



IT-Academy.by

**Образовательный центр
программирования
и высоких технологий**



Антон Смалюк

под редакцией Артёма Перевозникова

ПРОГРАММИРОВАНИЕ НА JAVA

Учебное пособие

Введение	5
Установка Java.....	5
Установка IntelliJ IDEA.....	5
Создание простейшей программы и ее компиляция	5
Структура исходного файла Java.....	6
Задания	7
Типы данных. Переменные. Операторы. Методы	7
Данные в программе	7
Переменные.....	10
Операторы	12
Методы	18
Задания	19
Операторы if/switch	20
Оператор if	20
Оператор выбора switch	25
Задания	26
Циклы.....	28
Оператор циклов while	28
Оператор do while.....	29
Оператор цикла for	30
Операторы break и continue	31
Задания	32
Массивы	34
Одномерные массивы	34
Многомерные массивы	38
Задания	38
Классы и объекты	40
Переменные и методы в классах	40
Модификаторы доступа	41
Методы в классах	41
Создание классов и объектов	41
Конструкторы. Ссылка this	43
Статические и константные члены класса	44
Стандартные классы и объекты Java	46
Задания	48

Строки и регулярные выражения	49
Строки	49
Примеры работы со строками	51
Работа со строками String, StringBuffer и StringBuilder	52
Сходства и отличия StringBuffer и StringBuilder	52
Регулярные выражения	52
Задания	55
Наследование и полиморфизм. Внутренние и анонимные классы	57
Наследование и полиморфизм	57
Пакеты	59
Абстрактные классы	60
Интерфейсы	62
Внутренние классы	64
Анонимные классы	65
Задания	66
Дженерики. Перечисляемые типы	67
Дженерики	67
Перечисляемые типы	68
Задания	69
Коллекции. List, Set, Map	70
Коллекции типа List	71
Коллекции типа Set	72
Коллекции типа Queue	73
Карты (Map)	73
Использование дженериков в коллекциях	74
Использование циклов в коллекциях	74
Итераторы	75
Задания	76
Исключения	77
Основы	77
Ключевое слово throw	78
Обработка исключений	78
Ключевое слово throws	79
Задания	79
Работа с файлами	80

Чтение текстовых файлов	80
Чтение двоичных файлов	82
Запись в файл.....	83
Конструкция try-with-resources	84
Класс File.....	85
Задания	87
Класс Thread и интерфейс Runnable.....	89
Создание потоков	89
Задания	92
Взаимодействие потоков. Producer – Consumer.....	93
Синхронизация	93
Взаимодействие потоков	94
Deadlock	96
Livelock	97
Starvation	98
Задания	98
Пулы потоков. Асинхронные вычисления	99
Пулы потоков	99
Интерфейс Callable	100
Интерфейс Future	100
Задания	100
Lambda, Streams API	101
Основные понятия.....	101
Функциональный интерфейс	101
Ссылки на методы	102
Ссылки на конструкторы	102
Optional<T>	103
Stream API.....	103
Терминальные методы	103
Конвейерные методы	104
Задания	105
Date Time API	106
Основные пакеты.....	106
Класс java.time.LocalDate	106
Класс java.time.LocalTime.....	107

Вспомогательные методы Date API	107
Парсинг и форматирование даты	108
Задания	108
Принципы дизайна ПО. SOLID. Паттерны	109
Понятие шаблона проектирования	109
Singleton	110
Метод Фабрика	111
Строитель	111
Команда.....	113
Задания	114
Reflection API. Аннотации.....	115
Reflection API	115
Аннотации	116
Задания	119
Основы XML/JSON.....	119
Понятие XML, достоинства и недостатки	119
Основные элементы документа XML, понятие тега.....	120
Структура документа XML	121
DTD, схема XML и обзор XSD	122
DOM и SAX и StAX parser	122
XSL/XSLT	126
Понятие JSON.....	126
Синтаксис JSON	126
Пример	126
Задание	128

IT-Academy

Введение

Java представляет собой объектно-ориентированный язык программирования. На данный момент он является одним из наиболее распространенных языков программирования для сложных проектов и Интернет-приложений.

Java не является полностью компилируемым языком. Компилятор переводит его исходный текст не в исполняемый файл в машинных кодах, а в специальный байт-код, который затем исполняется с помощью специальной программы, называемой виртуальной машиной (JVM). Достоинством такого подхода является то, что большинство программ Java после компиляции может исполняться на любом компьютере и на любой ОС, для которой создана виртуальная машина Java.

Следует признать, что данный подход имеет и свои недостатки. Например, из-за необходимости запуска виртуальной машины, даже сравнительно небольшие программы могут требовать значительных объемов памяти для исполнения.

Установка Java

Скачать установщик можно по ссылке: <https://www.oracle.com/java/technologies/javase-jdk11-downloads.html>, выбрав соответствующую операционную систему.

Альтернативные ссылки для скачивания:

- <https://jdk.java.net/java-se-ri/11>
- <https://adoptopenjdk.net/>

Нужно запустить установщик и следовать указаниям. Рекомендуемая папка для установки d:/opt/java/jdk

Далее необходимо установить переменную окружения JAVA_HOME, которая должна указывать на директорию, в которую установлена Java. Многие программы используют эту переменную, чтобы определить, где находится Java.

В переменную окружения PATH добавить путь к директории %JAVA_HOME%\bin. Эта переменная указывает операционной системе список директорий, в которых нужно искать исполняемые файлы, и чтобы можно было запускать Java из консоли, переменная PATH должна быть правильно настроена.

Установка IntelliJ IDEA

Скачать установщик можно по ссылке: <https://www.jetbrains.com/idea/download>, выбрав соответствующую операционную систему.

Нужно запустить установщик и следовать указаниям. Рекомендуемая папка для установки программы d:/opt/idea

Создание простейшей программы и ее компиляция

Чтобы создать простейшую программу на Java вы должны первым делом создать текстовый файл (либо в текстовом редакторе, либо в IntelliJ IDEA) с именем и расширением HelloWorld.java, и следующим содержанием:

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

```
        System.out.println("Hello World!");  
    }  
  
}
```

Следующим шагом необходимо набрать в консоли команду вида:

```
javac HelloWorld.java
```

Помните, что для успешного выполнения этой команды вы должны находиться в том же каталоге, что и ваш файл.

