Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект №1
Java Core.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

1 Предисловие

Так как у слушателей уже есть базовые знания языка Java первым разделом данного курса будет повтор основных важных тем в Java Core. Также мы затронем темы, которые непосредственно необходимы для выполнения практической работы.

2 Основные языковые конструкции

2.1 Тип данных boolean

Тип boolean представляет значения истинности. Существует только два возможных значения для данного типа, представляющих два булевых состояния: включено или выключено, да или нет, истина или ложь. Java резервирует слова true и false для представления этих булевых значений они же являются и двумя допустимыми литералами для задания значений переменных типа boolean.

В Java существуют довольно строгие ограничения по отношению к типу boolean: значения типа boolean **нельзя преобразовать ни в какой другой тип** данных, и наоборот. В частности, boolean не является целым типом, а целые значения нельзя применять вместо булевых.

2.2 Константы

В языке Java для обозначения констант служит ключевое слово **final**.

Ключевое слово **final** означает, что присвоить данной переменной какое-нибудь значение можно лишь один раз, после чего изменить его уже нельзя. Использовать в именах констант только прописные буквы необязательно, но такой стиль способствует удобочитаемости кода.

В программах на Java часто возникает потребность в константах, доступных нескольким методам в одном классе. Обычно они называются константами класса. Константы класса объявляются с помощью ключевых слов **static final**.

2.3 Символьные строки

По существу, символьная строка Java представляет собой последовательность символов в Юникоде. В языке Java отсутствует встроенный тип для символьных строк. Вместо этого в стандартной библиотеке Java содержится класс String. Каждая символьная строка, заключенная в кавычки, представляет собой экземпляр класса String:

```
String e = ""; // пустая строка
String greeting = "Hello";
```

2.3.1 Принцип постоянства символьных строк

В классе String отсутствуют методы, которые позволяли бы изменять символы в существующей строке. Так, если требуется заменить символьную строку в переменной greeting с "Hello” на "Help!", этого нельзя добиться одной

лишь заменой двух последних символов. Программирующим на C это покажется, по меньшей мере, странным. "Как же видоизменить строку?" — спросят они. В языке Java можно внести необходимые изменения в строку, выполнив сцепление подстроки, которую требуется сохранить, с заменяющими символами, как показано ниже. В итоге переменной greeting присваивается символьная строка "Help!".

```
greeting = greeting.substring(0, 3) + "p!";
```

Программируя на Java, нельзя изменять отдельные символы в строке, поэтому в документации на этот язык объекты типа String называются неизменяемыми, т.е. постоянными. Как число 3 всегда равно 3, так и строка "Hello" всегда состоит из последовательности кодовых единиц символов 'H', 'e', 'l', 'l' и 'o'. Изменить ее состав нельзя. Но, как мы только что убедились, можно изменить содержимое строковой переменной greeting и заставить ее ссылаться на другую символьную строку подобно тому, как числовой переменной, в которой хранится число 3, можно присвоить число 4.

Не приводит ли это к снижению эффективности кода? Казалось бы, намного проще изменять символы, чем создавать новую строку заново. Возможно, это и так. В самом деле, неэффективно создавать новую строку путем сцепления символьных строк "Hel" и "p! ". Но у неизменяемых строк имеется одно существенное преимущество: компилятор может сделать строки совместно используемыми.

Чтобы стал понятнее принцип постоянства символьных строк, представьте, что в общем пуле находятся разные символьные строки. Строковые переменные указывают на объекты в этом пуле. При копировании строковой переменной оригинал и копия содержат одну и ту же общую последовательность символов. Одним словом, создатели языка Java решили, что эффективность совместного использования памяти перевешивает неэффективность редактирования строк путем выделения и сцепления подстрок.

2.3.2 Проверка символьных строк на равенство

Чтобы проверить две символьные строки на равенство, достаточно вызвать метод equals(). Так, выражение s.equals(t) возвращает логическое значение true, если символьные строки s и t равны, а иначе — логическое значение false. Следует, однако, иметь в виду, что в качестве s и t могут быть использованы строковые переменные или константы. Например, следующее выражение вполне допустимо:

```
"Hello!".equals(greeting);
```

А для того чтобы проверить идентичность строк, игнорируя отличия в прописных и строчных буквах, следует вызвать метод equalsIgnoreCase (), как показано ниже.

```
"Hello".equalsIgnoreCase("hello");
```

Для проверки символьных строк на равенство нельзя применять операцию ==. Она лишь определяет, хранятся ли обе строки в одной и той же области памяти. Разумеется, если обе строки хранятся в одном и том же месте, они должны совпадать. Но вполне возможна ситуация, когда одинаковые символьные строки хранятся в разных местах. Ниже приведен соответствующий пример.

```
String greeting = "Hello”; // инициализировать переменную greeting
                          // символьной строкой "Hello"
if (greeting = "Hello") ...
        // возможно, это условие истинно
if (greeting.substring(0, 3) == "Hel") ...
        // возможно, это условие ложно
```

Если виртуальная машина всегда обеспечивает совместное использование одинаковых символьных строк, то для проверки их равенства можно применять операцию ==. Но совместно использовать можно лишь константы, а не символьные строки, получающиеся в результате таких операций, как сцепление или извлечение подстроки методом substring(). Следовательно, лучше вообще отказаться от проверки символьных строк на равенство с помощью операции ==, чтобы исключить в программе наихудшую из возможных ошибок, проявляющуюся лишь время от времени и практически не предсказуемую.

2.3.2 Пустые и нулевые строки

Пустой считается символьная строка нулевой длины. Чтобы проверить, является ли символьная строка пустой, достаточно составить выражение вида

if (str. length () == 0) или if (str.equals ("")).

Пустая строка является в Java объектом, в котором хранится нулевая (т.е. 0) длина символьной строки и пустое содержимое. Но в переменной типа String может также храниться специальное пустое значение null, указывающее на то, что в настоящий момент ни один из объектов не связан с данной переменной. Чтобы проверить, является ли символьная строка нулевой, т.е. содержит значение null, служит условие if (str == null).

