Объектно-ориентированное программирование на Java

С.Ю. Скоробогатов

Весна 2019

Список литературы по модулю

1. Слайды лекций (могут меняться в течение семестра). `www.dropbox.com/s/zfway51gmedagrs/module_java.pdf`
2. Г. Шилдт. Java 8. Руководство для начинающих. 6-е издание. – М.: Издательский дом «Вильямс», 2015. `www.dropbox.com/s/gk0m5j8te9nhly0/Schildt8.pdf`
3. Дж. Гослинг и др. Язык программирования Java SE 8. Подробное описание. 5-е издание. – М.: Издательский дом «Вильямс», 2015. `www.dropbox.com/s/eeh2ix4q0z4utb1/Gosling.pdf`
4. Б. Эккель. Философия Java. Библиотека программиста. 4-е издание. – СПб.: Питер, 2009. `www.dropbox.com/s/76j4399wj7nezci/Eckel.djvu`

Задания к лабораторным работам

Ссылки на методические указания для выполнения лабораторных работ, включающие списки вариантов заданий:

1. Базовые средства разработки для языка Java
   `www.dropbox.com/s/6ixh4em743km8rr/lab1.pdf`
2. Разработка простейшего класса на языке Java
   `???`
3. Полиморфизм на основе интерфейсов в языке Java
   `???`
4. Реализация итераторов в языке Java
   `???`
5. Монады в языке Java
   `???`
6. Графический пользовательский интерфейс
   `???`

История языка Java

Джеймс Гослинг

Основные этапы:

Начало 90-х: Oak – язык для программирования бытовых устройств.

1995: Java 1.0 (разработка интернет-приложений).

1997: Java 1.1 (вложенные классы, рефлексия).

1998/2000/2002: Java 1.2 (коллекции), Java 1.3, Java 1.4 (ассерты).

2004/2006: Java 5 (обобщения, аннотации, перечисления, переменное колическтво параметров методов, расширенный for), Java 6.

2011: Java 7 (новый синтаксический сахар, улучшенная схема обработки ислючений).

2014: Java 8 (замыкания, функциональные интерфейсы).

2017/2018: Java 9, Java 10 (псевдотип var), Java 11.

Основные особенности Java

JIT-компиляция: компилятор Java порождает независимый от аппаратной платформы и операционной системы байт-код, который транслируется в машинный код в процессе выполнения программы;

Переносимость: виртуальные машины Java (JIT-компилятор + run-time + библиотеки классов) существуют практически для всех аппаратных платформ и операционных систем, перенос программы на другую платформу не требует её перекомпиляции;

Автоматическое управление памятью: объекты, которые стали не нужны, автоматически удаляются сборщиком мусора;

Безопасность: строгая статическая типизация, отсутствие арифметики указателей и прочих «небезопасных» возможностей.

Установка Java Development Kit (JDK)

Последовательность действий для ОС Linux.

1. Перейти на страницу:
   `www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html`
2. Согласиться с лицензионным соглашением и скачать файл jdk-11.0.2_linux-x64_bin.tar.gz.
3. Разархивировать файл в домашний каталог, при этом появится подкаталог jdk-11.0.2.
4. Отредактировать файл .profile (или .bashrc), находящийся в домашнем каталоге, добавив в его конец строчки:

```
export JAVA_HOME=$HOME/jdk-11.0.2
export PATH=$JAVA_HOME/bin:$PATH
```

5. Перелогиниться или перезагрузить систему.

Проверка работоспособности JDK

Для проверки работоспособности откройте терминал и выполните команду

```
javac -version
```

Если JDK был правильно установлен, вы увидите сообщение

```
javac 11.0.2
```

Программа «Hello, World»

Программа на Java (HelloWorld.java):

```
1 public class HelloWorld
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello, world!");
6     }
7 }
```

Компиляция:

```
javac HelloWorld.java
```

Запуск:

```
java HelloWorld
```

Понятия объекта и инкапсуляции

Определение 1. *Объект* (object) – это самоописывающая структура данных, обладающая внутренним состоянием и способная обрабатывать передаваемые ей сообщения.

В языках программирования со статической проверкой (в частности, в Java) набор сообщений, которые может обрабатывать объект, фиксирован на этапе компиляции программы. При этом передача объекту сообщения, которое он не может обработать, выявляется на этапе компиляции.

Определение 2. *Инкапсуляция* (incapsulation) – один из основных принципов объектно-ориентированного программирования, заключающийся в том, что доступ к внутреннему состоянию объекта извне осуществляется только через механизм передачи сообщений.

Объект является самоописывающей структурой, потому что содержит информацию о классе, которому он принадлежит.

Определение 3. *Класс* (class) – это тип данных, значениями которого являются объекты, имеющие сходное внутреннее состояние и обрабатывающие одинаковый набор сообщений.

Класс можно рассматривать как шаблон для порождения объектов. Поэтому объекты называют также экземплярами класса (class instances).

В языке Java все значения являются объектами, кроме значений примитивных типов: `char` (16-разрядный символ Unicode), `byte` (знаковый, 8 бит), `short` (знаковый, 16 бит), `int` (знаковый, 32 бит), `long` (знаковый, 64 бит), `float`, `double`, `boolean`.
Обратите внимание, все целые типы – знаковые.

Объявление классов в Java

В языке Java классы делятся на публичные и непубличные. В каждом java-файле должен быть ровно один публичный класс, причём имя файла должно совпадать с именем класса. Непубличных классов в файле может быть несколько. Непубличные классы видны только в пределах того файла, где они объявлены.
Публичный класс объявляется следующим образом:

```
public class Имя
{
    ...
}
```

В объявлении непубличного класса отсутствует ключевое слово `public`.
Обратите внимание на то, что точка с запятой после объявления класса не ставится.

Внутри фигурных скобок в объявлении класса располагаются объявления членов класса. К членам класса относятся:

экземплярные поля – обеспечивают хранение внутреннего состояния объектов;

статические поля – предназначены для хранения данных, общих для всех объектов класса;

экземплярные методы – отвечают за обработку передаваемых объекту сообщений;

статические методы – выполняют действия, для которых не нужен доступ к конкретному объекту класса;

экземплярные конструкторы – инициализируют только что созданные объекты класса;

статический конструктор (набор static-блоков) – инициализирует статические поля класса;

вложенные классы – главным образом, представляют объекты, необходимые для реализации данного класса.

Доступ к членам класса

В общем случае, для доступа к членам класса используется бинарная операция «.». Её первый операнд – либо объект, либо класс, а второй операнд – имя члена класса.

```
1 person.setName("Вася");    // Вызов экз. метода
2 int count = Person.count; // Обращение к стат. полю
```

Объявления членов класса предваряются *модификаторами доступа*, которые управляют доступом к членам класса:

private доступ разрешён только из тела класса;

без модификатора доступ разрешён для самого класса и для классов из того же пакета.

protected доступ разрешён для самого класса, для классов из того же пакета, а также для наследников класса (пакеты и наследование мы рассмотрим позже);

public доступ возможен откуда угодно.

Определение 4. *Экземплярное поле* (instance field) – именованная составная часть внутреннего состояния объекта.

Многие объектно-ориентированные языки допускают прямое обращение извне к экземплярным полям (C++, Java, C#, но не Ruby). Эта практика является нарушением инкапсуляции.
В Java объявления экземплярных полей выглядят как объявления полей структур в языке C. При этом каждое объявление предваряется модификатором доступа.
Пример:

```
1 class Person
2 {
3     public String name;
4     public int yearOfBirth;
5     private String address;
6 }
```

Часто бывает нужно, чтобы часть внутреннего состояния объекта некоторого класса была общей (разделяемой) для всех объектов этого класса.

Определение 5. *Статическое поле* (static field), принадлежащее некоторому классу – это поле, разделяемое всеми объектами этого класса.

В языке Java статические поля объявляются с модификатором `static`.

Пример:

```
1  class Point
2  {
3      public int x, y;          // Координаты точки
4      public static int count;// Общее количество точек
5  }
```

Экземплярные и статические методы

Определение 6. *Экземплярный метод* (instance method) – это подпрограмма, осуществляющая обработку переданного объекту сообщения.

Передача объекту сообщения сводится к вызову соответствующего экземплярного метода.
Экземплярный метод имеет доступ к внутреннему состоянию объекта, то есть может читать и изменять значения экземплярных полей объекта.
Доступ к внутреннему состоянию объекта обеспечивается за счёт передачи в экземплярный метод ссылки (т.е. указателя) на объект. В Java эта ссылка передаётся неявно и имеет имя `this`.