В результате рядом с вашим файлом должен появиться другой файл с таким же именем и с расширением .class. Если он был успешно создан, программу можно запускать. Делается это командой:

```
java HelloWorld
```

Если вы не допустили ошибок, в ответ на вашу команду будет выведено:

```
Hello World!
```

Это значит, что программа была успешно скомпилирована и запущена.

Структура исходного файла Java

Java представляет собой объектно-ориентированный язык. Это означает, что программы на нем состоят из объектов – программных элементов, содержащих информацию о каком-то реальном или воображаемом объекте. Например, в программе, выполняющей задачу расстановки мебели в помещении, могут существовать объекты, описывающие столы, стулья, шкафы, стены самого помещения и т.д. В программе, работающей с компьютерной графикой, могут быть объекты, описывающие точки, линии, окружности и т.д.

Просто так начать создавать объекты нельзя. Сначала необходимо создать описание объекта, которое называют классом.

Класс часто называют сборочным чертежом объекта. Именно в классе описывается, из чего он состоит, какие свойства имеет и что он может делать. Обычно класс выглядит следующим образом:

```
class ИмяКласса {  
    содержимое класса  
}
```

В примере, который мы компилировали ранее, имя класса выглядело как:

```
HelloWorld
```

Имя класса задается с учетом следующих требований:

- допустимы латинские буквы;
- допустим знак подчеркивания;
- допустимы цифры;
- имя нельзя начинать с цифры, а также нежелательно со знака подчеркивания

Имя файла, в котором находится такой класс, должно совпадать с именем класса, причем регистр (то есть маленькие и большие буквы) тоже.

Задания

Задание 1

Набрать приведенный пример HelloWorld.java, откомпилировать его и запустить.

Для заметок:

Задание 2

В набранном из задания 1 примере изменить сообщение на “Hello Belarus!”. Откомпилировать пример и запустить.

Для заметок:

Задание 3

В набранном из задания 2 примере добавить к основному сообщению дополнительное “I like Java!”. Откомпилировать пример и запустить.

Для заметок:

Типы данных. Переменные. Операторы. Методы

Данные в программе

Перед тем, как узнать, что и как записывается в содержимом класса, нам необходимо познакомиться с тем, как записываются данные в программах. Java позволяет работать со следующими видами данных:

- целые числа;
- вещественные числа;
- строки;
- логические значения;
- специальные значения.

Числовые данные

Чаще всего используются обычные целые десятичные числа, которые могут быть как положительными, так и отрицательными:

```
10
-2
638
```

Следует помнить, что первый символ таких чисел не может быть 0.

Этому ограничению есть простое объяснение: в Java ноль в начале числа обозначает, что это не десятичное, а восьмеричное число. Если в тексте программы записать 020 – это будет означать, что используется восьмеричная система счисления и в привычной нам десятичной системе это число будет иметь значение шестнадцать. Если же записать число 019 – это будет ошибка, так как цифры 8 и 9 в восьмеричной системе не используются.

Имеется также возможность записывать числа в шестнадцатеричной системе счисления. В этом случае число должно начинаться с 0x. Шестнадцатеричная система имеет цифры от 0 до 15. Цифры начиная с 10 обозначаются буквами латинского алфавита, то есть 10 – a, 11 – b, 12 – c, 13 – d, 14 – e, 15 – f. Если число начинается с 0x, цифры надо обозначать строчными латинскими буквами. Число также может начинаться с 0X. В этом случае желательно использовать прописные буквы.

Вещественные числа или, говоря по-другому, десятичные дроби могут записываться двумя способами. Первый, обычный, состоит из целой и дробной частей, разделенных точкой:

```
1.5
-2.4
638.165
```

Следует помнить, что в качестве разделителя используется не запятая, а точка, принятая в американском стандарте записи.

Второй способ записи – так называемый "с плавающей запятой" или "научный" записывается следующим образом:

```
1.5E4
12e-3
```

Число состоит из первой части, умноженной на степень десятки, стоящей после E, то есть:

```
1.5E4 = 1.5*104 = 15000
12e-3 = 12*10-3 = 0.012
```

Символьные значения

Символьные значения записываются в виде одного символа, взятого в одинарные кавычки:

```
'x'
```

Символьные данные – одно из немногих мест программы, в котором позволяет использовать буквы не латинских алфавитов, в частности, кириллицы.

Логические значения

Обычно существует только два логических значения – "истина" и "ложь". В языке Java они записываются как true и false соответственно.

Специальные значения

Самым распространенным специальным значением является null, обозначающее пустое значение, то есть отсутствие какого-нибудь значения. В некоторых языках программирования null считается совпадающим с нулем, но в Java эти два значения считаются абсолютно разными.

Строковые данные

Практически любая программа должна уметь работать с текстом. Для этого используются строковые данные. Строка – фрагмент текста, то есть набор символов. Для того, чтобы отделить строковые данные от остального текста программы, строки берутся в двойные кавычки. Вот примеры строк:

```
"Число Пи равно 3.14159"
```

```
"Hello World!"
```

Строка может содержать практически любые знаки, в том числе буквы латинского и русского алфавитов, цифры, знаки препинания, специальные символы и т.д.

Всегда следует четко отличать строковые и символьные данные. Символьные данные могут содержать только один символ, взятый в кавычки. В строковых данных количество символов может быть любым.

Иногда следует записать пустую строку, то есть строку, не содержащую никаких символов. Для этого следует поставить две кавычки подряд: "".

Надо отличать пустую строку от строки с пробелом: " ". Пробел также является символом и поэтому такая строка не является пустой.

Некоторые символы не могут быть напрямую набраны на клавиатуре, при вводе исходного текста программы. Например, перевод строки будет восприниматься как переход на следующую строку программы, а не как перевод строки внутри строковых данных. Это же касается табуляций и некоторых других знаков. Для решения этой проблемы были созданы так называемые спецсимволы. Чтобы вставить в текст спецсимвол, сначала надо вставить символ \ (символ экранирования), а затем букву спецсимвола.

Могут использоваться следующие сочетания:

\n – перевод строки;

\t – табуляция;

\r – возврат каретки.