Иногда требуется проверить, не является ли символьная строка ни пустой, ни нулевой. Для этой цели служит условие if (str != null && str. length () != 0). Но сначала нужно проверить, не является ли символьная строка str нулевой.

2.4 Цикл в стиле for each

В языке Java имеется эффективная разновидность цикла, позволяющая перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик. Эта усовершенствованная разновидность цикла for записывается следующим образом:

for (переменная : коллекция) оператор

При выполнении этого цикла его переменной последовательно присваивается каждый элемент заданной коллекции, после чего выполняется

указанный оператор (или блок). В качестве коллекции может быть задан массив или экземпляр класса, реализующего интерфейс Iterable, например ArrayList.

3 Объекты и классы

3.1 Статические поля

Поле с модификатором доступа static существует в одном экземпляре для всего класса. Но если поле не статическое, то каждый объект содержит его копию. Допустим, требуется присвоить уникальный идентификационный номер каждому работнику. Для этого достаточно добавить в класс Employee поле id и статическое поле next Id, как показано ниже.

```
class Employee
{
        private int id;
        private static int nextld = 1;
}
```

Теперь у каждого объекта типа Employee имеется свое поле id, а также поле nextld, которое одновременно принадлежит всем экземплярам данного класса. Иными словами, если существует тысяча объектов типа Employee, то в них есть тысяча полей id: по одному на каждый объект. В то же время существует только один экземпляр статического поля nextld. Даже если не создано ни одного объекта типа Employee, статическое поле nextld все равно существует. Оно принадлежит классу, а не конкретному объекту.

Реализуем следующий простой метод:

```
public void setld()
{
        id = nextld;
        nextld++;
}
```

3.2 Статические константы

Статические переменные используются довольно редко. В то же время статические константы применяются намного чаще. Например, статическая константа, задающая число π, определяется в классе Math следующим образом:

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

Обратиться к этой константе в программе можно с помощью выражения Math. PI. Если бы ключевое слово static было пропущено, константа PI была бы обычным полем экземпляра класса Math. Это означает, что для доступа к такой константе нужно было бы создать объект типа Math, причем каждый такой объект имел бы свою копию константы PI.

3.3 Статические методы

Статическими называют методы, которые не оперируют объектами. Например, метод pow() из класса Math является статическим. При вызове метода Math.pow (x, a) вычисляется значение x в степени a. При выполнении этого метода не используется ни один из экземпляров класса Math. Иными словами, у него нет неявного параметра this. Это означает, что в статических методах не используется текущий объект по ссылке this.

Статическому методу из класса Employee недоступно поле экземпляра id, поскольку он не оперирует объектом. Но статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
        return nextId; // возвратить статическое поле
}
```

Чтобы вызвать этот метод нужно указать имя класса следующим образом: int n = Employee.getNextId(); Можно ли пропустить ключевое слово static при обращении к этому методу? Можно, но тогда для его вызова потребуется ссылка на объект типа Employee.

Статические методы следует применять в двух случаях.

• Когда методу не требуется доступ к данным о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе Math.pow()).

• Когда методу требуется доступ лишь к статическим полям класса (например, при вызове метода Employee.getNextId()).

3.4 Метод main()

Метод main() отличается от всех остальных методов тем, что является, как правило, точкой входа в программу. Этот метод вызывается виртуальной машиной Java. Как только заканчивается выполнение метода main(), так сразу же завершается, тем самым, работа самой программы.

Метод main(), так и любой другой метод, должен быть обязательно вложен в класс. После компиляции класс, содержащий метод main(), запускается на выполнение командой

java ПолноеИмяКласса

Отметим, что статические методы можно вызывать, не имея ни одного объекта данного класса. Например, для того чтобы вызвать метод Math.pow(), объекты типа Math не нужны. По той же причине метод main() объявляется как статический:

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

```
        // здесь создаются объекты
        }
}
```

Метод **main**() не оперирует никакими объектами. На самом деле при запуске программы еще нет никаких объектов. Статический метод main() выполняется и конструирует объекты, необходимые программе.

3.5 Параметры методов

Рассмотрим термины, которые употребляются для описания способа передачи параметров методам (или функциям) в языках программирования. Термин *вызов по значению* означает, что метод получает значение, переданное ему из вызывающей части программы. *Вызов по ссылке* означает, что метод получает из вызывающей части программы *местоположение* переменной. Таким образом, метод может *модифицировать* (т.е. видоизменить) значение переменной, передаваемой по ссылке, но не переменной, передаваемой по значению. Фраза "вызов по..." относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в Java. (На самом деле существует еще и третий способ передачи параметров — *вызов по имени*, представляющий в основном исторический интерес, поскольку он был применен в языке Algol, который относится к числу самых старых языков программирования высокого уровня.)

В языке Java всегда используется *только* вызов по значению. Это означает, что метод получает копии значений всех своих параметров. По этой причине метод не может видоизменить содержимое ни одной из переменных, передаваемых ему в качестве параметров.

Рассмотрим для примера следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Каким бы образом ни был реализован метод, после его вызова значение переменной percent все равно останется равным **10.**

Проанализируем эту ситуацию подробнее. Допустим, в методе предпринимается попытка утроить значение параметра, как показано ниже.

```
public static void tripleValue(double x); // не сработает!
{
        x = 3 * x;
}
```

Если вызвать этот метод следующим образом:

```
double percent = 10;
tripleValue(percent) ;
```

такой прием не сработает. После вызова метода значение переменной percent по-прежнему остается равным 10. В данном случае происходит следующее.

1. Переменная x инициализируется копией значения параметра percent (т.е. числом 10).
2. Значение переменной x утраивается, и теперь оно равно 30. Но значение переменной percent по-прежнему остается равным 10 (Рисунок 1).
3. Метод завершает свою работу, и его переменный параметр x больше не используется.

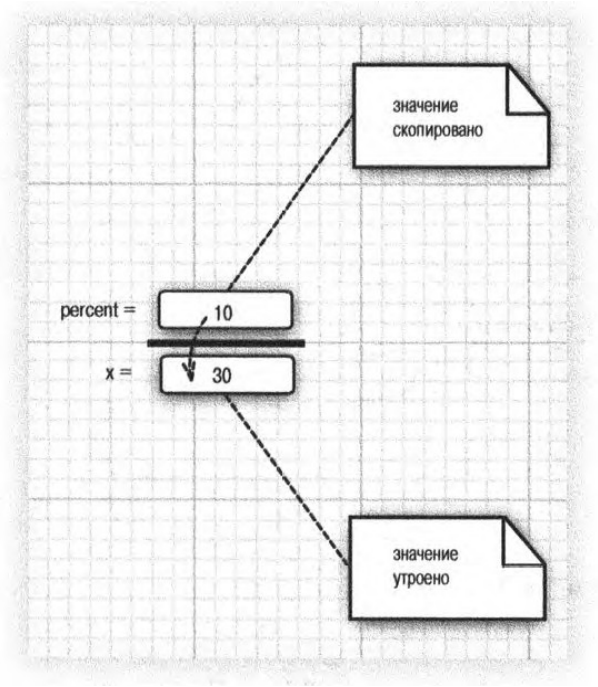

Рисунок 1 – Видоизменение значения в вызываемом методе не оказывает никакого влияния на параметр передаваемый из вызывающей части программы

Но существуют два следующих типа параметров методов.
• Примитивные типы (т.е. числовые и логические значения).
• Ссылки на объекты.

Как было показано выше, методы не могут видоизменить параметры примитивных типов. Совсем иначе дело обстоит с объектами. Нетрудно реализовать метод, утраивающий зарплату работников, следующим образом:

```
public static void tripleSalary(Employee x) // сработает!
{
        x.raiseSalary(200);
}
```

При выполнении следующего фрагмента кода происходят перечисленные ниже действия.

```
harry = new Employee(...);
tripleSalary(harry);
```

1. Переменная x инициализируется копией значения переменной harry, т.е. ссылкой на объект.
2. Метод raiseSalary() применяется к объекту по этой ссылке. В частности, объект типа Employee, доступный по ссылкам x и harry, получает сумму зарплаты работников, увеличенную на 200%.
3. Метод завершает свою работу, и его параметр x больше не используется. Разумеется, переменная harry продолжает ссылаться на объект, где зарплата увеличена втрое (Рисунок 2).

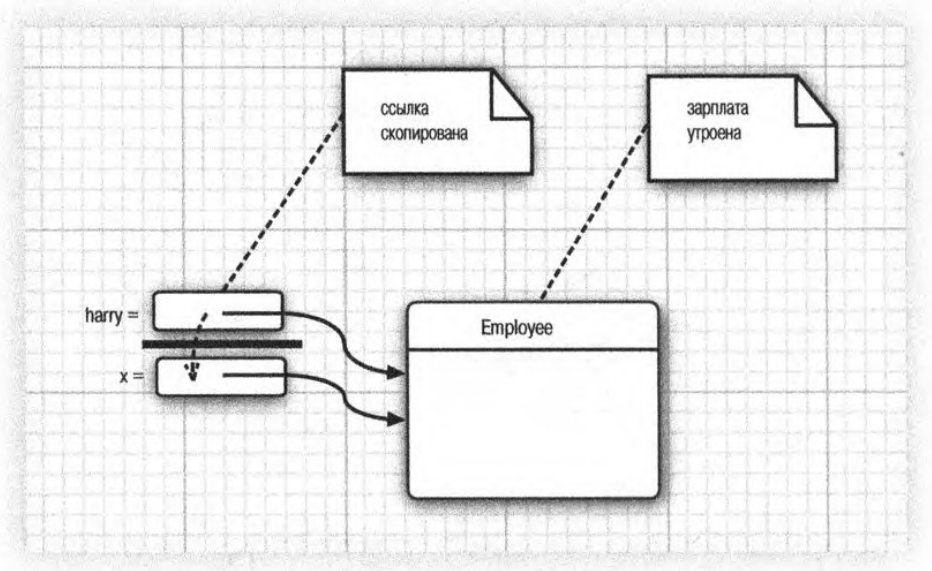

Рисунок 2 – Если параметр ссылается на объект, последний может быть видоизменен

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, совсем не трудно. В действительности такие изменения вносятся очень часто по следующей простой причине: метод получает копию ссылки на объект, поэтому копия и оригинал ссылки указывают на один и тот же объект.

Во многих языках программирования (в частности, C++ и Pascal) предусмотрены два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что в Java при передаче объектов используется вызов по ссылке. Но это совсем не так. Для того чтобы развеять это бытующее заблуждение, обратимся к конкретному примеру.

Ниже приведен метод, выполняющий обмен двух объектов типа Employee.

```
public static void swap(Employee x, Employee y) // не сработает!
```

```
{
        Employee temp = x;
        x = y;
        y = temp;
}
```

Если бы в Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод действовал бы следующим образом:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// ссылается ли теперь переменная a на Bob, а переменная b - на Alice?
```

Но на самом деле этот метод не меняет местами ссылки на объекты, хранящиеся в переменных a и b. Сначала параметры x и y метода swap() инициализируются копиями этих ссылок, а затем эти копии меняются местами в данном методе, как показано ниже.

```
// переменная x ссылается на Alice, а переменная y - на Bob
Employee temp = x;
x = y;
y = temp;
// теперь переменная x ссылается на Bob, а переменная y - на Alice
```

В конце концов, следует признать, что все было напрасно. По завершении работы данного метода переменные x и y уничтожаются, а исходные переменные a и b продолжают ссылаться на прежние объекты (Рисунок 3).