Определение 7. *Статический метод* (static method), объявленный в некотором классе – это метод, не имеющий доступа к внутреннему состоянию объектов этого класса.

Объявление методов в Java

В языке Java объявление метода располагается внутри тела класса. При этом статические методы объявляются с модификатором `static`.

Пример:

```
1  class Point
2  {
3      public double x, y;
4
5      public double dist()
6      {
7          return Math.sqrt(x*x + y*y);
8      }
9
10     public static boolean less(Point a, Point b)
11     {
12         return a.dist() < b.dist();
13     }
14 }
```

Определение 8. *Сигнатура метода* (method signature) – это информация о количестве и типах формальных параметров метода.

Если у метода нет формальных параметров, говорят, что он имеет *пустую сигнатуру*.

Определение 9. *Перегрузка метода* (method overloading) – это объявление для заданного класса двух или более методов, имеющих одинаковое имя, но различные сигнатуры.

Решение о том, куда именно передаётся управление при вызове перегруженного метода X, принимается на этапе компиляции на основе сопоставления типов фактических параметров вызываемого метода и сигнатур методов, имеющих имя X.

Пример: перегрузка методов


```
1  class Point
2  {
3      public double x, y;
4
5      public double dist(double x, double y)
6      {
7          double dx = this.x - x, dy = this.y - y;
8          return Math.sqrt(dx*dx + dy*dy);
9      }
10
11     public double dist(Point p)
12     {
13         return dist(p.x, p.y);
14     }
15 }
```

Определение 10. *Раннее связывание* (early binding) – это определение адреса вызываемого экземплярного метода во время компиляции программы.

Определение 11. *Позднее связывание* (late binding) – это определение адреса вызываемого экземплярного метода на основе информации о классе объекта во время выполнения программы.

Определение 12. Экземплярные методы, для вызовов которых выполняется позднее связывание, называются *виртуальными* (virtual methods).

В языке Java все методы – виртуальные.

Понятие экземплярного конструктора

Определение 13. *Экземплярный конструктор* (instance constructor) – это экземплярный метод, предназначенный для инициализации только что созданного объекта.

Инициализация объекта главным образом заключается в заполнении информации о классе, которому принадлежит объект.
В Java и других объектно-ориентированных языках операция создания объекта, как правило, объединена с вызовом конструктора. Это гарантирует отсутствие неинициализированных объектов.

Конструктор по умолчанию

В некоторых языках конструкторы имеют имена (Object Pascal), но чаще всего конструкторы фактически безымянны (C++, C#, Java, Ruby). Конструкторы, как и всякие методы, могут быть перегружены.

Определение 14. *Конструктор по умолчанию* (default constructor) – это экземплярный конструктор с пустой сигнатурой.

Во всех языках программирования конструктор по умолчанию автоматически создаётся компилятором, если для класса не определён ни один конструктор.

Объявление конструктора в Java

В языке Java экземплярный конструктор представляет собой метод, имя которого совпадает с именем класса, а возвращаемое значение отсутствует.

```
1  class Point
2  {
3      private double x, y;
4
5      public Point()
6      {
7          x = y = 0;
8      }
9
10     public Point(double x, double y)
11     {
12         this.x = x;
13         this.y = y;
14     }
15 }
```

Размещение объектов в памяти

В Java объекты, в отличие от значений примитивных типов, могут располагаться только в динамической памяти (куче). Кроме того, объекты не могут вкладываться друг в друга. Другими словами, объекты не могут лежать в локальных переменных и параметрах методов, в полях объектов и элементах массивов. Вместо этого там хранятся только указатели на объекты, которые в языке Java называются *объектными ссылками*.

Тем самым в Java отпадает необходимость различать объект и ссылку на объект, и, например, смысл объявления

```
Point p;
```

состоит в том, что `p` – это ссылка на объект класса `Point`. Заметим, кроме того, что значения примитивных типов (`int`, `float` и т.п.) не могут располагаться на верхнем уровне кучи. Их место – в локальных переменных и параметрах методов, в полях объектов и элементах массивов.

Операция «new»

Создание объекта в Java выполняется с помощью операции `new`:

```
new имя_класса(фактические_параметры_конструктора)
```

Пример:

```
Point p = new Point(1.5, 3.5);
```

Массивы в Java – тоже объекты, то есть они могут «жить» только на верхнем уровне кучи. Для создания массива применяется специальная форма операции `new`:

```
new тип_элемента [размер]
```

Пример:

```
int[] a = new int [10];
Point[] pa = new Point [20]; // массив ссылок!
```

Обратите внимание, что тип массива записывается как

```
тип_элемента[]
```

Массивовые литералы

Операция **new** имеет специальную форму для создания инициализированного массива (массивовый литерал):

```
new тип_элемента [] { значение1, значение2, ... }
```

Пример:

```
int[] a = new int[] { 1, 2, 3, 4 };

Point[] pa = new Point[] {
    new Point(1.0, 2.0),
    new Point(2.0, -4.5),
    new Point(0, -1.5)
};
```

В отличие от языка C, массивовые литералы могут применяться не только в объявлениях. Например, если метод в качестве параметра принимает массив, этот массив может быть создан при вызове метода:

```
find(x, new int[] { 10, 20, 30 });
```

Понятие статического конструктора

Определение 15. *Статический конструктор* (static constructor) – это статический метод, предназначенный для инициализации статических полей класса. Статический конструктор не имеет параметров и вызывается до вызова любого статического метода или конструктора класса.

Как правило, статические конструкторы не очень нужны, потому что в Java можно инициализировать статические поля прямо при их объявлении.

```
1  class Point
2  {
3      private static int count = 0;
4      private double x, y;
5
6      public Point(double x, double y)
7      {
8          this.x = x;  this.y = y;  count++;
9      }
10 }
```

«Static»-блоки

В Java объявление класса может содержать один или несколько «static»-блоков, записывемых как

```
static {
    последовательность_операторов
}
```

«Static»-блоки выполняются при первом обращении к классу, где под обращением к классу мы будем понимать создание объекта, вызов статического метода, обращение к статическому полю и т.д.
Если объявление класса содержит несколько «static»-блоков, то они будут выполняться в том порядке, в котором они перечислены в теле класса. Совокупность всех «static-блоков» класса играет роль статического конструктора.

Пример: использование «static»-блоков


```
1  class Point
2  {
3      static {
4          System.out.println("Point gets initialized");
5      }
6
7      private static double count;
8      static {
9          count = 0;
10     }
11
12     private double x, y;
13     public Point(double x, double y)
14     {
15         this.x = x;  this.y = y;
16         count++;
17     }
18 }
```

Понятие субтипизации

Определение 16. Тип данных A является *подтипом* (subtype) для типа данных B, если программный код, расчитанный на обработку значений типа B, может быть корректно использован для обработки значений типа A.

Определение 17. *Субтипизация* (subtyping) – это свойство языка программирования, означающее возможность использования подтипов в программах.

Для лучшего понимания субтипизации можно рассматривать тип данных как множество значений. Тогда тип A является подтипом для типа B, если $A \subseteq B$.
Семантика языков, поддерживающих субтипизацию и ориентированных на статическую проверку, допускает, что одно значение может иметь сразу несколько типов. (Действительно, значение может принадлежать сразу нескольким множествам.)

Явная и неявная субтипизация

Различают неявную и явную субтипизацию.

В языке программирования, поддерживающем *неявную субтипизацию*, решение о том, является ли тип A подтипом для типа B, принимается на основе анализа структуры значений этих типов. Поэтому неявную субтипизацию часто называют *структурной субтипизацией* (structural subtyping).

В языке программирования, поддерживающим *явную субтипизацию* (explicit subtyping или nominal subtyping), тип A является подтипом для типа B тогда и только тогда, когда A является *наследником* B.

В языке Java используется явная субтипизация.

Наследование как механизм явной субтипизации

Определение 18. *Наследование* (inheritance) – это способ получения нового класса на основе уже существующего класса, сочетающий усложнение внутреннего состояния объектов, расширение ассортимента обрабатываемых сообщений и изменение реакции на некоторые сообщения.

Если новый класс A создан на основе уже существующего класса B с помощью механизма наследования, то говорят, что класс A является *производным классом* (derived class) или *подклассом* (subclass) по отношению к классу B. В свою очередь, класс B выступает в роли *базового класса* (base class) или *суперкласса* (superclass) для класса A.