Например, текст "Hello \nworld!" будет воспринят как:

```
Hello
```

```
world!
```

Спецсимволы также являются единственным случаем, когда внутри одинарных кавычек может стоять несколько знаков, так как их комбинация представляет собой один знак. Так запись '\t' является вполне допустимой.

Спецсимволы позволяют решить еще одну проблему. Иногда в тексте встречаются кавычки. Если набрать их напрямую, возникнет ошибочная ситуация, например:

"Надо придумать синонимы к слову "потерять", и записать их"

В данном примере при выполнении программы данная строка будет воспринята как две разные строки, между которыми находится слово потерять и возникнет ошибка. Можно использовать спецсимволы, чтобы вставить кавычки в текст строки. В этом случае строку следует записать следующим образом:

"Надо придумать синонимы к слову \"потерять\", и записать их"

То есть, там, где надо вставить в текст кавычку, следует писать сочетание \".

Например, если понадобится строка, содержащая тег `<table cellpadding="5" cellspacing="0">` ее следует записать следующим образом:

`<table cellpadding=\"5\" cellspacing=\"0\">`

Так как символ косой черты – "слеш" используется для спецсимволов, просто так его в текст вставить нельзя. Для вставки в строку косой черты следует использовать спецсимвол – двойную косую черту \\. .

Переменные

Понятие переменной

При выполнении различных операций значения могут меняться. Чтобы хранить меняющиеся значения, используются переменные (по-другому их называют идентификаторы).

Переменная – область памяти, в которой хранится значение. Часто говорят, что переменная – именованная область памяти. То есть у каждой переменной есть свое имя, чтобы отличать их друг от друга. Имя переменной используется, чтобы указать, что используется именно это значение и никакое другое.

Java, как и его предки C/C++, является языком со строгой типизацией. Это означает, что тип значения, находящийся в переменной, задается при ее создании и не может изменяться. Если в переменной может находиться целое число, потом туда нельзя занести логическое значение, а потом строку.

Правила задания имен переменных

Имена переменных задаются по тем же правилам, что и имена классов (эти правила вообще относятся к большинству имен в программах на Java):

- допустимы латинские буквы;
- допустим знак подчеркивания;
- допустимы цифры;
- имя нельзя начинать с цифры, а также нежелательно со знака подчеркивания.

Имена переменных и вообще все имена в Java являются регистр зависимыми. То есть переменные с именами Xtest и xtest – разные переменные.

Создание новых переменных

В Java требуется создавать переменные для их последующего использования. Делается это так:

тип имя;

Например:

```
int xCoordinate;
```

В данном случае создается переменная `xCoordinate`, предназначенная для хранения целых значений.

При создании переменной в нее сразу можно занести значение. Для этого достаточно после имени поставить знак равно и написать требуемое значение.

Например:

```
int xCoordinate = 10;
```

или

```
var xCoordinate = 10;
```

где `var` – это зарезервированное слово.

В данном случае переменная изначально будет хранить значение 10. Если этого не сделать, значение будет равно 0.

Типы переменных

Для переменных имеются следующие типы:

- `boolean` – логический тип. Переменные такого типа могут хранить только логические значения;
- `char` – символьный тип. Переменные такого типа могут хранить символы;
- `byte` – целочисленный тип;
- `short` – целочисленный тип;
- `int` – целочисленный тип;
- `long` – целочисленный тип;
- `float` – вещественный тип;
- `double` – вещественный тип.

От выбора типа данных может зависеть скорость вашей программы, а также объем занимаемой ею памяти. Так на 32-разрядных системах операции с типом `long` могут требовать значительно больше времени, чем с типом `int`.

При записи числовых данных, в тексте программы, целые числа автоматически считаются типом `int`, а дробные – `double`. Иногда необходимо дать числу другой тип, например, сделать дробное число `float`, в этом случае используется так называемый суффикс, то есть буква, которая ставится в конце числа для обозначения его типа.

Например, чтобы дробное число было `float`, следует записать его как `1.5f`. Чтобы целое число было `long`, следует его записать с буквой `l`, например: `1234567l`

Таблица значений для типов

Тип	Разрядность	Диапазон	Значение по умолчанию
byte	8 бит	[-128;127]	0
short	16 бит	[-32768; 32767]	0
int	32 бит	[-2147483648; 2147483647]	0
long	64 бит	$[-2^{63}; 2^{63}]$	0L
float	32 бит	$\sim 1,4 \cdot 10^{-45}$ до $\sim 3,4 \cdot 10^{38}$	0.0f
double	64 бит	$\sim 4,9 \cdot 10^{-324}$ до $\sim 1,8 \cdot 10^{308}$	0.0d
char	2 байта	Натуральные из [0;65535], интерпретируются как коды символов по таблице Unicode	'\u0000'
boolean	Для хранения значения этого типа достаточно 1 бита, но в реальности память такими порциями не выделяется, поэтому переменные этого типа могут быть по-разному упакованы виртуальной машиной	false либо true	false

Операторы

Арифметические операторы

В большинстве языков программирования команды языка называются операторами. Java имеет операторы для выполнения основных арифметических операций:

+	Сложение
-	Вычитание
*	Умножение
/	Деление

Арифметические операторы записываются аналогично обычным математическим выражениям:

```
2 + 2
3 * 15
3 - 5 * 4
```

Отдельно следует упомянуть оператор % – остаток от целочисленного деления. В результате выполнения этого оператора получается остаток, например, 7%5 вернет остаток равный 2.

Как и в математических выражениях, последовательность действий зависит от вида оператора. В последнем примере сначала пять умножается на четыре, а затем результат вычитается из трех. Когда говорят о том, что один оператор должен выполняться раньше другого, это называют "приоритетом". Приоритет оператора умножения выше, чем оператора вычитания, поэтому он выполняется первым.

Также как и в математике, последовательность действий можно изменить, используя круглые скобки:

```
(3 - 5) * 4
```

В данном случае сначала выполняется вычитание, а затем умножение.

Но в отличие от математики, знаки операторов не могут пропускаться. В математике допустима запись $a(b+c)$, а в программировании все прописывается полностью: $a * (b + c)$.

Разумеется, в математических выражениях могут участвовать не только числа, но и переменные, которые содержат числа.

```
2 + xCoordinate
```

В данном случае двойка складывается со значением, находящимся в переменной `xCoordinate`. Если, например, в переменной находится 10, результатом выражения будет 12.