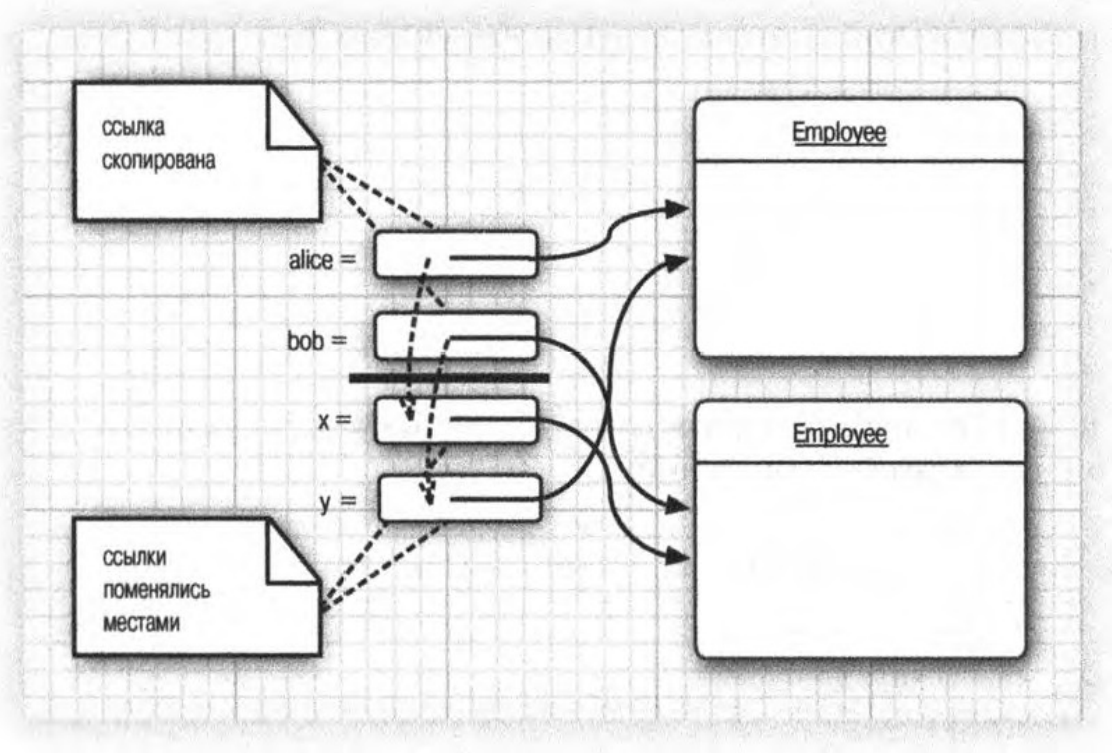

Рисунок 3 – Перестановка ссылок в вызывающем методе не имеет никаких последствий для вызывающей части программы

Таким образом, в Java для передачи объектов не применяется вызов по ссылке.

Вместо этого ссылки на объекты передаются *по значению.* Ниже поясняется, что может и чего он не может метод делать со своими параметрами.

• Метод не может изменять параметры примитивных типов (т.е. числовые и логические значения).

• Метод может изменять *состояние* объекта, передаваемого в качестве параметра.

• Метод не может делать в своих параметрах ссылки на новые объекты.

3.6 Уничтожение объектов и метод finalize()

В некоторых объектно-ориентированных языках программирования и, в частности, в C++ имеются явные деструкторы, предназначенные для уничтожения объектов.

Их основное назначение — освободить память, занятую объектами. А в Java реализован механизм автоматической сборки "мусора", освобождать память вручную нет никакой необходимости, и поэтому в этом языке деструкторы отсутствуют.

Разумеется, некоторые объекты используют кроме памяти и другие ресурсы, например файлы, или оперируют другими объектами, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы занимаемые ресурсы освобождались, когда они больше не нужны.

С этой целью в любой класс можно ввести метод finalize(), который будет вызван перед тем, как система сборки "мусора" уничтожит объект. Но на практике, если требуется возобновить ресурсы и сразу использовать их повторно, *нельзя* полагаться только на метод finalize(), поскольку заранее неизвестно, когда именно этот метод будет вызван.

3.7 Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами.* Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими. Стандартная библиотека Java содержит большое количество пакетов, в том числе java.lang, java.util, java.net и т.д. Стандартные пакеты Java представляют собой иерархические структуры. Подобно каталогам на диске компьютера, пакеты мшут быть вложены один в другой. Все стандартные пакеты относятся к иерархиям пакетов java и javax.

Пакеты служат в основном для обеспечения однозначности имен классов. Допустим, двух программистов осенила блестящая идея создать класс Employee. Если оба класса будут находиться в разных пакетах, конфликт имен

не возникнет. Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете, записанное в обратном порядке (оно по определению единственное в своем роде). В составе пакета можно создавать подпакеты и использовать их в разных проектах.

Единственная цель вложенных пакетов — гарантировать однозначность имен. С точки зрения компилятора между вложенными пакетами отсутствует какая-либо связь. Например, пакеты java.util и java.util, jar вообще не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

3.7.1 Импорт классов

В классе могут использоваться все классы из собственного пакета и все *открытые* классы из других пакетов. Доступ к классам из других пакетов можно получить двумя способами. Во-первых, перед именем каждого класса можно указать полное имя пакета, как показано ниже.

java.time. LocalDate today = **java.time.**LocalDate.now**() ;**

Очевидно, что этот способ не совсем удобен. Второй, более простой и распространенный способ предусматривает применение ключевого слова import. В этом случае имя пакета указывать перед именем класса необязательно.

Импортировать можно как один конкретный класс, так и пакет в целом. Операторы import следует разместить в начале исходного файла (после всех операторов package). Например, все классы из пакета java.time можно импортировать следующим образом:

```
import java.time.*;
```

После этого имя пакета не указывается, как показано ниже.

```
LocalDate today = LocalDate.now();
```

Отдельный класс можно также импортировать из пакета следующим образом:

```
import java.time.LocalDate;
```

Проще импортировать все классы, например, из пакета java.time. *. На объем кода это не оказывает никакого влияния. Но если указать импортируемый класс явным образом, то читающему исходный код программы станет ясно, какие именно классы будут в ней использоваться.

3.7.2 Статический импорт

Имеется форма оператора import, позволяющая импортировать не только классы, но и статические методы и поля. Допустим, в начале исходного файла введена следующая строка кода:

```
import static java.lang.System.*;
```

Это позволит использовать статические методы и поля, определенные в классе System, не указывая имени класса:

out.println("Goodbye, World!"); // вместо **System.out**

exit(0); // вместо **System.exit**

Статические методы или поля можно также импортировать явным образом:

import static java.lang.System.out;

Но в практике программирования на Java такие выражения, как System, out или System, exit, обычно не сокращаются. Ведь в этом *случае* исходный код станет более трудным для восприятия.

