ожидаем, что каждый поток будет увеличивать `t.x` с 1 до 4, и так пять раз. Но если мы посмотрим на результат работы программы, то он будет иным (рис. 4.20).

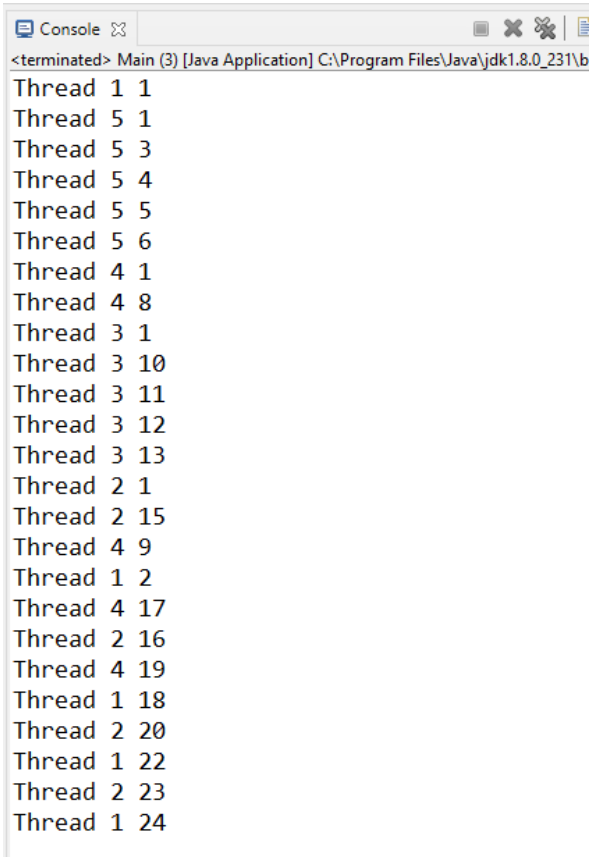


Рис. 4.20 – Выполнения потоков без синхронизации

Синхронизируем блок в методе `run()` в классе `TestThread`. При создании синхронизированного блока кода после оператора `synchronized` идет объект-заглушка: `synchronized(res)`.

```
public void run(){
     synchronized(t){ //synchronized block
          t.x=1;
          for (int i = 1; i < 5; i++){
          System.out.printf("%s %d \n", Thread.currentThread().get-
Name(), t.x);
          t.x++;
          try{
          Thread.sleep(100);
          }
          catch(InterruptedException e) {
               }
          }
     }
     }
```

В итоге консольный вывод изменится (рис. 4.21).

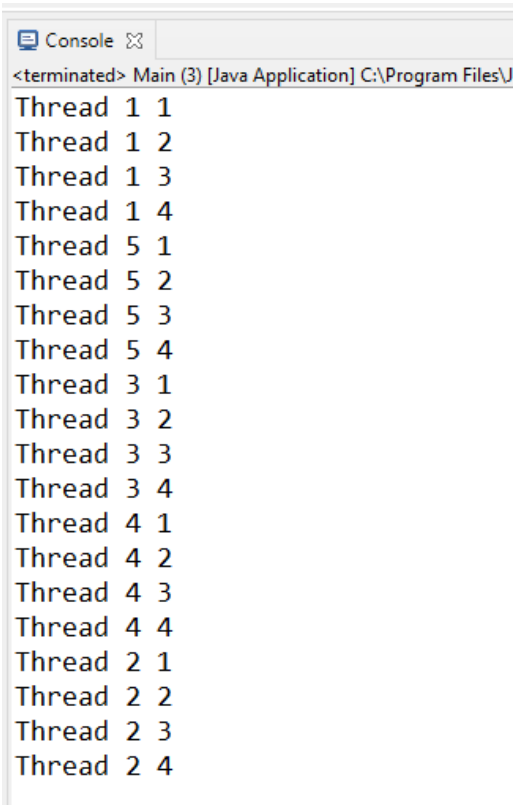


Рис. 4.21 – Пример синхронизации части метода

При применении оператора `synchronized` монопольный доступ имеет только один поток – первый, который начал его выполнение. Пока он не завершит, все остальные потоки ждут.

Второй способ: синхронизация всего метода.

Для применения `synchronized` к методу изменим классы программы:

Пример

Добавим `synchronized` к методу `count()` в классе `CommonResourse`.

```
class CommonResource{
      int x;
    synchronized void count(){//synchronized method
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().get-
Name(), x);
            x++;
            try{
                Thread.sleep(100);
```

```
            }
            catch(InterruptedException e){}
        }
    }
}


class TestThread extends Thread{
       CommonResource t;
          TestThread(CommonResource t){
              this.t=t;
          }

          public void run(){
              t.count();
          }
}

public class Main {
public static void main(String[] args)  {
CommonResource obj= new CommonResource();//only one object
                 for (int i = 1; i < 6; i++){
                Thread t = new Thread(new TestThread(obj));
             t.setName("Thread "+ i);
             t.start();
                  }
     }

     }
```

Результат работы в данном случае будет аналогичен примеру выше с блоком `synchronized` (рис. 4.22). Здесь опять в дело вступает *монитор* объекта `CommonResource` – общего объекта для всех потоков. Поэтому синхронизированным объявляется не метод `run()` в классе `TestThread`, а метод `count()` класса `CommonResource`. Когда первый поток начинает выполнение метода `count()`, он захватывает монитор объекта `CommonResource`, а все потоки продолжают ожидать его освобождения.

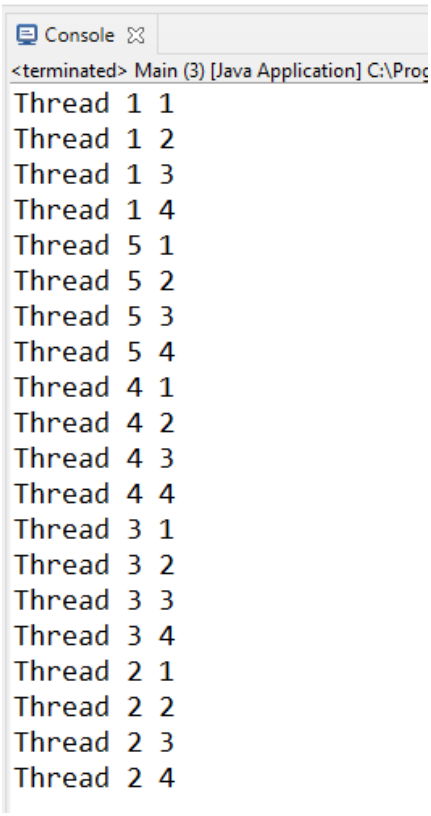


Рис. 4.22 – Пример синхронизации всего метода

Само по себе использование `synchronized` относится к доступу к разделяемым ресурсам, что достигается выполнением отдельных участков кода (с помощью монитора). Мы только устанавливаем порядок доступа к общему ресурсу участков кода. Тут имеется в виду синхронизация по истинности какого-либо утверждения насчет разделяемого ресурса (переменной или объекта), например, «подождать пока i не станет 5».

Существуют еще способы синхронизации потоков в Java. Например, синхронизация с использованием `wait()/notify()`.

Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод `wait()`, предварительно захватив его монитор. На этом его работа приостанавливается. Другой поток может вызвать на этом же самом объекте метод `notify()` (опять же, предварительно захватив монитор объекта), в результате чего ждущий на объекте поток «просыпается» и продолжает свое выполнение.

В отличие от `sleep()` и `join()`, `wait()` нельзя вызвать просто так. Выполняется метод `wait()` на объекте, на мониторе

которого мы хотим выполнить ожидание. Если никто не ждет, то и `notify()` ничего не делает.

Пример

`CommonResource` – класс, с которым будут работать потоки и вызывать методы `wait()` и `notify()`.

```
class CommonResource {
    private String res;

    public CommonResource(String str){
        this.res=str;
    }

    public String getRes() {
        return res;
    }

    public void setRes(String str) {
        this.res=str;
    }
     }
```

Класс `Waiter` будет ждать, пока другие потоки вызовут метод `notify()` для завершения работы. Обратите внимание, что поток `Waiter` владеет монитором объекта `CommonResource` с помощью синхронизированного блока. Вызов `wait()` происходит в синхронизированном блоке. Следовательно, прямо после вызова монитор должен быть захвачен. Внутри же метода `wait()` монитор отпускается и захватывается потоком, вызывающим `notify()`.

```
class Waiter implements Runnable{

        private CommonResource t;

        public Waiter(CommonResource m){
            this.t=m;
        }

        @Override
        public void run() {
            String name = Thread.currentThread().getName();
```

```
            synchronized (t) {
                try{
                    System.out.println(name+" жду notify() секунд
:"+TimeUnit.MILLISECONDS.toSeconds(System.currentTimeMillis()));
                    t.wait();
                }catch(InterruptedException e){
                    e.printStackTrace();
                }
                System.out.println(name+" получил notify() на секунд:
"+TimeUnit.MILLISECONDS.toSeconds(System.currentTimeMillis()));
                //обработаем сообщение
                System.out.println(name+" обработал: "+t.getRes());
            }
        }
    }
```

Класс `Notifier` будет обрабатывать сообщения, а затем вызывать метод `notify()`, чтобы разбудить потоки, ожидающие сообщения. Обратите внимание, что синхронизированный блок используется для управления монитором объекта сообщения.

```
class Notifier implements Runnable {

        private CommonResource t;

        public Notifier(CommonResource res) {
            this.t = res;
        }

        @Override
        public void run() {
            String name = Thread.currentThread().getName();
            System.out.println(name+" стартовал");
            try {
                Thread.sleep(1000);
                synchronized (t) {
                    t.setRes(name+" завершил работу");
                    t.notify();
                    //t.notifyAll();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
      }
```

В главном классе `Main` создадим несколько потоков `Waiter` и `Notifier` и запустим их.

```
public class Main {
public static void main(String[] args)  {
      CommonResource obj = new CommonResource("Ресурс");
    Waiter waiter1 = new Waiter(obj);
    new Thread(waiter1,"Waiter1").start();

    Waiter waiter2 = new Waiter(obj);
    new Thread(waiter2, "Waiter2").start();

    Notifier notifier = new Notifier(obj);
    new Thread(notifier, "Notifier").start();
    System.out.println("Все потоки стартовали");
      }
}
```

В результате мы увидим (рис. 4.23), что программа будет не завершена, потому что есть два потока, ожидающих объекта `CommonResource`, а метод `notify()` разбудил только один из них, а другой поток все еще ожидает уведомления.