Операторы сравнения

Кроме математических операторов, в Java имеются операторы сравнения, позволяющие сравнить два значения, и получить результат в виде логического значения, то есть "истина" или "ложь".

Оператор `==` позволяет сравнить два значения на равенство. Если значения, стоящие по обе стороны от него, равны, результатом будет истина, если они не равны, результатом будет ложь. Например:

```
x == 10
x == y
y * 2 == x - 10
```

В первом случае результатом будет истина, если значение переменной `x` равно десяти. Во втором случае результатом будет истина, если значения в переменных `x` и `y` совпадают. В третьем случае результатом будет истина, если совпадают результаты выражений справа и слева.

Как видно из примера, сравниваться могут и значения, записанные непосредственно в тексте программы, и находящиеся в переменных, и результаты выражений.

Иногда требуется сравнивать не на равенство, а на неравенство. В таких случаях используется оператор "не равно" `!=`

```
x != 10
```

Кроме сравнения на равенство, существуют также операторы "меньше" `<` и "больше" `>`. Результатом оператора больше будет истина, если значение слева больше значения справа, а у оператора меньше – наоборот.

```
x > 10
x < y
```

В первом случае результатом будет истина, если в переменной `x` находится число больше 10. Во втором случае результатом будет истина, если в переменной `x` находится значение меньшее, чем в `y`.

Есть также операторы "больше либо равно" `>=` и "меньше либо равно" `<=`

Логические операторы

Далеко не всякое условие может быть описано обычными операторами сравнения. Например, математическое условие $1 < x < 5$ будет истинным в том

случае, когда x одновременно больше единицы и меньше пяти, то есть его значение попадает в промежуток между пятью и единицей. Большинство языков программирования такую запись не понимают, они либо выдают ошибку, либо неправильный результат.

Чтобы записать это условие по правилам Java, следует разбить его на два отдельных: $1 < x$ и $x < 5$. Результат выражения должен быть истинным, только если и первое, и второе условия будут истинны. Для этого их следует соединить оператором "логическое и". Данный оператор записывается, как `&&` и его результат является истинным, только если оба выражения, стоящие слева и справа от него, являются истинными. Мы получаем запись:

```
1 < x && x < 5
```

Результат выражения будет истиной, если и первое, и второе условие будут истинными. Если хотя бы одно из условий является ложным, выражение будет ложным.

Существует также оператор "логическое или", который записывается как две вертикальные черты `||`. Результат этого оператора будет истинным, если хотя бы одно из выражений, стоящее справа и слева от него, будет истинным. Ложным он будет, только если оба выражения ложны. Например, нам надо проверить, что значение переменной не попадает в промежуток. Это можно записать следующим образом:

```
5 < x || x < 1
```

То есть значение не попадает в промежуток, если оно меньше 1 или больше 5.

С помощью логических операторов можно соединять не только простые условия, но и другие логические выражения. В логических выражениях также можно ставить скобки, чтобы управлять последовательностью действий.

Ещё к логическим операторам относится оператор "не" `!`. В отличие от логических "и" и "или", он является унарным, и ставится перед условием (если это сложное условие, его следует взять в скобки).

Например:

```
!(1 < x)
```

Результат будет истиной, если x будет меньше либо равен 1.

Оператор присваивания

Во многих случаях вычисление результата выражения само по себе не имеет смысла. Он появляется, если результат можно где-нибудь сохранить, например, в переменной. Для этих целей используются операторы присваивания.

Самый простой оператор присваивания имеет следующий вид:

```
переменная = выражение;
```

Справа находится выражение. Это может быть любое выражение, например, оно может состоять из одного значения, переменной или быть более сложным, с использованием арифметических или логических операторов.

Слева в таком операторе всегда должна быть переменная, в которую заносится результат, то есть всегда сначала вычисляется правая часть оператора, а затем результат заносится в переменную, находящуюся в левой части. Примеры операторов присваивания:

```
x = y * 2 / z;
```

```
x = 3;
```

```
x = y;
```

Также допустимо использовать переменную, стоящую слева в правой части выражения. Например, чтобы увеличить значение *x* в два раза, следует записать:

```
x = x * 2;
```

В этом случае в правом выражении используется старое значение переменной, а потом в нее же заносится результат.

Существует сокращенные формы записи подобных операторов. Они выглядят как:

+= Сложение с присваиванием

-= Удаление с присваиванием

***=** Умножение с присваиванием

/= Деление с присваиванием

%= Деление с остатком с присваиванием

Например, две следующие строчки идентичны по выполняемым действиям:

```
x += 2;
```

```
x = x + 2;
```

Приведение типов

Следует помнить, что, если в операторе присваивания результат справа относится к одному типу данных, а слева – к другому, компилятор может не позволить выполнить такое присваивание и выдать ошибку. Например, если слева булевская переменная, а справа целое число. И это касается не только принципиально разных типов. В переменную *int* нельзя просто присвоить значение *long*.

В случае если переменные сходных типов, например, хранят числа, это можно обойти с помощью так называемого приведения типов. Например:

```
int x;
```

```
double z =10.5;
```

```
x =(int) z;
```

Для приведения типов перед присваиваемым выражением надо поставить нужный нам тип в круглых скобках (часто остальное выражение тоже приходится брать в скобки из-за приоритетов). В этом случае присваивание выполнится и в *x* попадет значение 10. При этом часть данных может теряться, как в нашем примере, при преобразовании из дробного числа в целое всегда отбрасывается дробная часть.

Унарные операторы

Обычные арифметические операторы являются бинарными, так как имеют две части – правую и левую. В Си подобных языках и в Java, в частности, имеются два унарных оператора, то есть требующих только одного аргумента. Это операторы

увеличения и уменьшения на единицу, также называемые операторами инкремента и декремента.

Записываются они как два знака плюс для инкремента и два знака минус для декремента.

```
i++;
```

В данном примере переменная *i* увеличивается на единицу. Следует отметить, что в отличие от обычных арифметических операторов результат их работы необязательно сохранять с помощью оператора присваивания, так как они сами изменяют значение переменной, рядом с которой стоят.