3.7.3 Ввод классов в пакеты

Чтобы ввести класс в пакет, следует указать имя пакета в начале исходного файла *перед* определением класса. Например:

```
package ru.sfu;
public class Employee
{
}
```

Если оператор package в исходном файле не указан, то классы, описанные в этом файле, вводятся в *пакет по умолчанию.* У пакета по умолчанию нет имени. Все рассмотренные до сих пор классы принадлежали пакету по умолчанию.

Пакеты следует размещать в подкаталоге, путь к которому соответствует полному имени пакета. Например, все файлы классов из пакета ru.sfu должны находиться в подкаталоге ru/sfu. Компилятор размещает файлы классов в той же самой структуре каталогов.

3.7.4 Область действия пакетов

В приведенных ранее примерах кода уже встречались модификаторы доступа public и private. Открытые компоненты, помеченные ключевым словом public, могут использоваться любым классом. А закрытые компоненты, в объявлении которых указано ключевое слово private, могут использоваться только тем классом, в котором они были определены. Если же ни один из модификаторов доступа не указан, то компонент программы (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

4 Наследование

4.1 Определение подклассов

Ниже показано, каким образом определяется класс Manager, производный

от класса Employee. *Для* обозначения наследования в Java служит ключевое слово extends.

```
class Manager extends Employee
{
        //Дополнительные методы и поля
}
```

Ключевое слово extends означает, что на основе существующего класса создается новый класс. Существующий класс называется *суперклассом, базовым* или *родительским,* а вновь создаваемый — *подклассом, производным* или *порожденным.* В среде программирующих на Java наиболее широко распространены термины *суперкласс* и *подкласс,* хотя некоторые из них предпочитают пользоваться терминами *родительский* и *порожденный,* более тесно связанными с понятием наследования.

Класс Employee является суперклассом. Это не означает, что он имеет превосходство над своим подклассом или обладает более широкими функциональными возможностями. На самом деле все *наоборот:* функциональные возможности подкласса шире, чем у суперкласса. Как станет ясно в дальнейшем, класс Manager инкапсулирует больше данных и содержит больше методов, чем его суперкласс Employee.

4.2 Ключевое слово super

В Java существует ключевое слово **super**, которое обозначает суперкласс, т.е. класс, производным от которого является текущий класс. В данном случае, супер не означает превосходство, скорее даже наоборот, дочерний класс имеет больше методов, чем родительский. Само слово пошло из теории множеств, где используется термин супермножество.

У ключевого слова super имеются две общие формы:

1) Для вызова конструктора суперкласса:

```
super(списокАргументов);
```

2) Для обращения к члену суперкласса, скрываемому членом подкласса:

```
super.member;
```

4.3 Представление о вызовах методов

Важно понять, каким образом вызов метода применяется к объекту. Допустим, делается вызов x.f(*args)* и неявный параметр x объявлен как объект класса C (прим. Здесь C — имя класса, а не язык программирования)**.** В таком случае происходит следующее.

1. Сначала компилятор проверяет объявленный тип объекта, а также имя метода. Следует иметь в виду, что может существовать несколько методов под именем **f,** имеющих разные типы параметров, например, методы **f(int)** и **f (String).** Компилятор перечисляет все методы под именем **f** в классе **C** и все доступные методы под именем **f** в суперклассах, производных от класса **C.**

(Закрытые методы в суперклассе недоступны.) Итак, компилятору известны все **в**озможные претенденты на вызов метода под именем f .

2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов под именем f имеется только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется *разрешением перегрузки.* Например, при вызове x.f ("Hello") компилятор выберет метод f (String), а не метод f (int). Но ситуация может осложниться вследствие преобразования типов (int — в double, Manager — в Employee и т.д.). Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования типов оказывается несколько методов, соответствующих данному вызову, выдается сообщение об ошибке.

3. В конечном итоге компилятору становятся известными типы параметров и имя метода, который должен быть вызван.

4. Если метод является закрытым (private), статическим (static), конечным (final) или конструктором, компилятору точно известно, как его вызвать. (Модификатор доступа final описывается в следующем разделе.) Такой процесс называется *статическим связыванием.* В противном случае вызываемый метод определяется по фактическому типу неявного параметра, а во время выполнения программы происходит *динамическое связывание.* В данном примере компилятор сформировал бы вызов метода f (String) путем **динамического связывания**.

5. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую фактическому типу объекта, на который ссылается переменная x. Допустим, объект имеет *фактический* тип D подкласса, производного от класса C. Если в классе D определен метод f (String), то вызывается именно он. В противном случае поиск вызываемого метода f (String) осуществляется в суперклассе и т.д.

6. На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает для каждого класса *таблицу методов*, в которой перечисляются сигнатуры всех методов и фактически вызываемые методы. При вызове метода виртуальная машина просто просматривает таблицу методов. В данном примере виртуальная машина проверяет таблицу методов класса **D** и обнаруживает вызываемый метод **f (String)**. Такими методами могут быть **D.f (String)** или **X.f (String),** если **X** — некоторый суперкласс для класса **D.** С этим связана одна любопытная особенность. Если вызывается метод **super**. **f (param)**, то компилятор просматривает таблицу методов суперкласса, на который указывает неявный параметр **super.**

4.4 Предотвращение наследования: конечные классы и методы

Иногда наследование оказывается нежелательным. Классы, которые нельзя расширить, называются *конечными.* Для указания на это в определении

класса используется модификатор доступа **final.** Допустим, требуется предотвратить создание подклассов, производных от класса **Executive.** В таком случае класс **Executive** определяется следующим образом:

```
final class Executive extends Manager
{
        …
}
```

4.5 instanceof

instanceof – это ключевое слово. Это двоичный оператор, используемый для проверки, является ли объект (экземпляр) подтипом данного типа. Он возвращает либо true, либо false. Он возвращает true, если левая часть выражения является экземпляром имени класса с правой стороны. Если мы применим оператор instanceof с любой переменной с нулевым (null) значением, он вернет false. Это полезно, когда ваша программа может получить информацию о типе времени выполнения для объекта. Ключевое слово instanceof также известно как оператор сравнения типов, поскольку он сравнивает экземпляр с типом.