При наследовании производный класс получает все экземплярные поля базового класса и все экземплярные методы, кроме конструкторов. Тело любого конструктора производного класса должно начинаться с вызова одного из конструкторов базового класса.

Наследование как механизм явной субтипизации

Определение 19. *Одиночное наследование* (single inheritance) – это вариант наследования, при котором у класса не может быть более одного базового класса.

В языке Java поддерживается только одиночное наследование. Его синтаксис:

```
class ПроизводныйКласс extends БазовыйКласс {
    ...
}
```

По умолчанию класс является наследником встроенного класса `Object`.

Если конструктор базового класса является конструктором по умолчанию, его вызов добавляется компилятором Java в конструктор производного класса автоматически. В противном случае необходимо явно вызывать конструктор базового класса в конструкторе производного класса.

Вызов конструктора базового класса

В Java вызов конструктора базового класса осуществляется из тела конструктора производного класса с помощью оператора `super`:

```
super(фактические_параметры_конструктора);
```

Пример:

```
1  class Animal {
2      private String species;
3      public Animal(String species) {
4          this.species = species;
5      }
6  }
7
8  class Dog extends Animal {
9      private String breed;
10     public Dog(String breed) {
11         super("Canis␣lupus␣familiaris");
12         this.breed = breed;
13     }
14 }
```

Операция приведения типа

Операция динамического приведения объектной ссылки *obj* к типу T проверяет, является ли тип T одним из типов объекта *obj*, и возвращает *obj*, если является. В противном случае операция порождает исключение `ClassCastException`. В Java динамическое приведение записывается как `(тип)obj`. Пример:

```
1  class Animal {}
2
3  class Dog extends Animal {}
4
5  class Cat extends Animal {}
6
7  class Casts {
8      public static void main(String[] args) {
9          Animal a = new Dog(), b = new Cat();
10         Dog x = (Dog)a, y = (Dog)b;
11     }
12 }
```

Определение 20. *Переопределение метода* (method overriding) – это замена тела экземплярного метода базового класса в производном классе, позволяющая объекту производного класса по-другому обрабатывать то же самое сообщение.

Следует хорошо понимать разницу между переопределением и перегрузкой метода: при переопределении в классе не появляется новый метод, но изменяется реализация метода, унаследованного от базового класса.
Переопределение метода возможно только в случае, если для вызова этого метода используется позднее связывание, то есть если метод – виртуальный. Так как в языке Java все методы – виртуальные, то любой метод может быть переопределён.
В для переопределения метода, унаследованного от базового класса, достаточно объявить этот метод в производном классе.

Пример: переопределение метода


```
1  class Animal {
2      private String species;
3      public Animal(String species) {
4          this.species = species;
5      }
6      public void speak() {
7          System.out.println("--");
8      }
9  }
10
11 class Dog extends Animal {
12     private String breed;
13
14     public Dog(String breed) {
15         super("Canis␣lupus␣familiaris");
16         this.breed = breed;
17     }
18
19     public void speak() {
20         System.out.println("Bow-wow");
21     }
22 }
```

Определение 21. *Абстрактный метод* (abstract method) – это не имеющий тела виртуальный метод, который должен быть переопределён в производных классах.

В языке Java абстрактный метод объявляется с модификатором `abstract`, причём вместо тела метода ставится точка с запятой.

Определение 22. *Абстрактный класс* (abstract class) – это класс, имеющий абстрактные методы, которые либо объявлены в нём самом, либо унаследованы от базовых классов и не переопределены.

В программах на Java абстрактный класс объявляется с модификатором `abstract`. При этом экземпляры абстрактных классов создавать запрещено.

Пример: абстрактный класс


```
1  abstract class Animal {
2      private String species;
3
4      public Animal(String species) {
5          this.species = species;
6      }
7
8      public abstract void speak();
9  }
10
11 class Dog extends Animal {
12     private String breed;
13
14     public Dog(String breed) {
15         super("Canis lupus familiaris");
16         this.breed = breed;
17     }
18
19     public void speak() {
20         System.out.println("Bow-wow");
21     }
22 }
```

Объявление интерфейсов

Определение 23. *Интерфейс* (interface) – это тип данных, представляющий собой набор абстрактных методов.

Предназначение интерфейса – служить контрактом между разработчиком некоторого класса и разработчиком кода, использующего данный класс.
Аналог интерфейса – чистый абстрактный класс, в котором отсутствуют экземплярные поля и неабстрактные экземплярные методы. Создание экземпляров интерфейсов невозможно.
В Java для объявления интерфейса используется ключевое слово `interface`:

```
interface имя {
    ...
}
```

Внутри тела интерфейса перечисляются методы без модификаторов доступа и без тел.

Пример: объявление интерфейсов

Геометрическая фигура, имеющая координаты и способность перемещаться на плоскости:

```
1 interface MovableShape {
2     double x();
3     double y();
4     void move(double dx, double dy);
5 }
```

Вместо интерфейса `MovableShape` можно было бы объявить чистый абстрактный класс:

```
1 abstract class MovableShape {
2     public abstract double x();
3     public abstract double y();
4     public abstract void move(double dx, double dy);
5 }
```

Определение 24. *Реализация интерфейса* (interface implementation) – это класс, являющийся подтипом этого интерфейса. Если этот класс – неабстрактный, то в нём должны быть определены все без исключения методы интерфейса.

Класс может реализовывать сразу несколько интерфейсов. Тем самым, в Java интерфейсы частично компенсируют отсутствие множественного наследования. Синтаксически в Java реализуемые классом интерфейсы перечисляются после ключевого слова `implements`.

Пример:

```
class Ktulhu extends God
    implements BrainEater, NecronomiconCallee {
    ...
}
```

Наследование интерфейсов

Интерфейс может быть наследником других интерфейсов, которые перечисляются в его объявлении после ключевого слова `extends`.

Пример:

```
interface MovableShape {
    // ... см. ранее
}

interface ClosedShape {
    double perimeter();
    double area();
}

interface Polygon extends MovableShape, ClosedShape {
    boolean convex();  // Является ли выпуклым?
}
```

В отличие от классов, для интерфейсов реализовано множественное наследование.

Экземплярные и статические вложенные классы

Определение 25. Класс X называется *экземплярным вложенным классом* (inner class) в классе Y, если из методов класса X доступно внутреннее состояние объекта класса Y. При этом класс Y называется *объемлющим классом*.

Объекты вложенных классов имеют неявное поле, содержащее ссылку на объект объемлющего класса, и могут создаваться в экземплярных методах объемлющего класса, а также в экземплярных методах всех вложенных в этот объемлющий класс экземплярных классов.

Определение 26. Класс X называется *статическим вложенным классом* (static nested class) в классе Y, если из методов класса X доступны статические поля класса Y.

В Java объявление вложенного класса выглядит как обычное объявление класса, расположенное внутри тела объемлющего класса. При этом статический вложенный класс объявляется с модификатором `static`.

Пример: экземплярные вложенные классы


```
1  public class SingleLinkedList {
2      private Elem first;
3
4      private class Elem {
5          private int x;
6          private Elem next;
7
8          public Elem(int x) {
9              this.x = x;
10             next = first;
11             first = this;
12         }
13     }
14
15     public void add(int x) {
16         new Elem(x);
17     }
18
19     ...
20 }
```

В языке Java можно сделать поле класса, а также локальную переменную или формальный параметр метода *неизменяемыми*. Для этого при объявлении нужно использовать модификатор `final` и, кроме того, неизменяемое поле нужно инициализировать в конструкторе класса, а неизменяемую переменную – при её объявлении. Например:

```
1 class A {
2     final int s;
3     A(final int x, final int y) {
4         final int sum = x + y;
5         s = sum;
6     }
7 }
```

Определение 27. *Локальный класс* (local class) – экземплярный вложенный класс, объявленный внутри метода объемлющего класса и имеющий доступ к неизменяемым локальным переменным и параметрам этого метода.

Определение 28. *Анонимный класс* (anonymous class) – локальный класс, объявление которого совмещено с созданием его экземпляра.

Для объявления анонимного класса и создания его экземпляра используется специальная форма операции `new`:

```
new ИмяБазовогоКлассаИлиИнтерфейса(пар-ры констр.) {
    ... // Тело класса
}
```

Пример:

```
...
MovableShape s = new MovableShape() {
    private double x = /*...*/, y = /*...*/;
    double x() { return this.x; }
    double y() { return this.y; }
    void move(double dx, double dy) {
        this.x += dx;  this.y += dy;
    }
}
...
```

Определение 29. *Функциональный интерфейс* (functional interface) – интерфейс с единственным абстрактным методом.