Знаки ++ и -- можно ставить как после переменной, так и до. В указанном примере, когда оператор инкремента стоит отдельно от других выражений, то результат будет одинаковым в обоих случаях.

Разница существует в том случае, если оператор используется внутри выражения. Допустим имеется следующий код:

```
int i = 10;
int j = 10;
int z = 10 + i++;
int y = 10 + ++j;
```

В двух последних выражениях выполняются аналогичные операции, значения переменных *i* и *j* равны 10. Но в результате в *z* попадет 20, а в *y* – 21. Это произойдет потому, что в первом случае увеличение значения переменной происходит после того, как ее значение использовали в выражении. А во втором случае, когда знаки стоят до переменной, сначала увеличивается значение, а потом используется в выражении.

Использование операторов инкремента и декремента внутри выражений, как правило, не рекомендуется, так как затрудняет анализ и понимание кода.

Особенности операторов при работе со строками

Оператор + может применяться не только к числам, но и к строкам.

При сложении двух строк происходит их склеивание. Например:

```
String str = "Hello ";
String newStr = str + "world!";
```

В результате в переменной *newStr* будет находиться значение "Hello world!". Следует помнить, что при сложении двух слов, оператор не вставляет между ними пробел, поэтому пробел должен быть предусмотрен в одной из строк либо прибавлен отдельной строкой:

```
String str = "Hello";
String newStr = str + " " + "world!";
```

При сложении строк один из аргументов может быть и не строкой. В этом случае он автоматически преобразуется в строку:

```
int x = 15;
String str = "Значение x = " + x;
```

В результате в переменной *str* будет строка "Значение x = 15"

В связи с этим следует помнить, что если один из аргументов сложения будет строкой, сложение выполняется по строковому типу:

```
int x = 15;  
String y1 = x + "20";
```

В переменной y1 будет строковое значение "1520"

Пример использования арифметических операторов

Имеется промежуток времени в секундах. Следует вывести его на страницу в виде часов минут и секунд.

```
public class HelloWorld {  
    public static void main(String[] args) {  
        int s = 3700;  
        int sec = s % 60;  
        int m = (s - sec) / 60;  
        int min = m % 60;  
        int h = (m - min) / 60;  
        System.out.println(h + " часов " + min + " минут " + sec + " секунд");  
    }  
}
```

В данном примере все необходимые переменные объявляются по мере их использования, но такой подход необязателен. Можно объявить переменные в начале, а дальше их использовать.

Также следует обратить внимание, что переменные можно объявлять каждую в своей строке, а также и несколько вместе через запятую.

```
public class HelloWorld {  
    public static void main(String[] args) {  
        int s = 3700;  
        int sec;  
        int m;  
        int min, h;  
        sec = s % 60;  
        m = (s - sec) / 60;  
        min = m % 60;  
        h = (m - min) / 60;  
        System.out.println(h + " часов " + min + " минут " + sec + " секунд");  
    }  
}
```

Методы

Понятие о методах

Для выполнения некоторых часто выполняемых действий в языках программирования используются методы. Методы могут создаваться самим программистом, а могут быть стандартными, созданными авторами языка.

Каждый метод имеет свое собственное имя. Имена методов задаются по тем же правилам, что и имена переменных. Чтобы метод выполнил свои действия, его надо вызвать. Для этого следует написать его имя и круглые скобки. Круглые скобки, которые идут после имени, означают, что мы имеем дело именно с методом, а не с переменной.

Чаще всего метод выполняет действия не просто так, а используя некоторые исходные данные. Например, функция синус вычисляет значение синуса угла, используя в качестве исходных данных значение угла. Исходные данные, их также называют входными параметрами, пишутся внутри круглых скобок. Например:

```
x = Math.sin(5);
```

В данном случае метод вычисляет синус угла равного пяти.

Java позволяет не только использовать уже существующие методы, но и создавать пользователю собственные.

Общий вид объявления метода:

```
модификатор_доступа другие_модификаторы тип_результата  
имя_метода(список_параметров) {  
  
    оператор;  
  
}
```

Модификаторы доступа подробно рассматриваются в главе Классы и объекты.

Существуют другие модификаторы, один из которых `static` – необходим для того, чтобы вызвать метод из метода `main`.

Затем идет тип результата, так если в результате работы метода должно получиться число, здесь следует указать тип числа, например, `float` или `long`.

Если метод просто проделывает какие-то действия и не возвращает никакого значения, в качестве типа результата следует указать пустой тип `void`.

Имя метода задается программистом по собственному усмотрению, но следует учитывать ограничения, аналогичные правилам, используемым при создании переменных. То есть желательно использовать только латинские буквы, цифры и знак подчеркивания. Имя не должно начинаться с цифры, а также нежелательно использовать вначале знак подчеркивания.

После имени метода ставятся круглые скобки со списком входных переменных (параметров). Таким образом, задается, какие данные метод будет получать при своем вызове. Входных переменных может быть любое количество, если входных параметров не надо, следует поставить просто пустые скобки. Может быть один параметр, может быть несколько, перечисленных через запятую. Например, для метода, который вычисляет максимальное значение из двух заданных, и соответственно должен иметь два исходных значения заголовок может выглядеть так:

```
double max(double a, double b)
```

Фигурные скобки, которые идут после заголовка, и содержащиеся в них действия (операторы), называются телом метода.

Если метод возвращает значение, то его работа должна завершаться оператором `return`, после которого пишется возвращаемое значение. Например, у нас часто используется операция возведения в третью степень. Мы можем сделать для нее метод вида:

```
double cube(double x){
    double y; y = x * x * x;
    return y;
}
```

Метод получает исходное значение, которое попадает в переменную `x`. Затем создается переменная `y` и в нее заносится результат перемножения `x`. В конце значение `y` возвращается с помощью `return`.

И затем мы можем его использовать в выражениях, например:

```
z = cube(15) / (cube(3) + cube(x + 2));
```

Каждый раз, когда метод встречается в выражении, происходит его вызов, он получает значение, указанное в скобках. Если это выражение, то сначала вычисляется результат выражения, а затем передается в метод. Когда метод выполнил свою работу, его результат подставляется в выражение, откуда его вызвали.