4.6 Абстрактные классы

Чем дальше вверх по иерархии наследования, тем более универсальными и абстрактными становятся классы. В некотором смысле родительские классы, находящиеся на верхней ступени иерархии, становятся настолько абстрактными, что их рассматривают как основу для разработки других классов, а не как классы, позволяющие создавать конкретные объекты..

От реализации этого метода в классе можно вообще отказаться, если воспользоваться ключевым словом abstract, как показано ниже.

```
public abstract String getDescription();
// реализация не требуется
```

Для большей ясности класс, содержащий **один или несколько абстрактных методов**, можно объявить **абстрактным** следующим образом:

```
public abstract class Person
{
        public abstract String getDescription();
}
```

Помимо абстрактных методов, абстрактные классы могут содержать конкретные поля и методы.

Создать экземпляры абстрактного класса нельзя.

Следует иметь в виду, что для абстрактных классов *можно* создавать объектные переменные, но такие переменные должны ссылаться на объект неабстрактного класса.

4.7 Глобальный суперкласс Object

Класс Object является исходным предшественником всех остальных классов, поэтому каждый класс в Java расширяет класс Object. Если суперкласс явно не указан, им считается класс Object. А поскольку каждый класс в Java расширяет класс Object, то очень важно знать, какими средствами обладает сам класс Object.

4.7.1 Метод equals()

В методе equals() из класса Object проверяется, равны ли два объекта. А поскольку метод equals() реализован в классе Object, то в нем определяется только следующее: ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект равен самому себе. Для некоторых классов большего и не требуется.

Ниже приведены рекомендации для создания приближающегося к идеалу метода equals():

1. Присвойте явному параметру имя otherObject. Впоследствии его тип нужно будет привести к типу другой переменной под названием other.

2. Проверьте, одинаковы ли ссылки this и otheObject, следующим образом:

if (this == otherObject) return true;

Это выражение составлено лишь в целях оптимизации проверки. Ведь намного быстрее проверить одинаковость ссылок, чем сравнивать поля объектов.

3. Выясните, является ли ссылка otherObject пустой (null), как показано ниже. Если она оказывается пустой, следует возвратить логическое значение false. Эту проверку нужно сделать обязательно.

if (otherObject == null) return false;

4. Сравните классы this и otherObject. Если семантика проверки может измениться в подклассе, воспользуйтесь методом getClass() следующим образом:

if (getClass() != otherObject.getClass()) return false;

Если одна и та же семантика остается справедливой для *всех* подклассов, произведите проверку с помощью операции instanceof следующим образом:

if (! (otherObject instanceof *ИмяКласса))* **return false;**

5. Приведите тип объекта otherObject к типу переменной требуемого класса:

ИмяКласса **other = (***ИмяКласса)* **otherObject;**

6. Сравните все поля, как показано ниже. Для полей примитивных типов служит операция = , а для объектных полей — метод Obj e c t s . equals (). Если все поля двух объектов совпадают, возвращается логическое значение true, а иначе — логическое значение false,

return ***поле1*** == other.поле!

&& ***поле2.***equals(other.***поле2)***

&& …;

Если вы переопределяете в подклассе метод equals(), в него следует включить вызов super.equals (other).

4.7.2 Метод hashCode()

Хеш-код — это целое число, генерируемое на основе конкретного объекта. Хеш-код можно рассматривать как некоторый шифр: если x и y — разные объекты, то с большой степенью вероятности должны различаться результаты вызовов x.**hashCode()** и y.**hashCode()**.

Если переопределяется метод **equals**(), то следует переопределить и метод **hashCode**() для объектов, которые пользователи могут вставлять в хеш-таблицу.

Методы **equals** () и **hashCode** () должны быть совместимы: если в результате вызова x.**equals(y)** возвращается логическое значение **true,** то и результаты вызовов x**.hashCode**() и y.**hashCode**() также должны совпадать.

4.8 Объектные оболочки и автоупаковка

Иногда переменные примитивных типов вроде int приходится преобразовывать в объекты. У всех примитивных типов имеются аналоги в виде классов. Например, существует класс Integer, соответствующий типу int. Такие классы принято называть *объектными оболочками.* Они имеют вполне очевидные имена: Integer, Long, Float, Double, Short, Byte, Character и Boolean. (У первых шести классов имеется общий суперкласс Number.) Классы объектных оболочек являются неизменяемыми. Эго означает, что изменить значение, хранящееся в объектной оболочке после ее создания, нельзя.

Допустим, в списочном массиве требуется хранить целые числа. К сожалению, с помощью параметра типа в угловых скобках нельзя задать примитивный тип, например, выражение ArrayList<int> недопустимо. И здесь приходит на помощь класс объектной оболочки Integer. В частности, списочный массив, предназначенный для хранения объектов типа Integer, можно объявить следующим образом:

ArrayList<Integer> **list = new ArrayList<Integer>();**

Правда, в Java имеется удобное языковое средство, позволяющее добавлять и извлекать элементы из массива. Рассмотрим следующую строку кода:

list.add(3);

Она автоматически преобразуется в приведенную ниже строку кода. Подобное автоматическое преобразование называется *автоупаковкой.*

list.add(new Integer(3));

5 Интерфейсы

Интерфейс в Java не является классом. Он представляет собой ряд *требований*, предъявляемых к классу, который должен соответствовать интерфейсу. Как правило, один разработчик, собирающийся воспользоваться трудами другого разработчика для решения конкретной задачи, заявляет: "Если ваш класс будет соответствовать определенному интерфейсу, я смогу решить свою задачу". Обратимся к конкретному примеру. Метод **sort**() из класса **Array** позволяет упорядочить массив объектов при одном условии: объекты должны принадлежать классам, реализующим интерфейс **Comparable.**

Этот интерфейс определяется следующим образом:

```
public interface Comparable
{
        int compareTo(Object other);
}
```

Это означает, что любой класс, реализующий интерфейс **Comparable,** должен содержать метод **compareTo** (), получающий параметр типа **Object** и возвращающий целое значение.