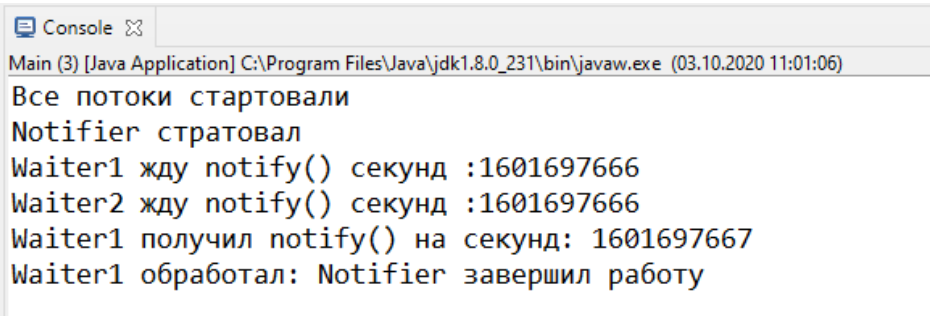


Рис. 4.23 – Пример синхронизации потоков с помощью `wait()` и `notify()`

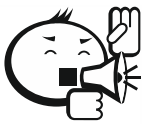
Есть еще метод `notifyAll()`, который служит для одной цели – отправить в работу ВСЕ ожидающие потоки. В то время как `notify()` действует только на один.

Закомментируем вызов `notify()` и раскомментируем вызов `notifyAll()` в классе `Notifier` (рис. 4.24).

```
Console
<terminated> Main (3) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe  (03.10.2020 11:08:36 – 11:08:37)
Все потоки стартовали
Notifier стратовал
Waiter1 жду notify() секунд :1601698116
Waiter2 жду notify() секунд :1601698116
Waiter2 получил notify() на секунд: 1601698117
Waiter2 обработал: Notifier завершил работу
Waiter1 получил notify() на секунд: 1601698117
Waiter1 обработал: Notifier завершил работу
```

Рис. 4.24 – Синхронизация с помощью метода `notifyAll()`

Метод `join()` позволяет текущему потоку «приостановить» выполнение своего кода и «пропустить» вперед другой поток. Другими словами, он заставляет текущие запущенные потоки прекращать выполнение, пока поток, к которому он присоединяется, не завершит свою задачу.



Разница между `join()` и `synchronized` в том, что `join()` ожидает полного завершения потока, в то время как блок `synchronized` может использоваться для предотвращения одновременного выполнения двумя потоками одного и того же фрагмента кода.

Также есть `join(long millis)`, этот метод приостановит выполнение текущего потока на указанное время в миллисекундах.

Один пример `join()` мы уже рассмотрели. Рассмотрим еще один:

Пример

```
class CommonResource{

    int x=0;
}


class TestThread extends Thread{
CommonResource t;

TestThread(CommonResource t){
this.t=t;
}
```

```
public void run(){
      t.x=1;
      for(int i=1;i<=5;i++){
           System.out.printf("%s %d \n", Thread.currentThread().get-
Name(), t.x);
          t.x++;
            try{
             Thread.sleep(0);
            }catch(Exception e){System.out.println(e);}
         }
}

}

public class Main {
public static void main(String[] args)  {
      CommonResource obj= new CommonResource();//only one object
      TestThread t1 = new TestThread(obj);
      TestThread t2 = new TestThread(obj);
      TestThread t3 = new TestThread(obj);

      t1.setName("TestThread 1");
      t2.setName("TestThread 2");
      t3.setName("TestThread 3");

      t1.start();

    //запустить поток t2 после ожидания в течение 2 секунд или если t1
завершится
    try {
        t1.join(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    t2.start();
    t3.start();

    //пусть все потоки завершат выполнение до завершения главного потока
    try {
        t1.join();
        t2.join();
        t3.join();
    } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("Все потоки завершились, завершение главного по-
тока");
}
```

```
         }
```

В результате мы увидим, что `t3` начинается до окончания `t2` (рис. 4.25).

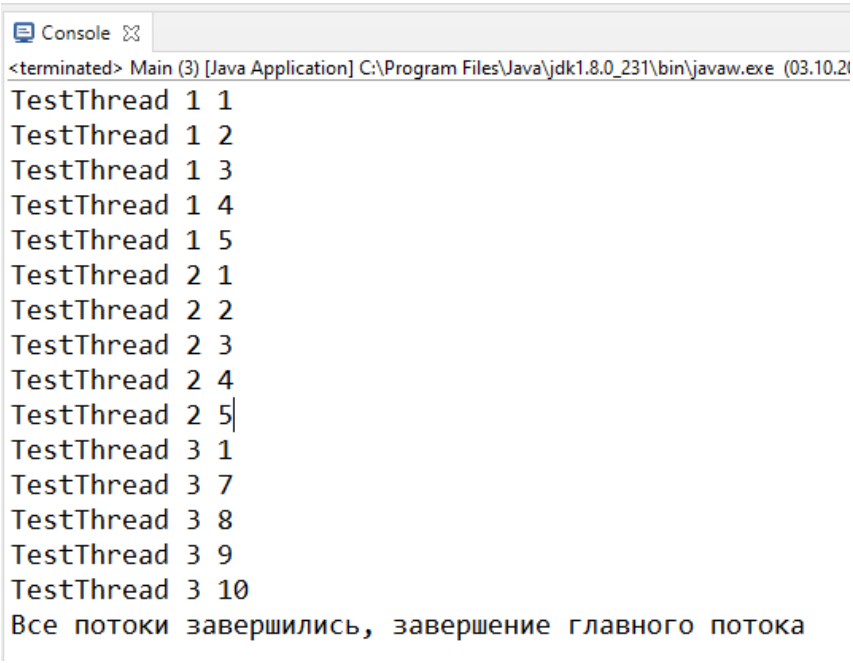


Рис. 4.25 – Синхронизация потоков с помощью метода `join()`

Запустим третий поток только тогда, когда поток `t2` завершится.

```
try {
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }


             t3.start();
```

В результате мы увидим, что `t3` начинается работать после завершения работы `t2` (рис. 4.26).

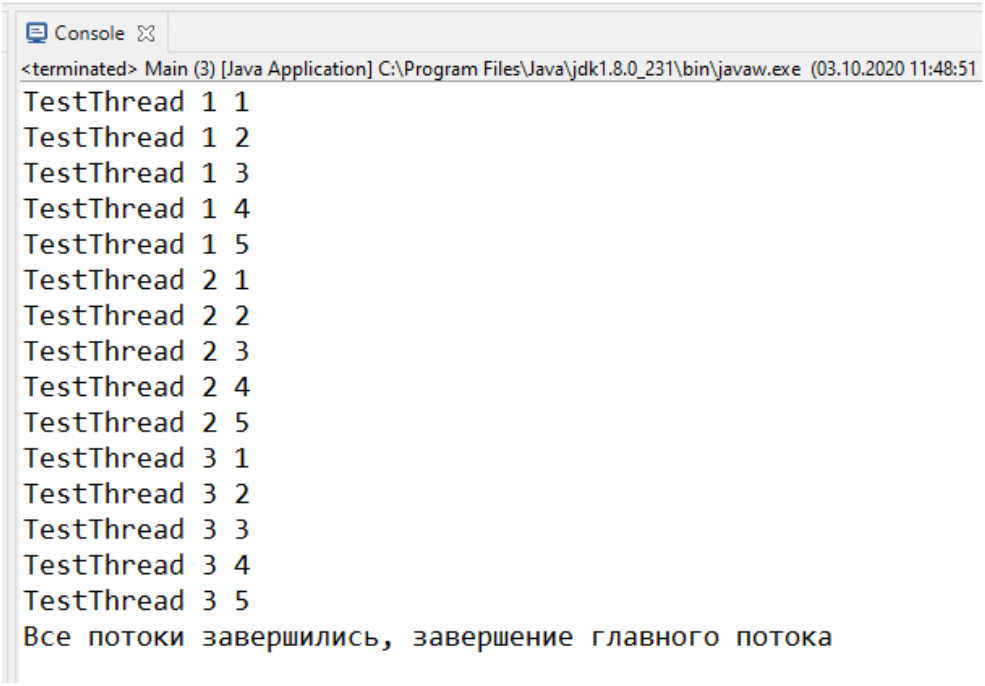


Рис. 4.26 – Управление потоками с помощью метода `join()`

Пакет `java.util.concurrent` предоставляет набор классов для организации межпоточного взаимодействия.

Пакет `java.util.concurrent` включает следующие объекты синхронизации:

- `Semaphore` – объект синхронизации, ограничивающий количество потоков, которые могут «войти» в заданный участок кода;
- `CountDownLatch` – объект синхронизации, разрешающий вход в заданный участок кода при выполнении определенных условий;
- `CyclicBarrier` – объект синхронизации типа «барьер», блокирующий выполнение определенного кода для заданного количества потоков;
- `Exchanger` – объект синхронизации, позволяющий провести обмен данными между двумя потоками;
- `Phaser` – объект синхронизации типа «барьер»; в отличие от `CyclicBarrier` предоставляет больше гибкости;
- `Semaphore` (*семафор*) – объект синхронизации пакета `java.util.concurrent`, ограничивающий одновременный доступ к общему ресурсу нескольким потокам с помощью счетчика.

Рассмотрим более подробно только *семафор*.

Для управления доступом к ресурсу *семафор* использует *счетчик*, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора. Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса `Semaphore`.

Класс `Semaphore` имеет два конструктора. Оба конструктора получают в качестве параметра количество одновременно разрешенных доступов к объекту и инициализируют этим значением счетчик. Второй конструктор дополнительно получает параметр «справедливости», позволяющий определить очередность предоставления разрешения ожидающим потокам.

```
Semaphore(int permits);
Semaphore(int permits, boolean fair);
```

Для получения разрешения у семафора надо вызвать метод `acquire()`, который имеет две формы.