Метод main

Вернемся к простейшей программе на Java.

```
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Для того, чтобы программа успешно запустилась, один из классов в ней должен иметь метод `main`. Заголовок метода должен быть именно таким, какой показан в данном примере. Этот метод запускается, когда вы запускаете вашу программу. Поэтому его часто называют "точка входа".

Так как метод `main` является точкой входа, при запуске программы следует всегда указывать именно тот класс, в котором он объявлен в нашем случае.

```
java HelloWorld
```

Как правило, в программе присутствует один класс с методом `main`. В простых программах иногда достаточно только одного класса и одного этого метода. В таком случае все необходимые операции должны быть выполнены внутри `main`.

Задания

Задание 4

Написать код, который выведет значения переменных на экран:

```
byte b = 0x55;
short s = 0x55ff;
int i = 1000000;
long l = 0xffffffffL;
char c = 'a' ;
float f = .25f;
double d = .00001234;
boolean bool = true;
Для заметок:
```

Задание 5

Написать метод, который принимает на вход два целых числа, делает их суммирование и складывает результат с произведением двух этих чисел, и возвращает полученный результат из метода. Передать на вход в метод любые два числа. Вывести полученный результат работы метода на экран.

Для заметок:

Задание 6

Задать промежуток времени в секундах в виде переменной. Следует вывести его на экран в виде часов минут и секунд, суток и недель.

Для заметок:

Операторы if/switch

Оператор if

Общий вид оператора if

Для того, чтобы выполнять в разных случаях разные действия в языке Java, используется специальный оператор if. Он выполняет разные операторы в зависимости от указанного ему логического значения.

Общий вид оператора if:

```
if (Условие)
    Оператор1;
else
    Оператор2;
```

Если *Условие* истинно, то будет выполняться *Оператор1*, если *Условие* ложно, будет выполняться *Оператор2*.

В качестве условия может использоваться любое выражение, результатом которого является логическое значение. Это может быть логическое выражение, например:

```
if (x == 3)
```

Также это может быть метод, возвращающий логическое значение:

```
if(str.equals("test"))
    System.out.println(b);
```

Условие может также состоять из одной переменной, если это переменная типа `boolean`, например:

```
boolean b = x == 3;
if(b)
    System.out.println(b);
```

Конструкция `else Оператор2;` может отсутствовать и тогда оператор `if` примет вид:

```
if (Условие)
    Оператор1;
```

Пример использования if

Например, нам следует проверить, является ли значение в переменной `x` четным и сообщить об этом пользователю.

```
if (x % 2 == 0)
    System.out.println("Число "+ x + " является четным");
else
    System.out.println("Число "+ x + " является нечетным");
```

В заголовке `if` стоит условие проверки четности. Как известно, четное число делится на 2 без остатка, поэтому мы вычисляем остаток с помощью `x % 2` и затем сравниваем его с нулем. Если остаток равен нулю результат будет истина.

И если результатом логического выражения является истина, будет выполнен оператор:

```
System.out.println("Число "+ x + " является четным");
```

Если же результатом является ложь, будет выполнен оператор:

```
System.out.println("Число "+ x + " является четным");
```

По условию в операторе `if` может выполняться только один оператор, то есть вы не имеете права поставить два оператора между `if` и `else`.

Это значит, что следующая запись приведет к ошибке:

```
if(x % 2 == 0)
    System.out.println("Число "+ x );
    System.out.println(" является четным");
```

Два оператора после `else` к синтаксической ошибке не приведут, но второй из них будет выполняться в любом случае, вне зависимости от условия.

Составной оператор

Чтобы обойти упомянутое ранее ограничение в один оператор, следует использовать так называемый составной оператор. Составной оператор представляет собой несколько операторов, заключенных в фигурные скобки `{ }`, например:

```
{
    f = a + b;
    c = f * a;
    m = f * c;
}
```

Составной оператор может включать в себя любые операторы и вызовы методов, в том числе и другие составные операторы:

```
{
    f = a + b;
    {
        x = f * a;
        m = f * c;
    }
    c = f * x * m;
}
```

В результате пример `if`, вызывавший ошибку, можно переписать следующим образом:

```
if(x % 2 == 0){
    System.out.println("Число "+ x );
    System.out.println(" является четным");
} else
```

Такая запись не будет вызывать ошибок и по условию выполнятся оба оператора.

Часто рекомендуют ставить фигурные скобки в любом операторе `if` вне зависимости от количества операторов:

```
if (x % 2 == 0) {
    System.out.println("Число "+ x + " является четным");
}
```

Вложенные операторы if

В качестве оператора, выполняемого по условию, в операторе if может использоваться другой оператор if. В таком случае говорят о вложенных условных операторах. Такая конструкция имеет вид:

```
if (Условие1)
    if(Условие2)
        Оператор1;
    else
        Оператор2;
else
    Оператор3;
```

Избежать возможных затруднений при анализе данной конструкции позволяет простое правило: else относится к ближайшему свободному if.

Возвращаясь к примеру с вложенными операторами if, следует иметь в виду, что рекомендуется вложенный if брать в фигурные скобки, так как это значительно облегчает чтение программы и уменьшает вероятность путаницы с определением соответствующих else. В этом случае запись вложенного if принимает вид:

```
if (Условие1) {
    if (Условие2)
        Оператор1;
    else
        Оператор2;
} else
    Оператор3;
```

Пример вложенных операторов if

Например, имеется задача найти и вывести на страницу максимальное значение из трех имеющихся: a, b и c. Одно из возможных решений задачи имеет вид:

```
if(a > b) {
    if(a > c) {
        System.out.println("максимальное число:" + a);
    } else {
        System.out.println("максимальное число:" + c);
    }
} else {
    if(b > c) {
        System.out.println("максимальное число:" + b);
    } else {
        System.out.println("максимальное число:" + c);
    }
}
```


Сначала сравниваются первые два числа, а затем большее из них сравнивается с третьим, если оно больше, значит, оно самое большое из трех, если оно меньше, значит, самым большим является третье.

Конструкция if else if

Очень часто встречаются задачи, в которых выбор следует сделать между более чем двумя возможными вариантами, каждому из которых соответствует свое условие. Тогда применяется конструкция if else if. Она имеет вид:

```
if (условие1) {  
    оператор1;  
} else if (условие2) {  
    оператор2;  
} else if (условие3) {  
    оператор3;  
}
```

Конструкция работает следующим образом: если верно Условие1, то выполняется оператор1, если же оно ложно, проверяется условие2, и если оно истинно, то выполняется оператор2, если и оно ложно, то проверяется следующее условие и т.д.

Пример использования конструкции if else if

Типичным примером использования такой конструкции является вывод названия дня недели, если известен его номер:

```
if(n == 1) {  
    System.out.println("Понедельник");  
} else if(n == 2) {  
    System.out.println("Вторник");  
} else if(n == 3) {  
    System.out.println("Среда");  
} else if(n == 4) {  
    System.out.println("Четверг"); }  
else if(n == 5) {  
    System.out.println("Пятница"); }  
else if(n == 6) {  
    System.out.println("Суббота");  
} else if(n == 7) {  
    System.out.println("Воскресенье");  
} else {  
    System.out.println("Дня с таким номером не существует");  
}
```

Здесь номер дня должен находиться в переменной `n`. Если номер находится в пределах от 1 до 7, то выводится название дня, иначе выводится сообщение об ошибке (последний `else`).

Оператор выбора `switch`

Если необходимо делать выбор из конкретных значений, можно использовать не конструкцию `if else if`, а специальный оператор `switch`. Его общий вид:

```
switch (выражение) {  
    case значение1:  
        операторы  
    break;  
    case значение2:  
        операторы2  
    break;  
    default:  
        операторы3  
}
```

В скобках `switch` должно стоять выражение, результатом которого должно быть значение примитивного типа или типов `String` и `Enum`, полученное значение далее будет сравниваться. После `case` ставится значение, и если результат выражения совпал с этим значением, выполняются операторы после двоеточия и до `break`. Если не одно из предложенных значений не совпало с результатом выражения, выполняются операторы после `default`.

Решение задачи про выбор дня недели с помощью `switch` будет выглядеть так:

```
switch (n) {  
    case 1:  
        System.out.println("Понедельник");  
        break;  
    case 2:  
        System.out.println("Вторник");  
        break;  
    case 3:  
        System.out.println("Среда");  
        break;  
    case 4:  
        System.out.println("Четверг");  
        break;  
    case 5:  
        System.out.println("Пятница");  
        break;  
    case 6:  
        System.out.println("Суббота");  
        break;  
    default:  
        System.out.println("Неверный номер дня недели");  
}
```

```

System.out.println("Суббота");
break;
case 7: System.out.println("Воскресенье");
break;
default:
System.out.println("Дня с таким номером не существует");
}

```

Надо учитывать, что если какой-либо из разделов switch не заканчивается оператором break, то начнут выполняться операторы из следующего раздела:

```

switch (n) {
    case 1: System.out.println("Понедельник");
    case 2: System.out.println("Вторник");
    break;
}

```

В данном фрагменте, если n имеет значение 1, то будут выведены и Понедельник, и Вторник.

Еще следует обратить внимание, что в отличие от if здесь можно писать несколько операторов после case и для этого не требуются фигурные скобки.

Задания

Задание 7

Создайте метод с одним целочисленным параметром. Метод должен определить, является ли последняя цифра числа семеркой и вернуть boolean значение.

Для заметок:

Задание 8

Имеется прямоугольное отверстие размерами a и b, где a и b – целые числа. Определить, можно ли его полностью закрыть круглой картонкой радиусом r (тоже целое число).

Для заметок:

Задание 9

Задать целое число в виде переменной, это число – сумма денег в рублях. Вывести это число на экран, добавив к нему слово «рублей» в правильном падеже.

Для заметок:

Задание 10

Задать три числа – день, месяц, год. Вывести на экран в виде трех чисел дату следующего дня.

Для заметок:

Задание 11

Имеются два дома размерами a на b и c на d . Имеется участок размерами e на f . Проверить, помещаются ли эти дома на данном участке. Стороны домов – параллельны сторонам участка, в остальном размещение может быть любым.

Для заметок:

Задание 12

Написать метод, который выводит расписание на неделю. Задать на вход в метод порядковый номер дня недели и отобразить на экране то, что запланировано на этот день.

Для заметок:

Циклы

Оператор циклов while

Часто для достижения требуемого результата приходится повторять какие-либо действия многократно до тех пор, пока результат не будет достигнут, то есть до выполнения какого-либо условия.

Для таких целей в языках программирования используются операторы циклов.

Простейшим оператором цикла является оператор while.

Общий вид оператора while:

```
while (условие)
    оператор
```

Цикл while работает следующим образом: проверяется условие, и если оно верно, выполняется оператор. Затем снова проверяется условие и выполняется оператор, и так будет продолжаться до тех пор, пока условие верно. Если условие стало не верным, выполняется следующий оператор после цикла. Оператор, выполняемый в цикле, называется телом цикла. Как и в операторах условия, телом цикла должен быть один оператор. Если необходимо выполнять в цикле несколько операторов, следует использовать составной оператор с фигурными скобками.

Например, следует вычислить сумму чисел от 1 до 200. Программа, реализующая данную задачу, выглядит следующим образом:

```
int sum = 0;
int i = 1;
while (i <= 200) {
    sum += i;
    i++;
}
```

Сначала создаются две переменные: sum – в которой хранится сумма чисел, i – переменная счетчик, позволяющая перебирать числа от 1 до 200. Изначально сумма = 0, на каждом шаге i добавляется к сумме, и после этого увеличивается на 1. Когда i станет больше 200, цикл прекратится, и выполнение программы продолжится.

В качестве условия может использоваться любое логическое выражение, включая условные и логические операторы.

Пример использования цикла while

Имеется следующая задача: надо получить ряд случайных чисел и посчитать среднее арифметическое от них. Ввод чисел прекращается, если очередным числом будет ноль.