Создание экземпляра анонимного класса, реализующего функциональный интерфейс, синтаксически может быть представлено в виде /emphлямбда-выражения, имеющего две формы:

```
1. (форм. параметры) -> тело_метода

2. (форм. параметры) -> выражение
```

Пример: функциональные интерфейсы и лямбда-выражения


```
1  interface IntBinaryExpr {
2      int eval(int x, int y);
3  }
4
5  public class Test {
6      public static void main(String[] args) {
7          IntBinaryExpr e = (x, y) -> x + y;
8
9          /* До Java 1.8 было бы:
10         IntBinaryExpr e = new IntBinaryExpr() {
11             public int eval(int x, int y) {
12                 return x + y;
13             }
14         };
15         */
16
17         System.out.println(e.eval(10, 20));
18     }
19 }
```

Зачем нужны обобщения? Пример: контейнерные классы

Определение 30. *Контейнерный класс* – это класс, объекты которого выступают в роли хранилищ других объектов.

В стандартной библиотеке языка Java существует большое количество контейнерных классов. Например:

- `java.util.ArrayList` – динамический массив;
- `java.util.Queue` – очередь FIFO;
- `java.util.TreeMap` – сбалансированное дерево;
- `java.util.HashMap` – хеш-таблица.

Начиная с Java 1.5 контейнерные классы в Java реализуются в виде обобщённых классов. Чтобы подойти к рассмотрению обобщённых классов, напишем контейнерный класс `Stack` и рассмотрим проблемы, возникающие из-за того, что этот класс – необобщённый.

Пример: необобщённый класс Stack


```
1  import java.util.Arrays;
2
3  public class Stack {
4      private int count = 0;
5      private Object[] buf = new Object[16];
6
7      public boolean empty() { return count == 0; }
8
9      public void push(Object x) {
10         if (count == buf.length)
11             buf = Arrays.copyOf(buf, buf.length*2);
12         buf[count++] = x;
13     }
14
15     public Object pop() {
16         if (empty())
17             throw new RuntimeException("underflow");
18         return buf[--count];
19     }
20 }
```

Пример: использование необобщённого класса Stack


```
Stack s = new Stack();
s.push(1);  s.push(2);  s.push(3);

int sum = 0;
while (!s.empty()) {
    int x = (Integer) s.pop();
    sum += x;
}
```

Обратите внимание на то, что попытка передать целое число методу `push`, которому требуется `Object` в качестве параметра, приводит к автоматическому заворачиванию целого числа в объект стандартного класса `Integer`. Кроме того, присваивание объекта класса `Integer` переменной `x` означает его автоматическое разворачивание.

То есть в Java имеется синтаксический сахар, без которого нам бы пришлось портить программу конструкциями вида

```
s.push(new Integer(1))...
...
int x = (Integer)(s.pop()).intValue();
```

Основная проблема при использовании необобщённых контейнерных классов

Перепишем код, использующий необобщённый класс `Stack`, следующим образом:

```
Stack s = new Stack();
s.push(1);  s.push(2);  s.push(3.14);

int sum = 0;
while (!s.empty()) {
    int x = (Integer) s.pop();
    sum += x;
}
```

В результате программа аварийно завершится с сообщением «java.lang.Double cannot be cast to java.lang.Integer». Действительно, мы «ошиблись» и положили число 3.14 в контейнер, в котором по логике программы должны быть только целые числа. При этом мы не получили от компилятора никаких сообщений, и программа «свалилась» только во время выполнения.

Понятие обобщённого класса

Определение 31. *Обобщённый класс* – это класс, имеющий формальные типовые параметры.

Определение 32. *Формальный типовый параметр* – это тип, который используется в объявлении обобщённого класса и при этом неизвестен во время компиляции обобщённого класса.

Типовые параметры в объявлении обобщённого класса перечисляются в угловых скобках после имени класса:

```
class Имя<параметр1, параметр2, ..., параметрN>
```

Например,

```
class Map<K, V>
```

Обратите внимание на то, что в качестве имён типовых параметров принято использовать заглавные латинские буквы.

Пример: обобщённый класс Stack


```
1  import java.util.Arrays;
2
3  public class Stack<T> {
4      private int count = 0;
5      private Object[] buf = new Object[16];
6
7      public boolean empty() { return count == 0; }
8
9      public void push(T x) {
10         if (count == buf.length)
11             buf = Arrays.copyOf(buf, buf.length*2);
12         buf[count++] = x;
13     }
14
15     @SuppressWarnings("unchecked")
16     public T pop() {
17         if (empty())
18             throw new RuntimeException("underflow");
19         return (T) buf[--count];
20     }
21 }
```

Пример: использование обобщённого класса Stack


```
Stack<Integer> s = new Stack<Integer>();
s.push(1);  s.push(2);  s.push(3);

int sum = 0;
while (!s.empty()) {
    int x = s.pop();
    sum += x;
}
```

Фактические типовые параметры указываются при использовании класса в угловых скобках после его имени и подставляются на место формальных типовых параметров.
Если теперь мы попытаемся добавить в стек `s` число 3.14, мы на этапе компиляции получим сообщение об ошибке «method push in class Stack<T> cannot be applied to given types».
Другим преимуществом обобщённого класса `Stack` является то, что теперь метод `pop` объекта `s` возвращает не `Object`, а `Integer`, и при инициализации переменной `x` отпадает необходимость в явном приведении типа.

Ограниченные обобщённые классы

Определение 33. *Ограниченный обобщённый класс* – это обобщённый класс, хотя бы для одного формального типового параметра которого задана верхняя граница.

Говорят, что для формального типового параметра задана его *верхняя граница*, если базовый класс для типового параметра – не `Object`. Другими словами, такому типовому параметру разрешено сопоставлять не любой класс, а некоторый класс X или любой наследник X.
Верхние границы для типовых параметров задаются в объявлении класса следующим образом:

```
class Имя<..., параметр extends X, ...>
```

Задание верхней границы типового параметра классом X позволяет вызывать методы класса X из тела обобщённого класса. Например, попробуем ограничить параметр класса `Stack` классом `Number`, от которого наследуют все числовые классы Java.

Пример: обобщённый ограниченный класс Stack


```
1  public class Stack<T extends Number> {
2      private int count = 0;
3      private Number[] buf = new Number[16];
4
5      ... // методы empty, push и pop -- без изменений
6
7      @SuppressWarnings("unchecked")
8      public T max() {
9          if (empty())
10             throw new RuntimeException("empty");
11
12         Number max = buf[0];
13         for (int i = 1; i < count; i++) {
14             Number x = buf[i];
15             if (x.doubleValue() > max.doubleValue())
16                 max = x;
17         }
18         return (T) max;
19     }
20 }
```

Пример: использование ограниченного обобщённого класса Stack


```
Stack<Integer> s = new Stack<Integer>();
s.push(1);  s.push(2);  s.push(3);
System.out.println(s.max());
```

Отметим, что в реализации класса `Stack` мы объявили поле `buf` как массив `Number`'ов, так как компилятор проследит, чтобы фактическим параметром класса `Stack` выступал только наследник класса `Number`.
Например, если мы попробуем параметризовать класс `Stack` классом `String`:

```
Stack<String> s = new Stack<String>();
```

мы получим на этапе компиляции ошибку «type argument String is not within bounds of type-variable T».

Ковариантность массивов

В Java массивы *ковариантны*. Это означает, что если класс `S` – наследник класса `T`, то тип `S[]` является подтипом `T[]`. В частности, ковариантность массивов позволяет написать такой код:

```
Integer[] ints = new Integer [10];
Number[] numbers = ints;
```

Спрашивается, что будет, если попытаться поместить в `numbers` ссылку на объект, тип которого – не `Integer`?

```
numbers[0] = 3.14;
```

Компилятор не выдаст никаких ошибок, но во время выполнения программа аварийно завершится с исключением «java.lang.ArrayStoreException». Действительно, если бы удалось положить ссылку на `Double` в `numbers`, то эта ссылка очутилась бы в массиве `ints`, в котором должны находиться только ссылки на `Integer`.