Для получения одного разрешения:

```
void acquire() throws InterruptedException;
```

Для получения нескольких разрешений:

```
void acquire(int permits) throws InterruptedBxception;
```

После вызова этого метода пока поток не получит разрешение, он блокируется.

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода `release()`.

Метод освобождает одно разрешение:

```
void release();
```

Метод освобождает количество разрешений, указанных в `permits`:

```
void release(int permits);
```

Используем *семафор* в простом примере:

```
import java.util.concurrent.Semaphore;
```

```
class CommonResource{

    int x=0;
}

class TestThread extends Thread {

    CommonResource res;
    Semaphore sem;
    String name;
    TestThread(CommonResource res, Semaphore sem, String name){
        this.res=res;
        this.sem=sem;
        this.name=name;
    }

    public void run(){

        try{
            System.out.println(name + " ожидает разрешения");
            sem.acquire();
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.println(this.name + ": " + res.x);
                res.x++;
                Thread.sleep(100);
            }
        }
        catch(InterruptedException e){System.out.println(e.getMes-
sage());}
        System.out.println(name + " освобождает разрешение");
        sem.release();
    }
}


public class Main {
public static void main(String[] args)  {
       Semaphore sem = new Semaphore(1); // 1 разрешение
     CommonResource res = new CommonResource();
     TestThread t1 = new TestThread(res, sem, "Thread 1");
     TestThread t2 = new TestThread(res, sem, "Thread 2");
     TestThread t3 = new TestThread(res, sem, "Thread 3");

     t1.start();
```

```
        t2.start();
        t3.start();

       //пусть все потоки завершат выполнение до завершения главного потока
      try {
            t1.join();
            t2.join();
            t3.join();
       } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
       }


       System.out.println("Все потоки завершились, завершение главного по-
тока");
}
```

На консоли увидим следующий результат (рис. 4.27).

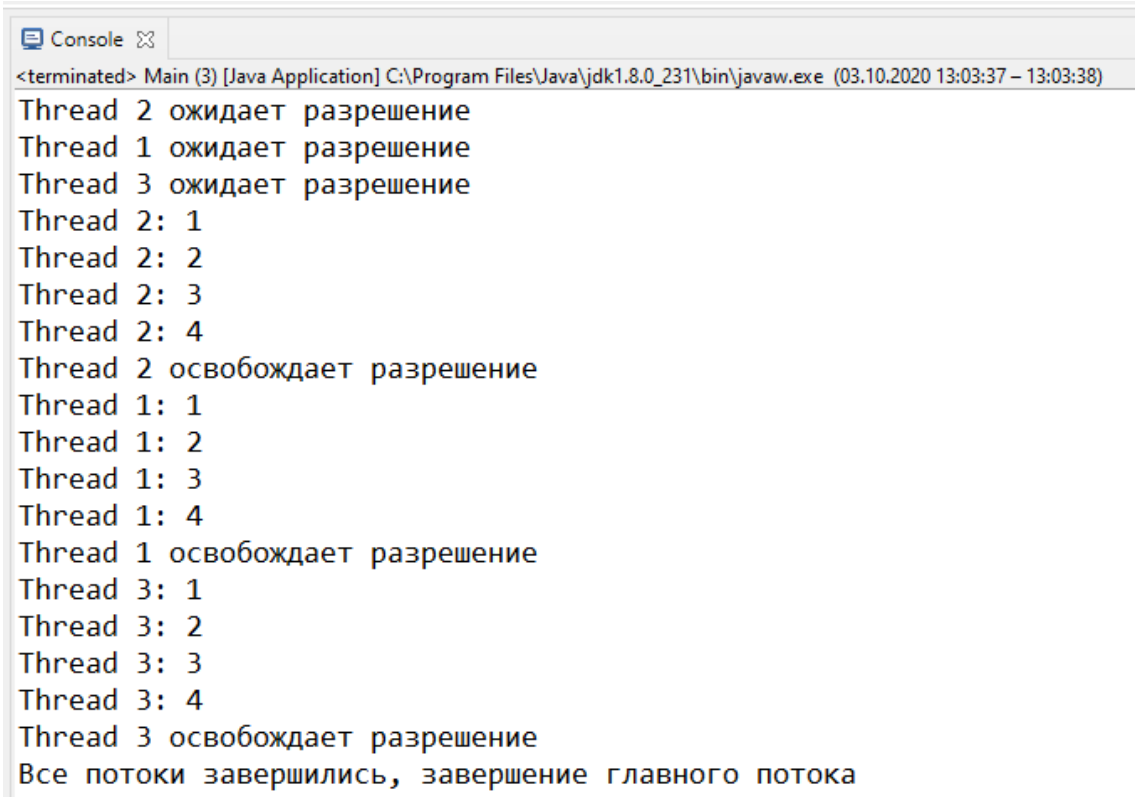
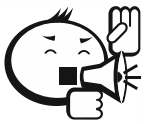


Рис. 4.27 – Реализация синхронизации с помощью семафора



Отличие `synchronized` и `Semaphore` в том, что при `synchronized` метод может быть доступен для одного потока за

раз. При использовании семафора к методу может обращаться множество потоков одновременно. В отличие от `synchronized`, в случае семафоров код должен обрабатывать получение и освобождение блокировок/разрешений, которые указываются при инициализации семафора. В случае синхронизации об этом позаботилась сама JVM.

4.5.6 Состояния потока

В классе `Thread` объявлено внутреннее перечисление `State`, простейшее применение элементов которого призвано помочь в отслеживании состояний потока в процессе функционирования приложения и, как следствие, в улучшении управления им.

Создадим простой поток.

```
class TestThread extends Thread {
public void run() {
try {
Thread.sleep(50);
} catch (InterruptedException e) {
System.err.print("ошибка потока");
            }
      }
      }
```

Далее в главном классе определим состояние потока на разных этапах жизненного цикла.

```
public class Main {
public static void main(String[] args)  {

      try{
            Thread thread = new TestThread();   // NEW – поток создан, но
еще не запущен
            System.out.println("1: " + thread.getState());
            thread.start(); // RUNNABLE – поток запущен
            System.out.println("2: " + thread.getState());
            Thread.sleep(10); // TIMED_WAITING
            // поток ждет некоторое время окончания работы другого потока
            System.out.println("3: " + thread.getState());
            thread.join();
             // TERMINATED –  поток завершил выполнение
            System.out.println("4: " + thread.getState()); }
```

```
        catch (
       InterruptedException e) {
       System.err.print("ошибка потока"); }
}

}
```

В результате компиляции и запуска на консоли увидим результат (рис. 4.28).

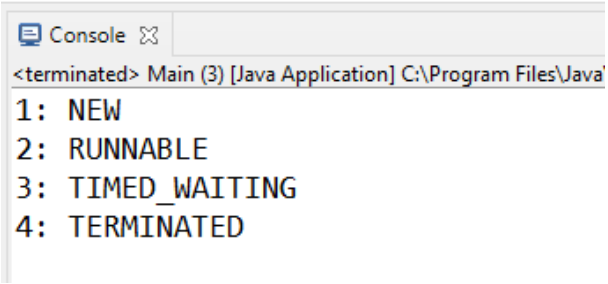


Рис. 4.28 – Определение состояния потоков

4.5.7 Блокировка

Для управления доступом к общему ресурсу в качестве альтернативы оператору `synchronized` мы можем использовать блокировки. Функциональность блокировок заключена в пакете `java.util.concurrent.locks`.

Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то поток на ресурс накладывает блокировку. После завершения работы потока блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.

Классы блокировок реализуют интерфейс `Lock`.

Реализации интерфейса `Lock` существенно расширяют возможности блокировок по сравнению с `synchronized`. Интерфейс `Lock` позволяет осуществлять более гибкое структурирование и поддерживает многократно связанный условный объект `Condition`.

`Lock` определяет следующие методы:

- `lock()` – получение блокировки (пример);
- `lockInterruptibly()` – получение блокировки, если текущий поток не прерывается (пример);

- `newCondition()` – получение нового `Condition`, связанного с блокировкой `Lock` (пример);
- `tryLock()` – получение блокировки, если она свободна во время вызова;
- `tryLock(long time, TimeUnit unit)` – получение блокировки в течение заданного времени;
- `unlock()` – освобождение блокировки;

Объект `Condition` позволяет управлять блокировкой.

Как правило, для работы с блокировками используется класс `ReentrantLock` из пакета `java.util.concurrent.locks`. Данный класс реализует интерфейс `Lock`, поэтому для создания объектов блокировки используют его реализацию. Класс `ReentrantLock` позволяет потоку блокировать метод, даже если он уже заблокировал другой метод. Полное описание интерфейса `Lock` дано в документации [12].

Перепишем наш пример.

```
import java.util.concurrent.locks.ReentrantLock;

class CommonResource{

    int x=0;
}

class TestThread implements Runnable{

    CommonResource res;
    ReentrantLock locker;
    TestThread(CommonResource res, ReentrantLock lock){
        this.res=res;
        locker = lock;
    }
    public void run(){

        locker.lock(); // устанавливаем блокировку
        try{
            res.x=1;
            for (int i = 1; i < 5; i++){
                System.out.printf("%s %d \n", Thread.currentThread().get-
Name(), res.x);
                res.x++;
```

```
                Thread.sleep(100);
            }
        }
        catch(InterruptedException e){
            System.out.println(e.getMessage());
        }
        finally{
            locker.unlock(); // снимаем блокировку
        }
    }
     }
```

Здесь также используется `CommonResource`, для управления которым создается 5 потоков. На входе в метод `run()` устанавливается заглушка:

```
locker.lock();
```

После этого только один поток имеет доступ к этому методу, а остальные потоки ожидают снятия блокировки. В блоке `finally` после окончания всей основной работы потока эта блокировка снимается. Причем делается это обязательно в блоке `finally`, так как в случае возникновения ошибки все остальные потоки окажутся заблокированными.

```
public class Main {
public static void main(String[] args)  {
      CommonResource commonResource= new CommonResource();
     ReentrantLock locker = new ReentrantLock(); // создаем заглушку
     for (int i = 1; i < 6; i++){

         Thread t = new Thread(new TestThread(commonResource, locker));
         t.setName("Thread "+ i);
         t.start();
     }

}
     }
```

В итоге мы получим вывод, аналогичный тому, который был в случае с оператором `synchronized` (рис. 4.29).