Все методы интерфейса автоматически считаются открытыми, поэтому, объявляя метод в интерфейсе, указывать модификатор доступа **public** необязательно.

Для того чтобы класс реализовал интерфейс, нужно выполнить следующие действия.

1. Объявить, что класс реализует интерфейс.
2. Определить в классе все методы, указанные в интерфейсе.

Для объявления реализации интерфейса в классе служит ключевое слово **implements,** как показано ниже.

class Employee **implements** Comparable

5.1 Свойства интерфейсов

Интерфейсы — это не классы. В частности, с помощью операции new нельзя создать экземпляр интерфейса следующим образом:

x = new Comparable(...); // Неверно!

Но, несмотря на то, что конструировать интерфейсные объекты нельзя, объявлять интерфейсные переменные можно следующим образом:

Comparable x; // Верно!

При этом интерфейсная переменная должна ссылаться на объект класса, реализующего данный интерфейс, как в приведенном ниже фрагменте кода.

x = new Employee (...); // Верно, если класс **Employee**
// реализует интерфейс **Comparable**

Как известно, в ходе операции instanceof проверяется, принадлежит ли объект заданному классу. Но с помощью этой операции можно также проверить, реализует ли объект заданный интерфейс:

if (anObject instanceof Comparable) { ... }

Аналогично классам, интерфейсы также могут образовывать иерархию наследования. Это позволяет создавать цепочки интерфейсов в направлении от более абстрактных к более специализированным.

И хотя у интерфейса не может быть ни полей экземпляров, ни статических методов, в нем можно объявлять константы, как показано ниже.

```
public interface Powered extends Moveable
{
    double milesPerGallon();
    double SPEED_LIMIT = 95; // открытая статическая конечная константа
}
```

5.2 Методы по умолчанию

Для любого интерфейсного метода можно предоставить реализацию *по умолчанию*.

Такой метод следует пометить модификатором доступа default, как показано ниже.

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
        //по умолчанию все элементы одинаковы
}
```

Безусловно, пользы от такого метода не очень много, поскольку в каждой настоящей реализации интерфейса Comparable этот метод будет переопределен. Но иногда методы по умолчанию оказываются все же полезными. Так, если требуется получить уведомление о событии от щелчка кнопкой мыши, то для этой цели, скорее всего, придется реализовать интерфейс с пятью методами.

Методы по умолчанию играют важную роль в дальнейшем *развитии интерфейсов.* Рассмотрим в качестве примера интерфейс Collection, многие годы входящий в состав стандартной библиотеки Java. Допустим, что некогда был предоставлен следующий класс, реализующий интерфейс Collection:

public class Bag implements Collection

а впоследствии, начиная с версии Java 8, в этот интерфейс был внедрен метод stream(). Допустим также, что метод stream () не является методом по умолчанию. В таком случае класс Bag больше не компилируется, поскольку он не реализует новый метод из интерфейса Collection. Таким образом, внедрение

в интерфейс метода не по умолчанию нарушает *совместимость на уровне исходного кода.*

Но допустим, что этот класс не перекомпилируется и просто используется содержащий его старый архивный JAR-файл. Этот класс по-прежнему загружается, несмотря на отсутствующий в нем метод. В программах могут по-прежнему строиться экземпляры класса Bag, и ничего плохого не произойдет. (Внедрение метода в интерфейс *совместимо на уровне двоичного кода.)* Но если в программе делается вызов метода stream() для экземпляра класса Bag, то возникает ошибка типа AbstractMethodError.

Подобные затруднения можно устранить, если объявить метод stream () как default. И тогда класс Bag будет компилироваться снова. А если этот класс загружается без перекомпиляции и метод stream () вызывается для экземпляра класса Bag, то такой вызов происходит по ссылке Collection.stream.

5.3 Также дополнительно для своего профессионального развития вы можете изучить темы “Клонирование объектов” и “Функциональные интерфейсы”.

6 Исключения

6.1 Классификация исключений

В языке Java объект исключения всегда является экземпляром класса, производного от класса Throwable. Как станет ясно в дальнейшем, если стандартных классов недостаточно, можно создавать и свои собственные классы исключений. На рисунке 4 показана в упрощенном виде иерархия наследования исключений в Java.

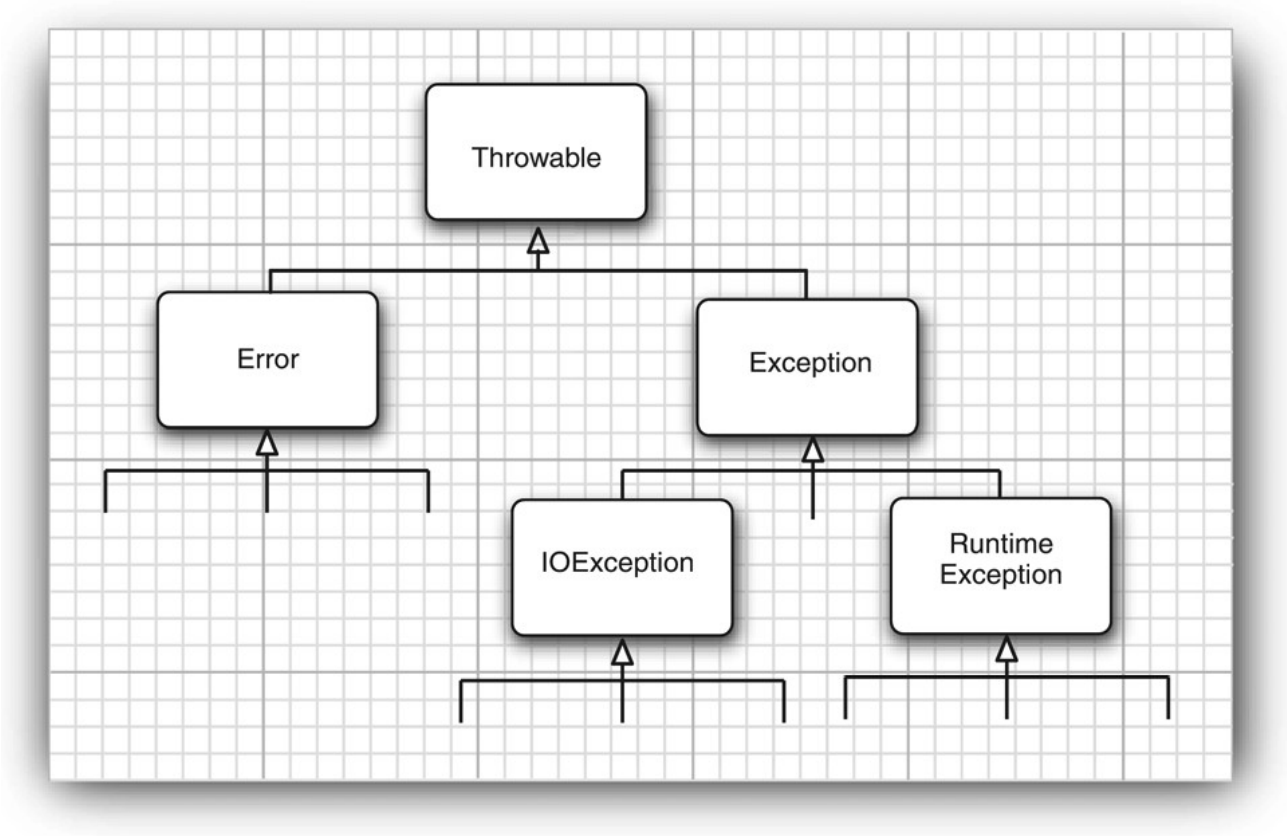

Рисунок 4 – Иерархия наследования исключений в Java

Обратите внимание на то, что иерархия наследования исключений сразу же разделяется на две ветви: Error и Exception, хотя общим предшественником для всех исключений является класс Throwable. Иерархия класса Error описывает внутренние ошибки и ситуации, возникающие в связи с нехваткой ресурсов в исполняющей системе Java. Ни один объект этого класса нельзя сгенерировать самостоятельно. При возникновении внутренней ошибки в такой системе возможности разработчика прикладной программы крайне ограничены. Можно лишь уведомить пользователя и попытаться аккуратно прервать выполнение программы, хотя такие ситуации достаточно редки.

При программировании на Java основное внимание следует уделять иерархии класса Exception. Эта иерархия также разделяется на две ветви: исключения, производные от класса RuntimeException, и остальные. Исключения типа RuntimeException возникают вследствие ошибок программирования. Все другие виды исключений являются следствием непредвиденного стечения обстоятельств, например, ошибок ввода- вывода, возникающих при выполнении вполне корректных программ.

Исключения типа **RuntimeException** практически всегда возникают по вине программиста. Так, исключения типа **ArraylndexOutBoundsException** можно избежать, если всегда проверять индексы массива. А исключение **NullPointerException** никогда не возникнет, если перед тем, как

воспользоваться переменной, проверить, не содержит ли она пустое значение **null.**

А как быть, если файл не существует? Нельзя ли сначала проверить, существует ли он вообще? Дело в том, что файл может быть удален сразу же после проверки его существования. Следовательно, понятие существования файла зависит от среды исполнения, а не от кода программы.

В спецификации языка Java любое исключение, производное от класса **Error** или **RuntimeError,** называется *непроверяемым*. Все остальные исключения называются *проверяемыми.* Для всех проверяемых исключений компилятор проверяет наличие соответствующих обработчиков.

6.2 Перехват нескольких исключений

Думаю, слушателю известно, что инициировать исключение можно с помощью оператора throw, блок кода, который тестируется на наличие исключений оборачивается блоком try, а блок обработки – это блок catch. Но как обработать сразу несколько исключений?

В блоке try можно перехватить несколько исключений, обработав их по отдельности. Для каждого типа исключения следует предусмотреть свой блок catch следующим образом:

```
try
{
        код, способный генерировать исключения
}
catch (FileNotFoundException e)
{
        чрезвычайные действия, если отсутствуют нужные файлы
}
catch (UnknownHostException e)
{
        чрезвычайные действия, если хосты неизвестны
}
catch (IOException e)
{
        чрезвычайные действия во всех остальных
        случаях появления ошибок ввода-вывода
}
```

Начиная с версии Java SE7, разнотипные исключения можно перехватывать в одном и том же блоке catch. Допустим, чрезвычайные действия, если нужные файлы отсутствуют или хосты неизвестны, одинаковы. В таком случае блоки catch перехвата обоих исключений можно объединить, как показано ниже. К такому способу перехвата исключений следует прибегать

лишь в тех случаях, когда перехватываемые исключения не являются подклассами друг для друга.

```
try
{
        код, способный генерировать исключения
}
catch (FileNotFoundException | UnknownHostException e)
{
        чрезвычайные действия, если нужные файлы отсутствуют
        или неизвестны хосты
}
catch (IOException e)
{
        чрезвычайные действия во всех остальных
        случаях появления ошибок ввода-вывода
}
```

6.3 Блок finally

Когда в методе генерируется исключение, оставшиеся в нем операторы не выполняются. Если же в методе задействованы какие-нибудь локальные ресурсы, о которых известно лишь ему, то освободить их уже нельзя. Можно, конечно, перехватить и повторно сгенерировать все исключения, но это не совсем удачное решение, поскольку ресурсы нужно освобождать в двух местах: в обычном коде и в коде исключения.

В языке Java принято лучшее решение — организовать блок finally. Код в блоке finally выполняется независимо от того, возникло исключение или нет.

Блок finally можно использовать и без блока catch.

6.4 Оператор try с ресурсами

В своей простейшей форме оператор try с ресурсами выглядит следующим образом:

```
try (Resource res = ...)
{
        использовать ресурс res
}
```

Если в коде имеется блок try, то метод res.close() вызывается автоматически.

Следует, однако, иметь в виду, что эта конструкция эффективна при одном условии: используемый ресурс принадлежит классу, реализующему интерфейс AutoCloseable.