Инвариантность обобщённых классов

Попробуем провести аналогичный эксперимент с обобщённым классом `Stack`:

```
Stack<Integer> ints = new Stack<Integer>();
Stack<Number> numbers = ints;
```

В результате мы получим ошибку компиляции:

```
error: incompatible types
  Stack<Number> numbers = ints;
                          ^
```

Дело в том, что обобщённые классы в Java *инвариантны*, т.е. если `G` – обобщённый класс, и класс `S` – наследник класса `T`, то классы `G<S>` и `G<T>` отношением наследования не связаны.
Отметим, однако, что если, например, `DerivedStack<T>` – наследник `Stack<T>`, то `DerivedStack<Integer>` – наследник `Stack<Integer>`.

Шаблоны обобщённых классов

Инвариантность обобщённых классов, на первый взгляд, ограничивает возможность полиморфной обработки объектов этих классов. Например, «наивно» написанный метод для вывода содержимого стека `Number`'ов будет непригоден для вывода стека `Integer`'ов:

```
public static void printStack(Stack<Number> stack) {
    while (!stack.empty()) {
        Number x = stack.pop();
        System.out.println(x);
    }
}
```

Определение 34. *Шаблон* обобщённого класса `G` – это неявный супертип для любого класса, порождённого из `G`. Шаблон получается параметризацией класса `G` специальным фактическим параметром «?».

Например, `Stack<?>`.

Особенности шаблонов

- Можно объявить переменную типа шаблон, но невозможно создать его объект.

```
Stack<?> x;                    // OK
x = new Stack<?>();            // Error!
```

Здесь под переменной мы понимаем не только локальные переменные, но и формальные параметры методов, поля, элементы массивов. В частности, можно создать массив с элементами типа шаблон:

```
Stack<?>[] a = new Stack<?> [10];
```

- Если тип переменной – шаблон некоторого обобщенного класса `G`, то этой переменной можно присвоить ссылку на любой объект класса `G`, независимо от того, какой фактический типовый параметр был передан классу `G` при создании объекта.

```
x = new Stack<Integer>();      // OK
x = new Stack<Float>();        // OK
```

Пример: полиморфизм на основе шаблонов


```
1  public class Test {
2      public static void printStack(Stack<?> stack) {
3          while (!stack.empty()) {
4              Number x = stack.pop();
5              System.out.println(x);
6          }
7      }
8
9      public static void main(String[] args) {
10         Stack<Integer> s = new Stack<Integer>();
11         s.push(1);  s.push(2);  s.push(3);
12         printStack(s);
13     }
14 }
```

Обратите внимание: метод `pop` шаблона `Stack<?>` возвращает `Number`, потому что `Number` – верхняя граница формального типового параметра обобщённого класса `Stack`.

Частичные шаблоны

Если обобщённый класс имеет несколько формальных типовых параметров, то возможно получить *частичный шаблон*, передав классу «?» вместо части фактических типовых параметров. Например,

```
class Map<K, V>
{
    ...
}

...

Map<Integer, ?> intMap;
intMap = new Map<Integer, String>();   // OK
intMap = new Map<Integer, Float>();    // OK
intMap = new Map<Double, String>();    // Error!
```

Последнее присваивание вызывает ошибку компиляции, потому что первый фактический параметр обобщённого класса `Map` – не «?» и не `Double`, а `Integer`.

Шаблоны, ограниченные сверху

Фактический параметр «?» шаблона имеет верхнюю границу, по умолчанию совпадающую с верхней границей соответствующего формального параметра обобщённого класса.

Например, верхней границей «?» в `Stack<?>` является `Number`. Поэтому метод `pop` для `Stack<?>` возвращает `Number`, так как какой бы тип не был реальным фактическим параметром класса `Stack`, его можно привести к `Number`.
Верхнюю границу «?» в шаблоне можно уточнить, используя следующую запись:

```
G<..., ? extends X, ...>
```

Естественно, уточнение верхней границы сузит количество подтипов шаблона. Например,

```
Stack<Float> floats = new Stack<Float>();
Stack<?> numbers = floats;                    // OK
Stack<? extends Integer> ints = floats;       // Error!
```

Методы, недоступные у шаблонов, ограниченных сверху

У шаблона, ограниченного сверху, недоступны методы, тип параметров которых соответствует «?» в шаблоне. Например, у шаблона `Stack<?>` недоступен метод `push`:

```
Stack<?> numbers = new Stack<Integer>();
numbers.push(10);  // Error!
```

Действительно, переменная `numbers` может содержать ссылку на `Stack<Float>`, и тогда передача методу `push` целого числа окажется некорректной.

Применительно к контейнерным обобщённым классам можно сказать, что использование их шаблонов, ограниченных сверху, не позволяет написать полиморфный метод, осуществляющий запись новых значений в объект контейнерного класса.

Шаблоны, ограниченные снизу

По умолчанию «?» в шаблонах ограничен сверху, однако можно, наоборот, оганичить его снизу, записав

```
G<..., ? super X, ...>
```

Подтипами такого шаблона будут все полученные из `G` классы, у которых в качестве соответствующего фактического типового параметра выступает `X` или любой суперкласс `X`. Например,

```
Stack<? super Integer> ints;
ints = new Stack<Integer>();   // OK
ints = new Stack<Number>();    // OK
ints = new Stack<Float>();     // Error!
```

У ограниченного снизу шаблона контейнерного класса можно вызывать методы для записи значений того класса, которым он ограничен:

```
Stack<? super Integer> ints = new Stack<Number>();
ints.push(10);                 // OK
```

Методы, недоступные у шаблонов, ограниченных снизу

У шаблона, ограниченного снизу, недоступны методы, тип возвращаемого значения которых соответствует «?» в шаблоне. Например, у шаблона `Stack<?>` недоступен метод `pop`:

```
Stack<? super Integer> ints = new Stack<Integer>();
ints.push(10);
int x = ints.pop();  // Error!
```

Действительно, переменная `ints` может содержать ссылку на `Stack<Number>`, и тогда метод `pop` может возвратить, например, `Float`.

Применительно к контейнерным обобщённым классам можно сказать, что использование их шаблонов, ограниченных снизу, не позволяет написать полиморфный метод, осуществляющий чтение значений из объекта контейнерного класса.

Пример: «Producer extends and Consumer super»

Одним из наследников класса `Number` является класс `BigInteger`, представляющий числа произвольной разрядности. В отличие от класса `Integer`, класс `BigInteger` может иметь наследников.

Напишем метод, который переписывает элементы стека `BigInteger`'ов в стек `BigInteger`'ов или `Number`'ов:

```
public static void pushReversed(
        Stack<? super BigInteger> dest,
        Stack<? extends BigInteger> src)
{
    while (!src.empty())
        dest.push(src.pop());
}
```

Метод воплощает принцип PECS: `src` (Producer) поставляет значения и ограничивается сверху, `dest` (Consumer) принимает значения и ограничивается снизу.

Типовые парметры у методов

Метод `pushReversed` из предыдущего примера плох тем, что ориентируется на списки `BigInteger`. Он полиморфен в том смысле, что в качестве параметра `src` ему можно передать список объектов производного от `BigInteger` класса, а в качестве `dest` – список объектов базового для `BigInteger` класса.

К счастью, методы в Java тоже могут иметь типовые параметры. Поэтому мы можем переписать `pushReversed` следующим образом:

```
public static <T extends Number> void pushReversed(
        Stack<? super T> dest, Stack<T> src)
{
    while (!src.empty())
        dest.push(src.pop());
}
```

Объявление и вызов обобщённого метода

Определение 35. *Обобщённый метод* – это статический или экземплярный метод класса, имеющий формальные типовые параметры.

В объявлении метода типовые параметры непосредственно предшествуют типу возвращаемого значения метода.
Вызов статического обобщённого метода записывается как

```
имя_класса.<факт. типовые параметры>имя_метода(...)
```

Вызов экземплярного обобщённого метода:

```
объект.<факт. типовые параметры>имя_метода(...)
```

Определение 36. *Нештатная ситуация* – это ситуация, в которой выполнение некоторого фрагмента кода программы оказывается по тем или иным причинам невозможно.

Определение 37. *Исключительная ситуация* – это нештатная ситуация, возникшая в силу внешних по отношению к программе причин.

Примеры: не открылся файл, произошло незапланированное закрытие сетевого соединения.

Определение 38. *Ошибочная ситуация* – это нештатная ситуация, возникшая из-за ошибки в коде программы.

Примеры: выход за границы массива; деление на ноль; динамическое приведение типа объектной ссылки к типу, которого объект не имеет; переполнение стека.