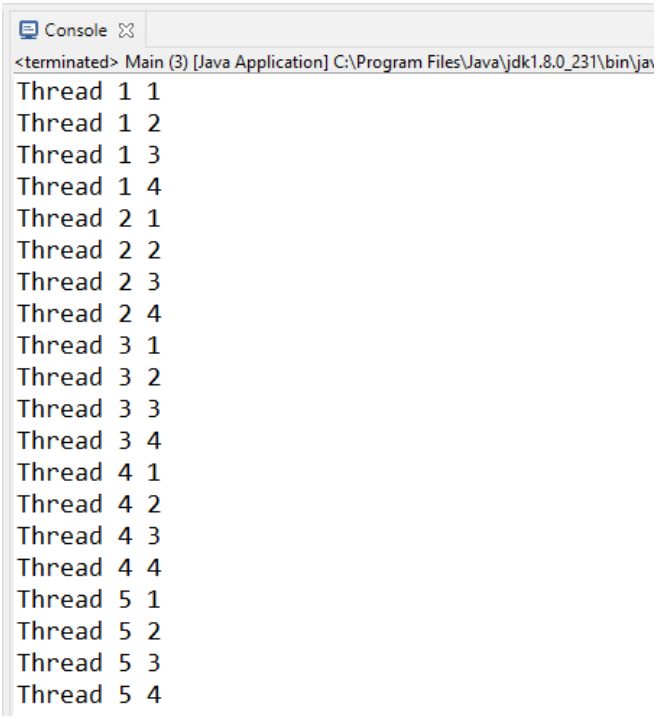


Рис. 4.29 – Пример распределения ресурсов между потоками с помощью блокировки ресурсов

Контрольные вопросы по главе 4

1. Что такое `serialVersionUID`? Что будет, если его не определить?
2. Что будет, если класс `Serializable`, а его класс-родитель – нет?
3. Что такое поток?
4. Что такое класс `Thread`?
5. Что такое интерфейс `Runnable`?
6. Что такое метод или блок кода с ключевым словом `synchronized`?
7. Как можно приостанавливать выполнение потока?

5 Лямбда-выражения

Среди новшеств, которые были привнесены в язык Java с выходом JDK 8, особенно стоит выделить *лямбда-выражения* (*lambda expression*).

Java задумывалась как объектно-ориентированный язык в 1990-е гг., когда объектно-ориентированное программирование было главной парадигмой в разработке приложений. Задолго до этого были и функциональные языки программирования, такие как *Lisp* и *Scheme*, но их преимущества не были оценены за пределами академической среды. В последнее время значимость функционального программирования сильно выросла, потому что оно хорошо подходит для параллельного программирования и программирования, основанного на событиях. Так и появилась стратегия – смешать объектно-ориентированное и функциональное программирование.

Туториал по использованию лямбда-выражений представлен на официальном сайте Oracle [13].

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы. Основу лямбда-выражения составляет *лямбда-оператор*, который представляет собой стрелку `->`. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение имеет три компонента:

1. Блок кода.
2. Параметры.
3. Значения для свободных переменных, т. е. переменных, которые не являются параметрами и не определены в коде.

Попробуем записать «Hello world!» на лямбдах:

```
() -> System.out.println("Hello world!");
```

Результат представлен на рисунке 5.1.

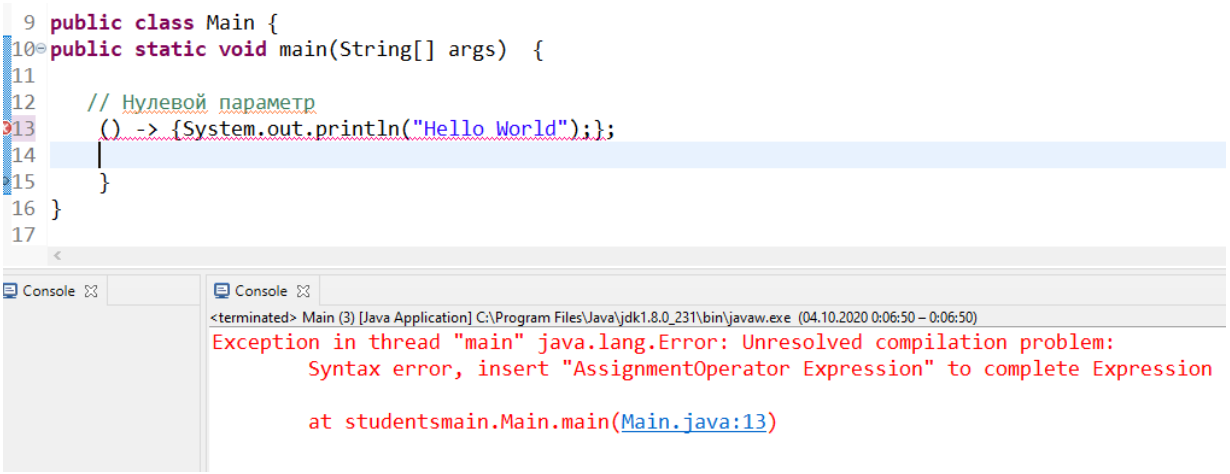


Рис. 5.1 – Ошибка компиляции: тип этого выражения должен быть функциональным интерфейсом

Мы ничего не принимаем, ничего не возвращаем (мы не можем поставить `return` перед вызовом `System.out.println()`, т. к. тип возвращаемого значения в методе `println()` – `void`), мы просто хотим вывести на консоль надпись. И проблема в том, что, как указано выше, мы ничего не делаем с лямбдой. Это значит, что компилятор не знает, какой функциональный интерфейс вывести в качестве типа лямбда-выражения.

Таким образом, лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе.

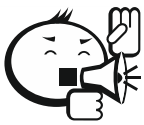
Функциональный интерфейс (*Functional Interface*) *в Java – интерфейс, в котором объявлен только один абстрактный метод.*

Лямбда-выражения *– это способ визуализации функционального программирования в объектно-ориентированном мире Java.*

Объекты являются основой языка программирования Java, и у нас никогда не может быть методов без объекта, поэтому язык Java обеспечивает поддержку использования *лямбда-выражений только с функциональными интерфейсами*.

Поскольку в функциональных интерфейсах есть только *один-единственный* абстрактный метод, не возникает путаницы в применении лямбда-выражения к методу. Синтаксис лямбда-выражений:

```
(аргумент) -> (тело);
```



Как и в случае с блоками `if-else`, мы можем избежать фигурных скобок ({}), поскольку у нас есть единственный оператор в

теле метода. Для нескольких операторов мы должны использовать фигурные скобки, как и любые другие методы.

Важно: многострочные лямбда-выражения всегда должны иметь оператор `return`, в отличие от однострочных.

. . . . . .

Перепишем наш пример.

```
@FunctionalInterface
interface HelloService {
    void hello();

  }

public class Main {
public static void main(String[] args)  {
   // Нулевой параметр
      HelloService str=() -> System.out.println("Hello World");
      //вызов метода
      str.hello();
      }
      }
```

Поскольку интерфейсные методы, не определяемые по умолчанию, неявно считаются *абстрактными*, их не обязательно объявлять с модификатором доступа *abstract*, хотя это и можно сделать при желании, ошибки в этом не будет.

Теперь мы увидим заветные слова (рис. 5.2).

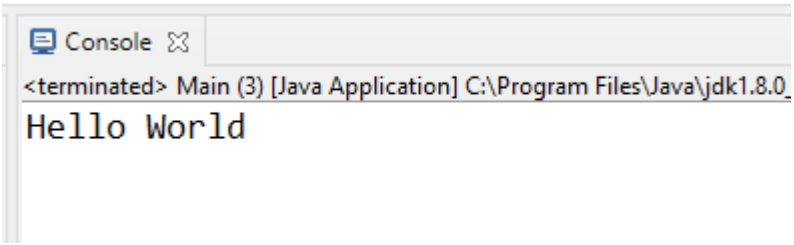


Рис. 5.2 – Вывод «Hello World» на лямбдах

Давайте посмотрим, как мы можем записать интерфейс `Runnable`, используя лямбда-выражение.

```
Runnable r1 = () -> System.out.println("My Runnable");
```

`java.lang.Runnable` – это пример функционального интерфейса, поэтому мы можем использовать лямбда-выражение для создания его экземпляра.

Поскольку метод `run()` не принимает аргументов, лямбда-выражение также не имеет аргументов. Подобное преобразование всегда осуществляется неявно, когда мы не указываем функциональный интерфейс.

Поток `Thread` можно проинициализировать двумя способами:

```
public class Main {
public static void main(String[] args)  {

            // Старый способ:
            new Thread(new Runnable() {
                @Override
                public void run() {
System.out.println("Runnable with Anonymous Class");
                }
            }).start();

            // Новый способ:
            new Thread(
() -> System.out.println("Runnable with Lambda Expression")
            ).start();
      }
      }
```

Результат представлен на рисунке 5.3.

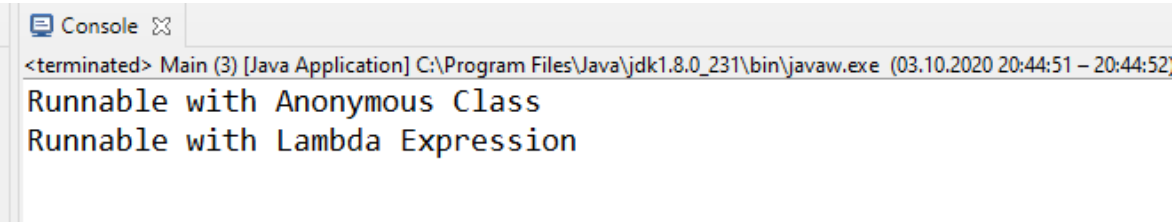
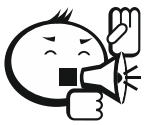


Рис. 5.3 – Реализация потока с помощью лямбда-выражений



Лямбда-выражение в Java – это, по сути, анонимный (т. е. неименованный) метод. Однако сам этот метод никогда не выполняется. Он лишь позволяет назначить реализацию кода метода, определяемого функциональным интерфейсом. Таким образом, лямбда-выражение представляет собой некую форму анонимного класса.