Перехват нештатных ситуаций

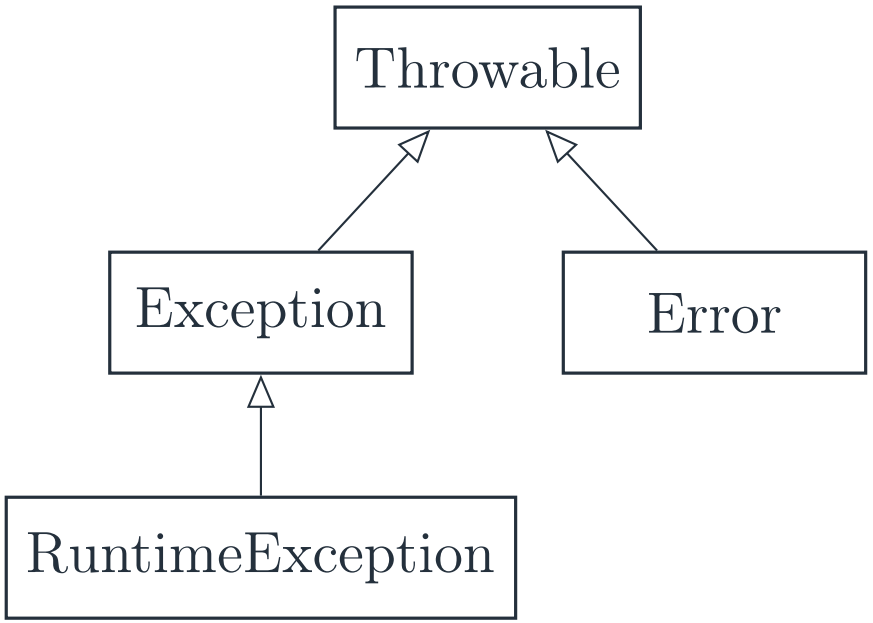
Определение 39. *Перехват нештатных ситуации* – это механизм, обеспечивающий продолжение работы программы при возникновении нештатной ситуации.

Два механизма перехвата нештатных ситуаций:

- Обработка кодов возврата.
  Функция, во время выполнения которой может возникнуть нештатная ситуация, возвращает некоторое значение, которое говорит о том, успешно или неуспешно функция выполнила свою задачу. Перехват ситуации заключается в том, что в коде, вызывающем такую функцию, стоят проверки ее возвращаемого значения.
- Обработка исключений.
  В случае возникновения нештатной ситуации генерируется так называемое исключение, описывающее нештатную ситуацию. Генерация исключения приводит к передаче управления на фрагмент кода программы, называемый обработчиком исключения.

Иерархия классов исключений

Определение 40. *Исключение* – это объект, описывающий нештатную ситуацию.



`Throwable` – базовый класс для всех классов исключений.
`Error` – базовый класс для классов исключений, описывающих «смертельные» для программы системные ошибочные ситуации (переполнение стека и т.п.).
`Exception` – базовый класс для всех классов несистемных исключений.
`RuntimeException` – базовый класс для классов несистемных исключений, описывающих ошибочные ситуации.

Методы класса Throwable

Основные конструкторы класса `Throwable`:

```
Throwable(String message)
Throwable(String message, Throwable cause)
```

Оба конструктора принимают в качестве параметра сообщение `message`, описывающее нештатную ситуацию. Второй конструктор, кроме того, позволяет организовать так называемую «цепочку искючений», когда исключение – `cause` – вложено в другое исключение. Здесь подразумевается, что вложенное исключение является причиной объемлющего исключения.

Основные методы класса `Throwable`:

```
Throwable getCause();
String getMessage();
String toString();
void printStackTrace();
```

Отметим, что при создании исключения в нём запоминается структура стека вызовов.

Пример: создание собственного класса исключения


```
1 public class SomeException extends Exception
2 {
3     public SomeException()
4     {
5         super("some message");
6     }
7 }
```

Отметим, что в качестве базового класса для создаваемого исключения можно выбрать `Throwable`, `Exception`, `RuntimeException`, `Error` или любой другой класс – подкласс класса `Throwable`.

Тем не менее, наиболее типично наследовать собственные классы исключений от `Exception` или `RuntimeException` в зависимости от того, какой тип нештатных ситуаций описывается исключением.

В силу особенностей реализации обобщений в Java, классы исключений не могут быть обобщёнными.

Неперехваченные исключения

Порождённое и неперехваченное исключение приводит к аварийному завершению программы с выдачей дампа стека вызовов, записанного в исключении. Например, рассмотрим программу, осуществляющую деление на ноль:

```
1 public class Test
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(10/0);
6     }
7 }
```

Результат запуска программы:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:6)
```

Операторы перехвата исключений

Синтаксис перехвата исключений в Java выглядит следующим образом:

```
try {                                   // try-блок
    /* код, в котором может возникнуть
    нештатная ситуация */
    ...
}
catch (SomeException e) {               // catch-блок
    /* обработчик исключений, которые можно привести
    к типу SomeException */
    ...
}
catch (SomeException2 e) {              // catch-блок
    ...
}
... /* другие catch-блоки */ ...
finally {                               // finally-блок
    /* код, который должен вызываться
    при любом выходе из try-блока */
    ...
}
```

Пример: перехват исключения


```
1  public class Test
2  {
3      public static void main(String[] args)
4      {
5          try {
6              System.out.println(10/0);
7          }
8          catch (ArithmeticException e) {
9              System.out.println(e.getMessage());
10         }
11         finally {
12             System.out.println("exit␣from␣try");
13         }
14         System.out.println("end␣of␣program");
15     }
16 }
```

Результат запуска программы:

```
/ by zero
exit from try
end of program
```

Порождение исключения

Порождение исключения выполняется оператором

```
throw исключение;
```

Например,

```
throw new SomeException();
```

Если класс исключения не является подклассом классов `Error` и `RuntimeException`, то для любого порождённого исключения в коде программы должен быть предусмотрен обработчик. Это требование синтаксически выражается следующим образом: если в процессе выполнения некоторого метода `m` могут порождаться исключения `x`, `y` и `z`, и эти исключения в теле метода не перехватываются, то объявление метода должно выглядеть как

```
тип m(параметры) throws x, y, z
{
    ...
}
```

Пример: неправильное порождение исключения


```
1  public class X
2  {
3      void m1()
4      {
5          throw new SomeException();
6      }
7  }
```

Компилятор выдаёт ошибку:

```
X.java:5: error: unreported exception SomeException;
must be caught or declared to be thrown
                  throw new SomeException();
```

Ошибка исправляется следующим образом:

```
1  public class X
2  {
3      void m1() throws SomeException
4      {
5          throw new SomeException();
6      }
7  }
```

Понятие функтора

Определение 41. *Функтор* – это обобщённый класс, параметризованный типом `T` и инкапсулирующий значение(-я) типа `T`, удовлетворяющий следующим условиям:

- объекты функтора – неизменяемые;
- у функтора имеется метод `map`, получающий в качестве параметра функцию (т.е. лямбда-выражение) преобразования значений типа `T`;
- метод `map` применяет функцию к значению(-ям), лежащему(-им) внутри функтора, и возвращает новый объект функтора, в котором инкапсулирован(-ы) результат(-ы) применения функции;
- если передать методу `map` функцию, которая просто возвращает значение своего аргумента (функция-идентичность), то единственным эффектом вызова метода `map` будет порождение объекта функтора, идентичного исходному объекту.

Пример функтора: корни уравнения

Рассмотрим функтор `Roots`, инкапсулирующий то или иное представление корней некоторого уравнения:

```
public class Roots<T> {
    private HashSet<T> container;

    private Roots(HashSet<T> container) {
        this.container = container;
    }
    ...
}
```

`T` – это тип одного корня. Мы не делаем никаких предположений об этом типе и храним корни в множестве, т.к. полагаем, что у уравнения все корни различны.
Конструктор функтора `Roots` – приватный, т.к. объекты функтора будут порождаться либо методами, решающими уравнения, либо методом `map`.

Порождение объекта функтора Roots через решение уравнения

Пусть в функторе `Roots` будет метод `of`, порождающий объект функтора путём решения квадратного уравнения:

```
public class Roots<T> {
    ...
    public static Roots<Double> of(
            double a, double b, double c, double eps){
        HashSet<Double> roots = new HashSet<>();
        if (a == 0.0) {
            if (b != 0.0) roots.add(-c/b);
        } else {
            double d = b*b - 4*a*c;
            if (d >= 0.0) {
                if (d < eps) d = 0.0;
                roots.add((-b+Math.sqrt(d)) / (2*a));
                roots.add((-b-Math.sqrt(d)) / (2*a));
            }
        }
        return new Roots<>(roots);
    }
}
```

Реализация метода map в функторе Roots

Метод `map` применяет функцию к корням, лежащем в объекте функтора:

```
public class Roots<T> {
    ...
    public <R> Roots<R> map(Function<T, R> f) {
        HashSet<R> c = new HashSet<>();
        for (T t : container) c.add(f.apply(t));
        return new Roots<>(c);
    }
    ...
}
```

Функциональный интерфейс `Function` входит в стандартную библиотеку Java и определяется как

```
public interface Function<T,R> {
    R apply(T t);
    ...
}
```

Перебор корней в функторе Roots

Мы обеспечим доступ к корням через метод `forEach`, который применяет не возвращающую значения функцию ко всем корням:

```
public class Roots<T> {
    ...
    public void forEach(Consumer<T> f) {
        for (T t : container) f.accept(t);
    }
    ...
}
```

Функциональный интерфейс `Consumer` из стандартной библиотеки Java определён как

```
public interface Consumer<T> {
    void accept(T t);
    ...
}
```

Демонстрация использования функтора Roots: 1

Решим задачу: камень брошен с поверхности Земли со скоростью v под углом φ к горизонту. На каком расстоянии от точки бросания он достигнет высоты h?
Нам понадобится вспомогательный класс `Point`:

```
public class Point {
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double dist() {
        return Math.sqrt(x*x + y*y);
    }

    public String toString() {
        return String.format("(%f, %f)", x, y);
    }
}
```

Демонстрация использования функтора Roots: 2

Решение квадратного уравнения

$$-gt^2/2 + vt\sin\varphi - h = 0$$

проведём в методе `main` класса `Stone`:

```
public class Stone {
    private static final double G = 9.81;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double v = sc.nextDouble();
        double fi = sc.nextDouble() * Math.PI/180;
        double h = sc.nextDouble();
        Roots.of(-G/2, v*Math.sin(fi), -h, 1e-10)
             .map(t -> new Point(t*v*Math.cos(fi), h))
             .map(pt -> pt.dist())
             .forEach(x -> System.out.println(x));
    }
}
```

Введение в монады на примере класса Roots

Что, если мы хотим использовать функтор `Roots` для хранения корней уравнения четвёртой степени вида

$$a(x^2 + px)^2 + b(x^2 + px) + c = 0?$$

Это уравнение решается в два этапа, на первом из которых мы переходим к переменной

$$y = x^2 + px.$$

Запишем фрагмент программы, решающий уравнение:

```
Roots<Roots<Double>> roots =
    Roots.of(a, b, c, 1e-10)
          .map(y -> Roots.of(1.0, p, -y, 1e-10));
```

Очевидно, что уравнение имеет до четырёх корней типа `double`, которые было бы удобно хранить в объекте типа `Roots<Double>`, а не `Roots<Roots<Double>>`.

Добавление метода flatMap в класс Roots

Избавиться от обозначенной на предыдущем слайде проблемы можно, превратив функтор `Roots` в *монаду* путём добавления метода `flatMap`:

```
public class Roots<T> {
    ...
    public <R> Roots<R> flatMap(
            Function<T, Roots<R>> f) {
        HashSet<R> c = new HashSet<>();
        map(f).forEach(rs -> c.addAll(rs.container));
        return new Roots<>(c);
    }
    ...
}
```

Метод `flatMap` «уплощает» корни, объединяя множества корней из нескольких объектов класса `Roots` в одно множество.

Использование метода flatMap

Теперь мы можем получить сразу все корни нашего уравнения четвёртой степени в виде `Roots<Double>`:

```
Roots<Double> roots =
    Roots.of(a, b, c, 1e-10)
         .flatMap(y -> Roots.of(1.0, p, -y, 1e-10));
```

Выводы, которые можно сделать из примера:

- функторы и монады предназначены для лаконичной записи последовательности преобразований данных;
- лаконичность записи достигается сокрытием громоздких управляющих конструкций языка Java (операторов выбора и циклов) внутри методов функторов и монад;
- благодаря тому, что функторы и монады, а также их методы – обобщённые, в процессе преобразований может меняться тип данных.

Определение 42. *Монада* – это функтор с дополнительным методом `flatMap`, удовлетворяющий следующим условиям:

- метод `flatMap` получает в качестве параметра функцию, способную преобразовать каждое значение внутри монады и завернуть результат преобразования в новый объект-монаду;
- метод `flatMap` применяет функцию ко всем значениям, хранящимся внутри монады, и объединяет получившиеся объекты-монады в один объект;
- метод `flatMap` ассоциативен, т.е. выражение

```
m.flatMap(f).flatMap(g)
```

эквивалентно выражению

```
m.flatMap(x -> f(x).flatMap(g))
```

Проблема нулевых ссылок

Огромную проблему при программировании на Java создаёт наличие в языке нулевой объектной ссылки – `null`:

- Нулевая ссылка широко используется в программах на Java как индикатор отсутствия значения.
  Например, метод `get` контейнерного класса `HashMap<K, V>`, объявленный как

```
public V get(Object key)
```

  возвращает `null` в случае, если словарная пара с ключом key отсутствует в хеш-таблице.
- Предполагается, что программист должен помнить, что некоторая переменная может содержать нулевую ссылку, и вставлять в код программы соответствующие проверки при необходимости обращения к объекту, на который указывает эта переменная.
- Легко забыть о необходимости проверок, и тогда – `NullPointerException`.

«Заворачивание» значения в Optional<T>

Объект монады `Optional<T>` является контейнером для нуля или одного объекта класса `T`.
Подразумевается, что объект `Optional<T>` находится в одном из двух состояний:

- является пустым;
- содержит ненулевую ссылку на объект класса `T`.

Создание объекта `Optional<T>`:

- создание пустого объекта:

```
static <T> Optional<T> empty()
```

- заворачивание ненулевой объектной ссылки:

```
static <T> Optional<T> of(T value)
```

- заворачивание объектной ссылки с порождением пустого объекта, если ссылка – нулевая:

```
static <T> Optional<T> ofNullable(T value)
```

«Разворачивание» Optional<T>: 1

«Развернуть» объект `Optional<T>` можно с путём вызова метода `get`:

```
T get()
```

В случае пустого объекта метод `get` порождает исключение `NoSuchElementException`.
Перед вызовом метода `get` следует убедиться, что объект `Optional<T>` – непустой, с помощью метода `isPresent`:

```
boolean isPresent()
```

Например, пусть `table` – хеш-таблица, отображающая строки в целые числа. Тогда безопасное получение значения, связанного с ключом, может быть записано как:

```
Optional<Integer> i =
    Optional.ofNullable(table.get("qwerty"));
if (i.isPresent()) System.out.println(i.get());
```

«Разворачивание» Optional<T>: 2

Рассмотренный на предыдущем слайде сценарий работы с `Optional<T>` мало чем отличается от использования нулевых ссылок:

```
Integer i = table.get("qwerty"))
if (i != null) System.out.println(i);
```

Более идеоматичым является использование метода `ifPresent`:

```
void ifPresent(Consumer<? super T> consumer)
```

Если значение существует в объекте `Optional<T>`, метод `ifPresent` вызовет для него лямбда-выражение. Если объект `Optional<T>` – пустой, ничего не произойдёт.
Перепишем вышеприведённый пример:

```
Optional<Integer> i =
    Optional.ofNullable(table.get("qwerty"));
i.ifPresent(v -> System.out.println(v));
```

«Разворачивание» Optional<T>: 3

Отметим, что лямбда-выражение вида

```
v -> obj.method(v)
```

которое всего лишь вызывает существующий метод некоторого объекта, может быть записано более кратко в виде так называемой *ссылки на метод*, имеющей вид

```
obj::method
```

Принимая также во внимание, что в последнем примере переменная `i` не особенно нужна, мы можем оформить этот пример ещё более каноничным для Java 8 способом:

```
Optional.ofNullable(table.get("qwerty"))
    .ifPresent(System.out::println);
```

Пример: композиция частичных функций

Пусть необходимо выполнить композицию частичных функций

$$f\left(g\left(h\left(x\right)\right)\right).$$

Здесь каждая функция – это вызов некоторого метода.
Так как функции частичные, то они не определены для некоторых значений своих аргументов, т.е. могут возвращать нулевую ссылку. Но при этом каждая функция подразумевает, что передаваемое ей значение существует, т.е. выражается ненулевой ссылкой.
В качестве примера рассмотрим следующую ситуацию. Имеется три хеш-таблицы. Нужно:

- заглянуть в первую таблицу по ключу x и взять оттуда значение y;
- использовать y как ключ для второй таблицы и взять соответсвующее ему значение z;
- залезть в третью таблицу по ключу z.

Композиция частичных функций без монад


```
public static String compose(
        HashMap<String, Integer> a,
        HashMap<Integer, Float> b,
        HashMap<Float, String> c,
        String x)
{
    if (x != null) {
        Integer y = a.get(x);
        if (y != null) {
            Float z = b.get(y);
            if (z != null) {
                return c.get(z);
            }
        }
    }
    return null;
}
```

Как видим, нам потребовался каскад `if`'ов из-за того, что любой из трёх вызовов метода `get` может вернуть `null`.

Метод map

Метод `map` класса `Optional<T>` позволяет обойтись без проверок при записи композиции частичных функций:

```
<U> Optional<U> map(
        Function<? super T, ? extends U> mapper)
```

Метод `map` принимает лямбда-выражение, определяющее частичную функцию (она может возвращать `null`).
Схема работы метода `map`:

- если объект `Optional<T>` пустой, то метод `map` ничего не делает и просто возвращает пустой `Optional<U>`.
- в противном случае значение, записанное в объекте `Optional<T>`, передаётся в лямбда-выражение, и метод map возвращает `Optional<U>`, в который «завёрнут» результат вычисления лямбда-выражения.
- результат «заворачивается» по принципу работы метода `ofNullable`, т.е. если он представляет собой нулевую ссылку, то `Optional<U>` будет пустым.

Композиция частичных функций с использованием Optional<T>

Код приведённого выше метода `compose` можно переписать с использованием метода `map`.
При этом будет логично изменить тип возвращаемого методом compose значения на `Optional<String>`:

```
public static Optional<String> compose(
        HashMap<String, Integer> a,
        HashMap<Integer, Float> b,
        HashMap<Float, String> c,
        String x)
{
    return Optional.ofNullable(x)
                .map(a::get)
                .map(b::get)
                .map(c::get);
}
```

Потоки как развитие идеи итераторов

Интерфейс `Stream<T>` в каком-то смысле является развитием идеи итераторов, т.е. представляет последовательность некоторых значений, называемую *потоком*.
Получить объект `Stream<T>` можно из объекта любого контейтерного класса стандартной библиотеки языка Java путём вызова метода `stream`:

```
Stream<T> stream()
```

Вытащить все значения из потока `Stream<T>` можно с помощью метода `forEach`:

```
void forEach(Consumer<? super T> action)
```

Этот метод получает в качестве параметра лямбда-выражение и вызывает его для каждого значения из последовательности, представляемой потоком.

Преобразование последовательностей

Методы интерфейса `Stream<T>` позволяют преобразовывать последовательности:

- формирование потока, содержащего только те элементы исходного потока, которые удовлетворяют предикату:

```
Stream<T> filter(Predicate<? super T> predicate)
```

- формирование потока, представляющего последовательность значений, получаемую путём вызова функции для каждого элемента исходного потока:

```
<R> Stream<R> map(
    Function<? super T, ? extends R> mapper)
```

- конкатенация потоков, получаемых путём вызова функции для каждого элемента исходного потока:

```
<R> Stream<R> flatMap(
    Function<? super T,
             ? extends Stream<? extends R>> mapper
)
```

Пример: filter и map

В качестве примера использования преобразований `filter` и `map` приведём решение следующей задачи.
Пусть дан динамический массив `ArrayList<Integer>`.
Требуется вывести в стандартный поток вывода последовательность чисел, получаемую путём возведения двойки в степени, заданные числами из этого динамического массива.
При этом отрицательные степени должны пропускаться:

```
public static void powers(ArrayList<Integer> list) {
    list.stream()
            .filter(x -> x >= 0)
            .map(x -> 1 << x)
            .forEach(System.out::println);
}
```

Пример: flatMap

Чтобы продемонстрировать применение преобразования `flatMap`, возьмём целочисленную матрицу `ArrayList<ArrayList<Integer>>` и распечатаем последовательность её ненулевых элементов:

```
public static void nonZeroes(
        ArrayList<ArrayList<Integer>> m) {
    m.stream()
            .flatMap(row ->
                row.stream().filter(x -> x != 0)
            )
            .forEach(System.out::println);
}
```

2.2.2 Собирание последовательностей

Потребление значений из потоков выполняется так называемыми *коллекторами* – объектами классов, реализующими интерфейс `Collector`. Коллекторы предназначены для выполнения анализа потоков и записи результатов реализуемых потоками преобразований в объекты контейнерных классов.

Чтобы задействовать некоторый коллектор, следует вызвать метод `collect` потока:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

Начиная с определённого размера программной системы, для её структурирования начинает не хватать одной только декомпозиции на классы и требуется механизм пакетов:

- образуются группы взаимосвязанных классов, которые целесообразно как-то выделить в отдельные подсистемы и ограничить доступность некоторых классов рамками соответствующей подсистемы;
- возникает потребность упростить навигацию по исходным текстам программной системы (чтобы легче находить классы, которые по смыслу относятся к той или иной подсистеме);
- появляются трудности с изобретением уникальных имён классов (особенно в случае командной разработки).

Определение 43. *Пакет* – это контейнер классов, предоставляющий для них отдельное пространство имён и дополнительные возможности по управлению доступом.

Имена пакетов и квалифицированные имена классов

Определение 44. *Имя пакета* – это непустая последовательность идентификаторов, разделённых точками.

В именах пакетов не принято использовать заглавные буквы (разве что для аббревиатур) и допускается наличие символов подчёркивания. Например,

```
java.util.concurrent
org.omg.CORBA_2_3.portable
ru.bmstu.iu9
```

Определение 45. *Квалифицированное имя класса* – это имя, с помощью которого можно обращаться к публичному классу извне пакета, которому этот класс принадлежит.

Квалифицированные имена записываются как

```
имя_пакета.имя_класса
```

Например, `java.util.concurrent.ConcurrentLinkedQueue`.

Для создания пакета нужно поместить в начало каждого файла, содержащего исходный текст классов, которые вы планируете включить в проект, директиву

```
package имя_пакета;
```

Классы, объявленные в файлах, в которых отсутствует директива `package`, считаются принадлежащими безымянному *пакету по умолчанию* (default package).
Компилятор Java требует, чтобы файлы пакета размещались в файловой системе в каталоге, путь к которому относительно текущего каталога (в котором запущен компилятор) соответствовал имени пакета.
Например, файлы пакета `ru.bmstu.iu9` должны находиться в каталоге `ru/bmstu/iu9`.
Файлы, принадлежащие пакету по умолчанию, должны находиться в текущем каталоге.

Правила видимости для классов пакета и их членов

Напомним, что один java-файл может содержать не более одного публичного класса и произвольное количество непубличных классов.
Публичные классы можно использовать как внутри, так и снаружи пакета, а непубличные классы доступны из любого места внутри пакета, но извне не видны.
К членам классов пакета, объявленным без модификаторов доступа, можно обращаться из любого места внутри пакета, но запрещено обращаться снаружи пакета.

Импорт классов из другого пакета

Для того чтобы обратиться к классу, принадлежащему другому пакету, этот класс нужно *импортировать*. Существует три способа импорта класса.

1. Использование квалифицированного имени. Например,

```
java.util.HashSet<String> a =
    new java.util.HashSet<>();
```

2. Указание класса в директиве `import`:

```
import java.util.HashSet; // в начале файла
...
HashSet<String> a = new HashSet<>();
```

3. Использование директивы `import` для импорта всех классов пакета:

```
import java.util.*; // в начале файла
...
HashSet<String> a = new HashSet<>();
```

Импорт статических членов класса

Для упрощения доступа к статическим членам классов существует специальная форма директивы `import`, позволяющая импортировать их в текущий файл:

```
import static имя_пакета.имя_класса.имя_члена;
```

или

```
import static имя_пакета.имя_класса.*;
```

Например, вычисление значения $\cos\frac{\pi}{3}$, без импорта статических членов класса `Math` выглядящее как

```
double l = Math.cos(Math.PI/3);
```

можно переписать более лаконично:

```
import static java.lang.Math.*;
...
double l = cos(PI/3);
```