Зачастую определять собственный функциональный интерфейс не нужно, поскольку начиная с версии 8 внедрен пакет `java.util.function`, предо-

ставляющий несколько предопределенных функциональных интерфейсов. Вот несколько основных из них:

- Функциональный интерфейс `Predicate<T>` проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение `true`.
- `Consumer<T>` выполняет некоторое действие над объектом типа `T`, при этом ничего не возвращая.
- Функциональный интерфейс `Function<T,R>` представляет функцию перехода от объекта типа `T` к объекту типа `R:`.
- `Supplier<T>` не принимает никаких аргументов, но должен возвращать объект типа `T:`.
- `UnaryOperator<T>` принимает в качестве параметра объект типа `T`, выполняет над ними операции и возвращает результат операций в виде объекта типа `T:`.
- `BinaryOperator<T>` принимает в качестве параметра два объекта типа `T`, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа `T`.

Можно создать свой функциональный интерфейс. `@FunctionalInterface` используется, чтобы гарантировать, что функциональный интерфейс не может иметь более одного абстрактного метода.

```
@FunctionalInterface
interface FuncInterface
{
    // абстрактный метод
    void abstractMethod(int x);

    //  default-метод с реализацией
    default void normalMethod()
    {
       System.out.println("Hello");
    }
}



public class Main {
public static void main(String[] args)  {

      // лямбда-выражение для реализации выше
```

```
 // функционального интерфейса
FuncInterface fobj;
fobj= (int x)->System.out.println(x*x);

// вызываем указанное выше лямбда-выражение и выводим результат
fobj.abstractMethod(5);
//вызываем неабстрактный метод – default-метод с реализацией
fobj.normalMethod();

 }
 }
```

Результат выполнения программы показан на рисунке 5.4.

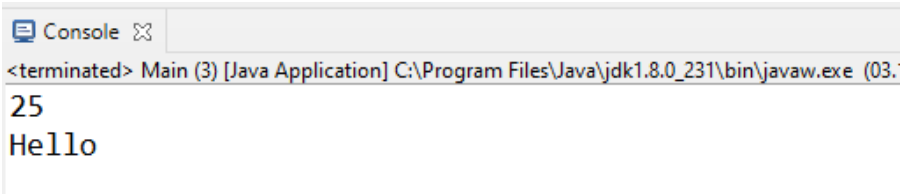


Рис. 5.4 – Пример реализации собственного функционального интерфейса

Если мы попытаемся добавить в интерфейс еще один абстрактный метод, то увидим ошибку (рис. 5.5), которая указывает на то, что интерфейс стал общим, а не функциональным, и ссылка должна быть сделана только из функционального интерфейса.

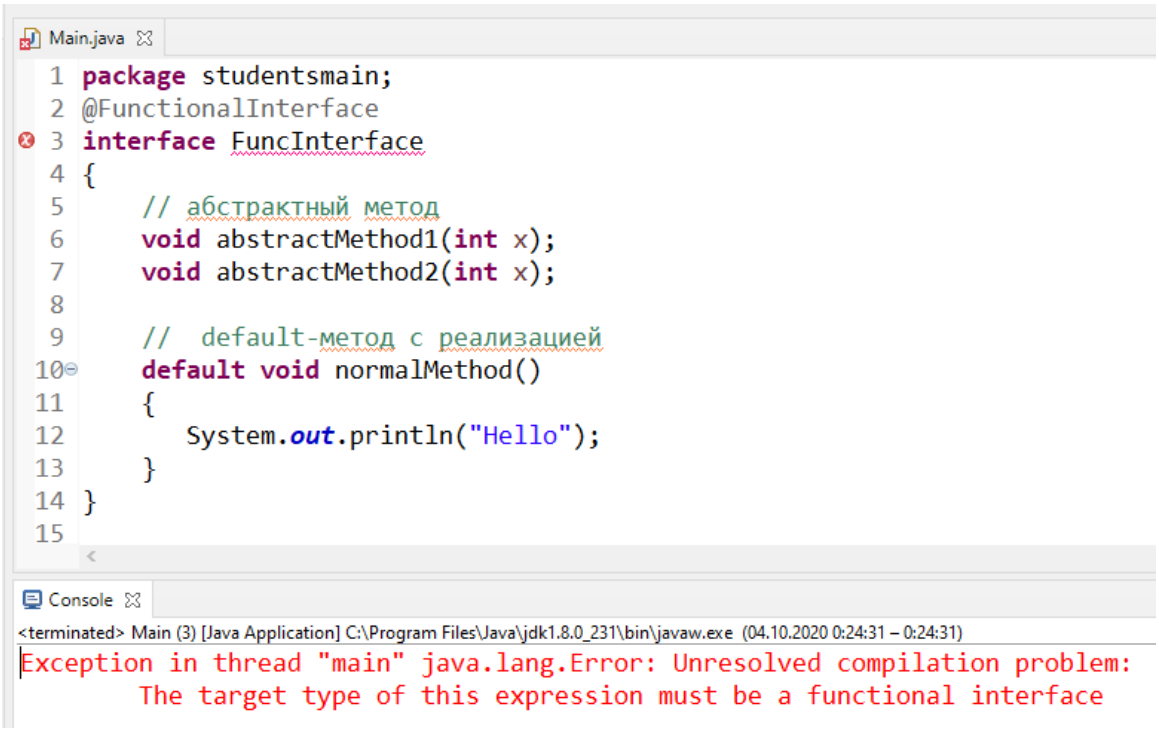


Рис. 5.5 – Ошибка несоответствия типа интерфейса

Функциональный интерфейс может быть *обобщенным*, однако в лямбда-выражении использование обобщений (дженериков) не допускается. В этом случае надо типизировать объект интерфейса определенным типом, который потом будет применяться в лямбда-выражении.

Пример

```
@FunctionalInterface

interface FuncInterface<T>{
    T calculate(T x, T y);
}


public class Main {
public static void main(String[] args)  {

     FuncInterface<Integer> o1 = (x, y)-> x + y;
     FuncInterface<String> o2 = (x, y) -> x + y;

    System.out.println(o1.calculate(1, 2)); //получим сумму = 2
    System.out.println(o2.calculate("1", "2")); //получим конкантенацию
строк 12
     }
}
```

В результате в первом случае получим сумму чисел, а во втором – конкатенацию строк (рис. 5.6).

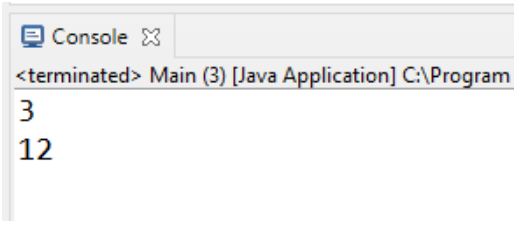


Рис. 5.6 – Пример использования обобщенного интерфейса

Таким образом, при объявлении лямбда-выражения объекту интерфейса уже известно, какой тип параметры будут представлять и какой тип они будут возвращать. То есть внутри класса `Main`:

- `FuncInterface <String> o1` – создает ссылку на интерфейс, который работает со `String`.

- `FuncInterface <Integer> o2` – создает ссылку на интерфейс, который работает с `Integer`.

Лямбда-выражения можно передавать и в качестве аргумента.

Для передачи лямбда-выражения в качестве аргумента параметр, получающий это выражение в качестве аргумента, должен иметь тип функционального интерфейса, совместимого с этим лямбда-выражением. В следующем примере попробуем передать лямбда-выражение в качестве аргумента методу:

```
@FunctionalInterface
interface FuncInterface{
      String func (String n);
}


public class Main {
      //Первый параметр этого метода имеет тип функционального
      // интерфейса, а второй параметр
      // обозначает обрабатываемую символьную строку
      static String stringUP (FuncInterface sf, String str) {
            return sf.func(str);
      }

public static void main(String[] args)  {

      String inStr = "тусур – чемпион!";
      String outStr;
      System.out.println( "Это исходная строка: "+ inStr);
      // Ниже приведено простое лямбда-выражение, преобразующее
      // в верхний регистр все символы исходной строки,
      // передаваемой методу stringUP ()
      outStr = stringUP ((str) -> str. toUpperCase(), inStr) ;
      System.out. println( "Это строка в верхнем регистре : "+ outStr) ;
      }
      }
```

Результат представлен на рисунке 5.7.

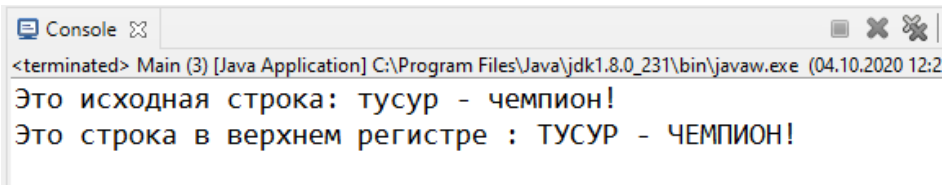


Рис. 5.7 – Передача лямбда-выражения в качестве аргумента

Очень удобно использовать лямбды для коллекций. Приведем простой пример вывода всех элементов заданного массива.

```
import java.util.Arrays;
import java.util.List;
public class Main {
public static void main(String[] args)  {

	// Старый способ:
	List<Integer> list1 = Arrays.asList(1, 2, 3);
	for(Integer n: list1) {
	    System.out.println(n);
	}

	// Новый способ:
	List<Integer> list2 = Arrays.asList(11, 22, 33);
	list2.forEach(n -> System.out.println(n));

	// Новый способ с использованием оператора ::
	List<Integer> list3 = Arrays.asList(111, 222, 333);
	list3.forEach(System.out::println);

	}
	}
```

Результат выполнения представлен на рисунке 5.8.

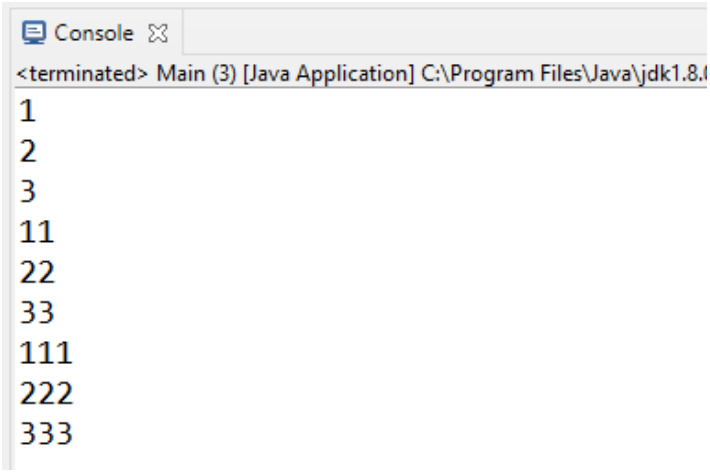


Рис. 5.8 – Перебор элементов коллекции с помощью лямбда-выражения

Далее рассмотрим пример использования функционального интерфейса `Predicate` для фильтрации списка. `java.util.function.Predicate` декларирует абстрактный метод `test()`, который принимает объект и возвращает значение типа `boolean` в зависимости от соответствия переданного объекта

требуемым критериям. Также используем метод `startsWith()`, который возвращает значение `true`, если последовательность символов представленного аргумента является префиксом последовательности символов, представляемой данной строкой; в противном случае значение `false`.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Main {
     public static void main(String[] args)  {

        // создадим список строк
    List<String> names =
        Arrays.asList("Иванов","Петров","Сидоров","Исаев");

    // объявим тип предикату как строку и используем
    // лямбда-выражение для создания объекта.
    //Найдем строки, которые начинаются с заданного символа
    Predicate<String> p = (s)->s.startsWith("И");

    // перебор элементов коллекции
    for (String st:names)
    {
        // вызов метода test() из Predicate
        if (p.test(st))
            System.out.println(st);
    }

}
}
```

Результат представлен на рисунке 5.9.

Рис. 5.9 – Пример использования лямбда-выражений и предиката для фильтрации коллекций

Вернем к нашему классу `Student`.

```
public class Student {

    private int age;
    private String name;

        public Student(String name, int age) {
        this.name=name;
        this.age=age;
        }

        public Student() {
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name=name;
        }

        public void setAge(int age) {
            this.age=age;
        }

         public int getAge() {
            return age;
        }
            @Override
            public String toString() {
                return "студент "+name+", "+age+"лет.";
        }
     }
```

Создадим в главном классе `Main` список студентов. Используем лямбда-выражения и предикат, чтобы получить список студентов, которым 21 год и больше.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class Main {

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

```
        List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 20));
    sList.add(new Student("Петров", 21));
    sList.add(new Student("Сидоров", 19));
    sList.add(new Student("Исаев", 35));

    print(sList, p -> p.getAge() >= 21);
}

private static void print(List<Student> sList, Predicate<Student>
checker) {
    for (Student student : sList) {
        if (checker.test(student)) {
            System.out.println(student + " >= 21.");
        } else {
            System.out.println(student  + " < 21.");
        }

    }

}
     }
```

`print(sList, p -> p.getAge() >= 21);` метод принимает лямбда-выражение (потому что в `Predicate` используется параметр), где можно определить требуемое выражение;

`checker.test(student);` метод проверки проверяет правильность этого выражения.

Результат представлен на рисунке 5.10.

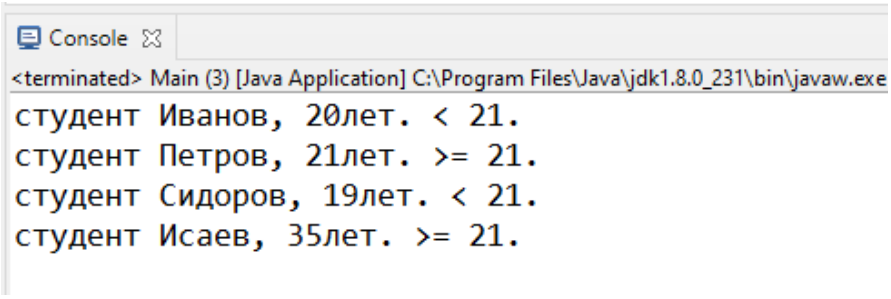


Рис. 5.10 – Пример применения лямбда-выражения и предиката

Рассмотрим сортировку коллекций.

Служебный класс `Collections` существовал в Java еще со времен версии 1.2, когда в язык были добавлены коллекции. Статический метод `sort()`

класса Collections принимает в качестве аргумента List и возвращает void. В процессе сортировки коллекция модифицируется. Этот подход не согласуется с функциональными принципами, провозглашенными в Java 8 и приветствующими неизменяемость.

Далее создадим список студентов и попытаемся отсортировать его с помощью Collections.sort:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {

public static void main(String[] args)  {

     List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 20));
    sList.add(new Student("Петров", 21));
    sList.add(new Student("Сидоров", 19));
    sList.add(new Student("Исаев", 35));

    System.out.println("Список перед сортировкой : " + sList);
    Collections.sort(sList);
    System.out.println("Список после сортировки : " + sList);
}
     }
```

Это приводит к ошибке во время компиляции:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:
The method sort(List<T>) in the type Collections is not applicable for
the arguments (List<Student>)
```

Collections.sort требует параметра типа для реализации java.util.Comparable.

Comparable – *интерфейс, определяющий стратегию сравнения объекта с другими объектами того же типа. Это называется «естественным сравнением, natural comparison method» класса.*

Классы-обертки Byte, Short, Integer, Long, Double, Float, Character, String уже реализуют интерфейс Comparable.

Соответственно, чтобы иметь возможность сортировки, мы должны определить наш объект `Student` как сопоставимый путем реализации интерфейса `Comparable`:

```
@Override
            public int compareTo(Student student) {
                return (this.getAge() - student.getAge());
    }
```

Порядок сортировки определяется значением, возвращаемым методом `compareTo()`.

Метод возвращает число, указывающее, является ли сравниваемый объект меньше, равным или больше объекта, передаваемого в качестве аргумента. Он возвращает:

- ноль, если два объекта равны;
- число > 0, если первый объект (на котором вызывается метод) больше, чем второй (который передается в качестве параметра);
- число < 0, если первый объект меньше второго.

Изменим наш класс `Student`:

```
    //реализуем в классе интерфейс Comparable
public class Student implements Comparable<Student> {

    private int age;
    private String name;

        public Student(String name, int age) {
        this.name=name;
        this.age=age;
        }

        public Student() {
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name=name;
        }
```

```
        public void setAge(int age) {
            this.age=age;
        }

         public int getAge() {
            return age;
        }
            @Override
            public String toString() {
                return "\nстудент "+name+", "+age+"лет.";
        }
//переопределим метод compareTo, будем сортировать по возрасту
            @Override
            public int compareTo(Student student) {
                return (this.getAge() - student.getAge());
            }
    }
```

В результате в классе `Main` ошибка компиляции исчезнет и можно будет запустить сортировку. В результате список будет отсортирован по возрасту (рис. 5.11).

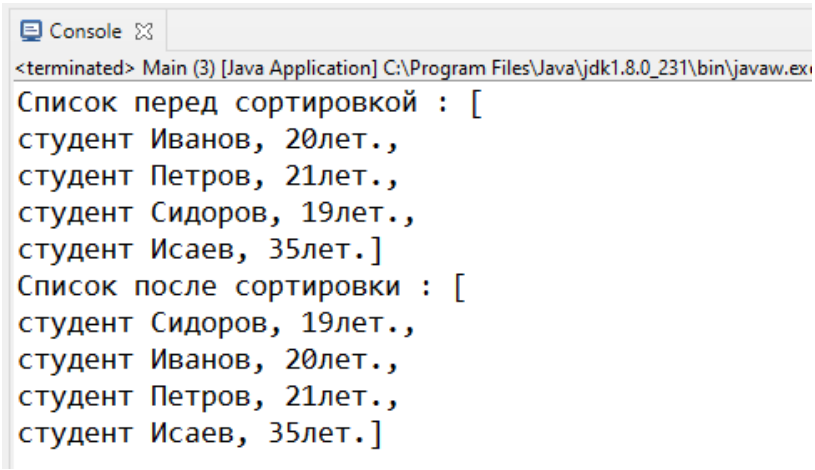


Рис. 5.11 – Пример сортировки коллекций с помощью `Comparable`

```
//переопределим метод compareTo, будем сортировать по фамилии
            @Override
            public int compareTo(Student student) {
                return name.compareTo(student.getName());
                 }
```

В результате список будет отсортирован по фамилии (рис. 5.12).

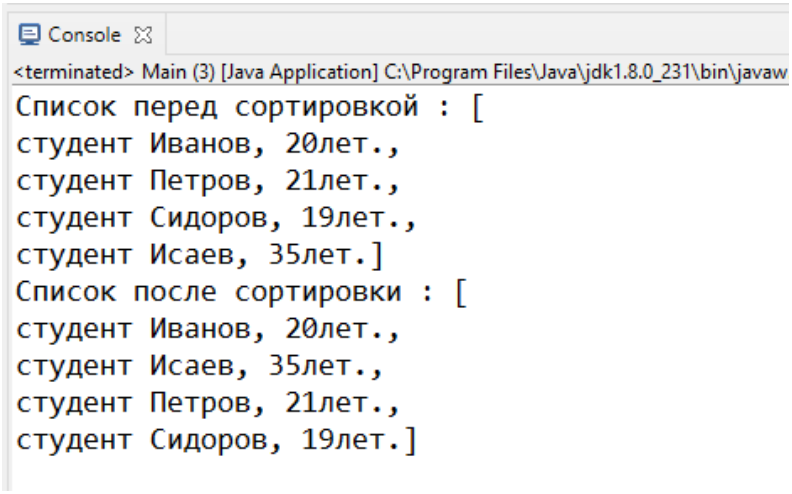


Рис. 5.12 – Пример сортировки коллекций с помощью метода `compareTo()`

Также можем сравнивать по длине фамилии:

```
//переопределим метод compareTo(), будем сортировать по длине фамилии
            @Override
            public int compareTo(Student student) {
                return name.length()-student.getName().length();
                 }
```

В результате список будет отсортирован по длине фамилии (рис. 5.13).

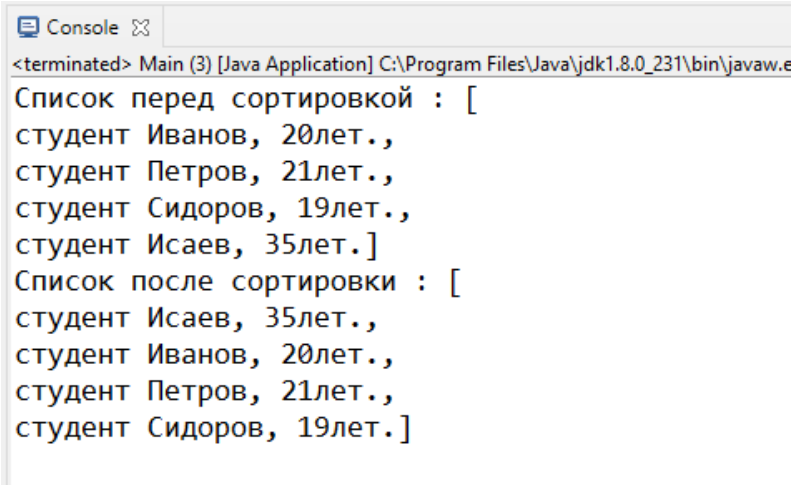


Рис. 5.13 – Пример сортировки коллекций
с помощью метода `compareTo()` с дополнительными параметрами

В Java 8 для такой же сортировки к потоку применяется метод `sorted()`, но при этом порождается новый поток, а не модифицируется исходная коллекция. Для строк естественным является лексикографический порядок, который сводится к алфавитному в случае, когда все строки состоят из символов в нижнем регистре. Чтобы отсортировать строки, следует воспользоваться перегруженным

вариантом метода `sorted()`, который принимает в качестве аргумента объект типа `Comparator`.

Интерфейс `Comparable` используется только для сравнения объектов класса, в котором данный интерфейс реализован. `Comparator` представляет отдельную реализацию, ее можно использовать многократно и с различными классами.

Интерфейс `Comparator` содержит ряд методов, ключевым из которых является метод `compare(Object obj1, Object obj2)`, который позволяет сравнивать между собой два объекта. На выходе метод возвращает значение 0, если объекты равны, положительное значение или отрицательное значение, если объекты не тождественны.

Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:

```
import java.util.Comparator;
//класс компаратор
class StudentComparator implements Comparator<Student>{

    public int compare(Student a, Student b){

        return a.getName().compareTo(b.getName());
    }
     }
```

Теперь используем класс компаратора для создания объекта:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {

public static void main(String[] args)  {
     //создаем объект компаратора
     StudentComparator scomp = new StudentComparator();
     List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 20));
    sList.add(new Student("Петров", 21));
    sList.add(new Student("Сидоров", 19));
```

```
    sList.add(new Student("Исаев", 35));

    System.out.println("Список перед сортировкой : " + sList);
 // Используем объект StudentComparator для сортировки элементов
    Collections.sort(sList,scomp);
    System.out.println("Список после сортировки : " + sList);
}
     }
```

В результате наш список будет отсортирован по фамилии (рис. 5.14).

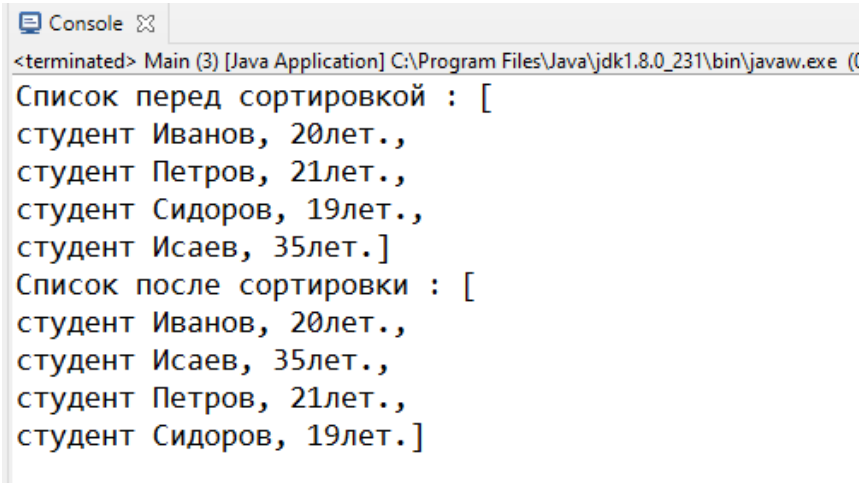


Рис. 5.14 – Пример сортировки коллекций с помощью метода `sort()`

Лямбда-выражения помогают писать реализацию `Comparator` «на лету». Нам не нужно создавать отдельный класс для обеспечения логики, если требуется использовать разово в одном месте. Кстати, компаратор в фрагменте кода может быть сделан даже более кратким, если вместе с лямбда-выражениями использовать метод построения компаратора `comparing()`.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Main {

public static void main(String[] args)  {

     List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 20));
    sList.add(new Student("Петров", 21));
    sList.add(new Student("Сидоров", 19));
    sList.add(new Student("Исаев", 35));

  //Сортируем по возрасту
```

```
    sList.sort(Comparator.comparing(e -> e.getAge()));

    System.out.println(sList);

    //или сортируем по фамилии
    sList.sort(Comparator.comparing(Student::getName));

    System.out.println(sList);
}
     }
```

Уже не требуется отдельный класс-компаратор. В результате можно сразу сортировать по разным критериям (рис. 5.15).

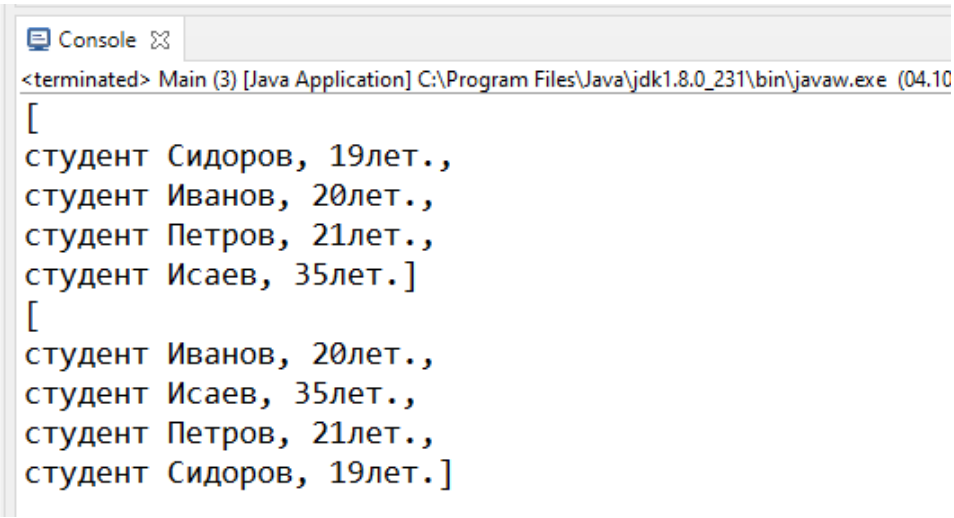


Рис. 5.15 – Пример сортировки коллекций с помощью `Comparator`

Если мы хотим отсортировать по имени, но в обратном порядке, следует использовать `reversed()`.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {

public static void main(String[] args)  {

     List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 20));
    sList.add(new Student("Петров", 21));
    sList.add(new Student("Сидоров", 19));
    sList.add(new Student("Исаев", 35));
```

```
    Comparator<Student> comparator = Comparator.comparing(e -> e.get-
Name());
    sList.sort(comparator.reversed());
    System.out.println(sList);
}
}
```

В результате наш список будет отсортирован по фамилии в обратном порядке (рис. 5.16).

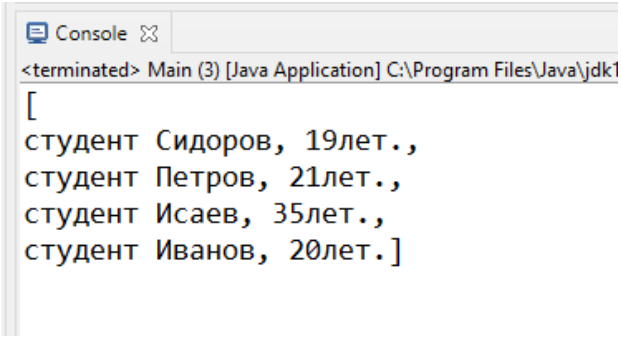


Рис. 5.16 – Пример обратной сортировки коллекций с помощью `Comparator`

Лямбда-выражения сравнения не обязательно должны быть такими простыми – мы также можем писать более сложные выражения. Например, если при сортировке объектов по фамилии встречаются одинаковые фамилии, сортируем по возрасту:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {

public static void main(String[] args)  {

     List<Student> sList = new ArrayList<Student>();
    sList.add(new Student("Иванов", 25));
    sList.add(new Student("Иванов", 21));
    sList.add(new Student("Сидоров", 19));
    sList.add(new Student("Исаев", 35));

    // сортируем по фамилии
    sList.sort(Comparator.comparing(Student::getName));
    System.out.println(sList);
```

```
    // сортируем по фамилии, если они равны, то сортируем по возрасту

    sList.sort((s, t) -> {
        if (s.getName().equals(t.getName())) {
            return Integer.compare(s.getAge(), t.getAge());
        } else {
            return s.getName().compareTo(t.getName());
        }
    });

    System.out.println(sList);
}
     }
```

В результате видим два списка (рис. 5.17), они не одинаковые.

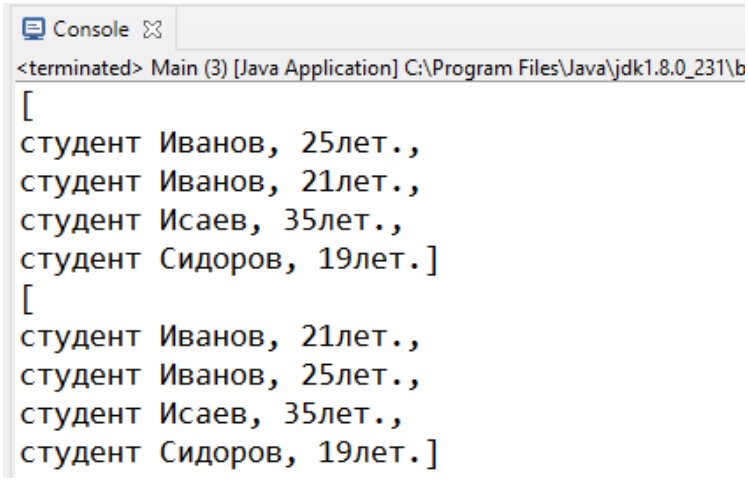


Рис. 5.17 – Пример сортировки коллекций по разным параметрам

Если сортировка объектов должна быть основана на естественном порядке, используйте `Comparable`, тогда как если вам нужно выполнить сортировку по атрибутам разных объектов, используйте `Comparator`.

Выводы

1. Добавление лямбда-выражений в язык делает практичным использование функциональных объектов, в которых ранее это не имело бы смысла.
2. Сокращение строк кода. Одно из очевидных преимуществ использования лямбда-выражения состоит в том, что объем кода сокращается.

3. Поддержка последовательного и параллельного выполнения. Еще одно преимущество использования лямбда-выражения заключается в том, что мы можем извлечь выгоду из поддержки последовательных и параллельных операций Stream API.

Контрольные вопросы по главе 5

1. Что такое функциональный интерфейс? Каковы правила определения функционального интерфейса?
2. Опишите некоторые из функциональных интерфейсов в стандартной библиотеке.
3. Какова структура лямбда-выражений?
4. Можно ли объявить в функциональном интерфейсе несколько абстрактных методов?
5. Чем отличаются интерфейсы `Comparable` и `Comparator`?

Заключение

Механизмы универсальности, впервые появившиеся в версии Java 1.5, по-прежнему остаются с нами, но в Java 8 сигнатуры универсальных методов стали гораздо сложнее. В большинстве функциональных интерфейсов, добавленных в язык, используются как универсальные типы, так и ограниченные метатипы – во имя типобезопасности. Несмотря на то что 15 сентября 2020 г. вышла версия 15, многие до сих пор используют версию 8.

Существует множество разных причин, по которым компании все еще пользуются Java 8. Вот некоторые из них:

1. Боязнь обновлений у программистов и тестировщиков.
2. Инструменты сборки (Maven, Gradle и т. д.) и некоторые библиотеки изначально имели ошибки с версиями Java позже 8 и нуждались в обновлениях.
3. До Java 8 в основном использовали JDK-сборки Oracle, и не нужно было заботиться о лицензировании. Однако в 2019 г. Oracle изменила схему лицензирования, что привело к тому, что «Java больше не является бесплатной».

Из новшеств в версии 15 стоит выделить новый тип классов, называемых скрытыми. На скрытые классы не могут прямо ссылаться другие классы, и все их использование может осуществляться только через рефлексию. Также их нельзя обнаружить по имени, и их методы не появляются в стек-трейсах. Создаются такие классы с помощью нового метода `Lookup.defineHiddenClass()` [14].

Каждому специалисту необходимо развиваться каждый день. Как говорят корифеи, «первая вещь, которую должен выучить программист, – это то, что его обучение никогда не закончится».

Литература

1. Вайсфельд, М. Объектно-ориентированное мышление / М. Вайсфельд. – СПб. : Питер, 2014. – 304 с.
2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Гради Буч. – 3-е изд. – М. : ООО «ИД «Вильямс», 2008. – 720 с.
3. Гуськова, О. И. Объектно-ориентированное программирование в Java : учеб. пособие / О. И. Гуськова. – М. : МПГУ, 2018. – 240 с.
4. Мейер, Б. Почувствуй класс / Б. Мейер ; пер. с англ. под ред. В. А. Бил-лига. – М. : Национальный открытый университет «ИНТУИТ» : БИ-НОМ. Лаборатория знаний, 2011. – 775 с.
5. Дубаков, А. А. Введение в объектно-ориентированное программирование на Java : учеб. пособие / А. А. Дубаков. – СПб. : Университет ИТМО, 2016. – 250 с.
6. Java. Промышленное программирование : практ. пособие / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.
7. Коузен, К. Современный Java: рецепты программирования / К. Коузен ; пер. с англ. А. А. Слинкина. – М. : ДМК Пресс, 2018. – 274 с.
8. Lesson: Generics (Updated) [Electronic resource] // oracle.com : site. – URL: https://docs.oracle.com/javase/tutorial/java/generics/index.html
9. Шилдт, Г. Java. Полное руководство / Г. Шилдт ; пер. с англ. – 10-е изд. –СПб. : ООО «Альфакнига», 2018. – 1488 с.
10. Interface Serializable [Electronic resource] // oracle.com : site. – URL: https://docs.oracle.com/javase/1.5.0/docs/api/java/io/Serializable.html
11. Interface Lock [Electronic resource] // oracle.com : site. – URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html .
12. Lambda Expressions [Electronic resource] // oracle.com : site. – URL: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html .

14. JDK 15 Documentation [Electronic resource] // oracle.com : site. – URL: https://docs.oracle.com/en/java/javase/15/ .

Глоссарий

Абстракция (*abstraction*) выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов, и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрактный класс – класс, экземпляр которого нельзя создать. Иначе: класс, который содержит хотя бы один абстрактный метод, называется *абстрактным*.

Абстрактный метод (*abstract method*) – метод, реализация которого неизвестна на данный момент.

Агрегация (*aggregation*), или *включение*, – отношение между классами типа «содержит» или «состоит из».

Анонимные классы (*anonymous class*) – классы, которые не имеют имени. Их создание происходит в момент инициализации объекта.

Анонимный объект (*anonymous object*) – объект, на который нет ссылки.

Апкастинг (*upcasting*) (приведение к базовому типу) – замена объекта суперкласса объектом подкласса.

Ассоциация (*association*) – отношение, когда объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону связь между объектами не носит характера «владения» или контейнеризации.

Даункастинг (*downcasting*) возвращает замещенный объект к определению через подкласс, т. е. это приведение объекта суперкласса к объекту подкласса.

Десериализация (*Deserialization*) – процесс извлечения объекта из потока байтов.

Дженерики (*обобщения*) – особые средства языка Java для реализации обобщенного программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

Идентичность (*names*) – такое свойство объекта, которое отличает его от всех других объектов.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях.

Инкапсуляция (*encapsulation*) – сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса).

Интерфейс (*interface*) – явно указанная спецификация набора абстрактных методов, которые должны быть представлены в классе, реализующем эту спецификацию.

Исключениями или *исключительными ситуациями* (*exception*) называются ошибки, возникшие в программе во время ее работы.

Коллекциями (*collection*) называют структуры, предназначенные для хранения однотипных данных.

Конструктор (*constructor*) – особенный метод класса, который вызывается автоматически в момент создания объектов этого класса. Имя конструктора совпадает с именем класса.

Компаратор (*Comparable*) – интерфейс, определяющий стратегию сравнения объекта с другими объектами того же типа. Это называется «естественным сравнением, natural comparison method» класса.

Композиция (*Composition*) – разновидность жесткой взаимосвязи между объектами, составляющими класс. Когда объект уничтожается, объекты, составляющие его, также уничтожаются.

Класс (*class*) – шаблон, или прототип, по которому создаются объекты.

Лямбда-выражения – способ визуализации функционального программирования в объектно-ориентированном мире Java.

Метод (*method*) – последовательность команд, которые вызываются по определенному имени.

Модификатор (*modifier*) – ключевое слово языка, которое может каким-либо образом изменить смысл некоторого определения (например, класса или метода).

Монитор (*monitor*) – специальный объект, который следит за «состоянием» метода или объекта. Он смотрит, «занят» ресурс или «свободен» в данный момент.

Многопоточность в Java – выполнение двух или более потоков одновременно для максимального использования центрального процесса.

Множественным наследованием называется ситуация, когда класс наследует от двух или более классов.

Модель (*model*) – абстракция физической системы, рассматриваемая с определенной точки зрения и представленная на некотором языке или в графической форме.

Наследование – свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником, дочерним или производным классом.

Наследование (*inheritance*) – отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов.

Обобщенные типы – механизм компилятора, посредством которого можно некоторым стандартным образом создавать (и использовать) типы (классов, интерфейсов и т. п.), получая единый код и параметризуя (или обобщая) все остальное.

Обработка исключительных ситуаций (*exception handling*) – механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (исключения), которые могут возникнуть при выполнении программы.

Объект (*object*) – сущность, обладающая определенным поведением и способом представления.

Объектно-ориентированное программирование (*object-oriented programming*) – технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств.

Перегрузка методов (*method overloading*) – различные реализации методов с одинаковыми именами, но разными сигнатурами в Java.

Переопределение метода (*method overriding*) – изменение работы метода, унаследованного от класса-предка классом-потомком, путем описания нового метода с точно такими же именем и параметрами.

Поведение (*behaviors*) – то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Подстановочный знак (*wildcards ?*) в Java есть специальный параметр типа, который контролирует безопасность типов при использовании универсальных типов.

Поле (*атрибут, переменные*) *класса* – характеристика объекта, описывающая его свойство.

Поток данных (*stream*) представляет собой абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

Полиморфизм (*polymorphism*) – положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов.

Приоритет потоков – некоторое число в объекте потока, более высокое значение которого означает больший приоритет.

Сериализация (*Serializable*) – процесс, когда состояние объекта и его метаданные (например, имя класса объекта и имена его атрибутов) сохраняются в последовательность байтов, по которой затем его можно полностью восстановить.

Состояние (*attributes*) объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Сигнатура (s*ignature*) определяется именем метода и его аргументами (количеством, типом, порядком следования).

Сырой тип (*raw type*) – имя обобщенного класса или интерфейса без аргументов типа.

Универсальный класс объявляет одну или несколько переменных типа.

Функциональный интерфейс (*Functional Interface*) в Java – интерфейс, в котором объявлен только один абстрактный метод.

Экземпляр класса (*instance*) – отдельная реализация класса, все экземпляры класса имеют одинаковые свойства, которые описаны в определении класса.

Synchronized – ключевое слово, которое позволяет заблокировать доступ к методу или части кода, если его уже использует другой поток.

UML (Unified Modeling Language – унифицированный язык моделирования) – язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.