```
double sr = 0;
double sum = 0;
int n = 0;
int x = (int) (Math.random() * 20);
while (x != 0) {
    sum += x;
```

```

n++;
x = (int) (Math.random() * 20);
}
if(n != 0){
    sr = sum / n;
} else {
    sr = 0;
}
System.out.println("среднее:" + sr);

```

Переменная `sr` служит для хранения среднего значения, переменная `sum` служит для хранения суммы чисел, переменная `n` служит для хранения количества введенных чисел. Сначала получается первое число в переменную `x`. Если оно не равно нулю, оно прибавляется к сумме, и увеличивается счетчик `n` на единицу. Затем получается новое число и цикл повторяется. Если оно равно нулю, цикл прекращается. В конце стоит проверка на случай, если первое же число будет равным нулю, чтобы избежать деления на ноль.

Оператор `do while`

Существует еще один вариант оператора `while`, `do while`. Общий вид этого оператора выглядит так:

```

do
    оператор
while (условие);

```

Условие здесь проверяется не перед началом прохода цикла, а после его завершения, отсюда следует особенность, что в отличие от `while`, даже если условие не выполняется изначально, хотя бы один раз тело цикла будет выполнено.

Еще важной синтаксической особенностью является точка с запятой после круглых скобок. В обычных операторах циклов точка запятой после круглых скобок ставится только в особых случаях. Здесь она обязательна.

Пример записи цикла `do while`:

```

int s = 50;
int i=0;
do {
    s = s - 4;
    i++;
} while(i < 4 && i > 0);
System.out.println("s=" + s + " i=" + i);

```

В данном случае мы имеем дело с ситуацией, когда условие в `while` на момент запуска цикла не выполняется, так как `i` равно 0. Но поскольку цикл проверяет условие в конце тела, он запускается, в теле значение `i` меняется и цикл продолжает свою работу.

Оператор цикла for

Часто циклы выполняются определенное, заданное заранее, число раз. Для отсчета количества повторений используют специальную переменную, называемую переменной-счетчиком или параметром.

Для написания циклов с параметром (параметр – это переменная-счетчик, в дальнейшем просто “счетчик”) был создан оператор цикла for.

Общий вид оператора for:

```
for (инициализация; условие; инкремент)
    операторы;
```

Здесь инициализация – это блок присвоения начального значения параметру цикла (счетчику); условие – блок, отвечающий за продолжение работы цикла, пока его значение истинно (то есть отлично от нуля) цикл продолжается; инкремент – блок, изменяющий параметр цикла (счетчик).

Любой из этих блоков необязателен для работы оператора, но формально должен присутствовать и разделяться символом ; от других блоков.

В качестве условия в операторе for может использоваться любое логическое выражение, так же как и в операторе while. Кроме того, следует помнить, что условие необязательно связано с параметром цикла.

В качестве примера использования оператора for переделаем задачу, рассмотренную в разделе для оператора while:

```
int sum = 0;
for(int i = 1; i <= 200; i++){
    sum += i;
}
```

Как видно из данного примера, for позволяет записать цикл с параметром более компактно, чем while.

Пример программы, использующей оператор цикла for

Имеется задача: вывести на экран первые десять целых чисел, которые делятся на 3 без остатка.

```
int n = 10;
int k = 1;
for(int i = 1; k <= n; i++) {
    if(i % 3 == 0){
        System.out.println(k + ":" + i); k++;
    }
}
```

В переменной n содержится количество чисел, которые следует вывести. Переменная k служит для подсчета количества выведенных чисел. Переменная i служит для перебора всех целых чисел по порядку. В теле цикла находится условный оператор, проверяющий остаток от деления очередного i на 3. Если остаток равен нулю, число выводится, и переменная k увеличивается на единицу. Когда переменная k превышает 10, цикл завершается.

Следует обратить внимание на один важный момент: условие цикла непосредственно не связано с первым и третьим оператором заголовка.

Вложенные циклы

В теле цикла может находиться практически любой оператор, в том числе и другой оператор цикла. Например, необходимо вывести таблицу умножения. Исходный текст программы будет выглядеть следующим образом:

```
for(int i = 1; i < 10; i++) {  
    for(int j = 1; j < 10; j++) {  
        System.out.print(i * j + " ");  
    }  
    System.out.println("");  
}
```

Сначала начинается цикл, перебирающий строки таблицы. Тело данного цикла содержит вывод строки. В нем запускается второй цикл, создающий элементы таблицы. Каждый раз выводится произведение номера строки и столбца плюс пробел, для отделения чисел друг от друга. Когда этот цикл завершается, выводится перевод строки с помощью `println`. После этого происходит увеличение номера строки и процесс повторяется. Второй цикл называется вложенным, и полностью выполняется столько раз, сколько проходов содержит первый цикл.

Операторы break и continue

Иногда выполняющийся цикл следует остановить, если выполняется некоторое условие. Для этого существует специальный оператор `break`. Оператор `break` прекращает выполнение цикла и передает управление операторам, которые находятся после тела цикла.

Например, существует задача вывести значение выражения $y = 1/x$, при значениях x от -10 до 10, с шагом 1. Обычный цикл будет выглядеть следующим образом:

```
int x;  
int y;  
for(x = -10; x <= 10; x += 1) {  
    y = 1 / x;  
    System.out.println("x = " + x + " y= " + y + "\n" );  
}
```

С данным фрагментом программы имеется только одна проблема, в один из моментов x может быть равен 0. В результате будет выполнено деление на 0 и выведены некорректные данные на страницу. Во избежание этой проблемы возможны два действия: или прекратить выполнение цикла, если x равен 0; или пропустить шаг, на котором x равен 0. Для выполнения первого варианта можно использовать оператора `break`.

```
int x;  
int y;
```



```

for(x = -10; x <= 10; x +=1) {
    if(x == 0){
        break;
    }
    y = 1 / x;
    System.out.println("x=" + x + " y=" + y + "\n" );
}

```

В таком случае цикл будет выполняться для значений x от -10 до -1.

Если необходимо реализовать второй вариант, то есть. пропустить нежелательный проход цикла, можно воспользоваться оператором continue. Данный оператор заставляет пропустить все операторы до конца тела цикла, и начать новый проход.

```

int x;
int y;
for(x = -10; x <= 10; x +=1) {
    if(x == 0){
        continue;
    }
    y = 1 / x;
    System.out.println("x = " + x + " y = " + y + "\n" );
}

```

В данном случае будут выданы все значения функции, кроме одного нежелательного.

Задания

Задание 13

Вычислить факториал целых чисел от 0 до 10 с помощью цикла while.
Для заметок:

Задание 14

Вычислить произведение чисел от 1 до 25 с помощью цикла do while.
Для заметок:

Задание 15

Посчитать сумму цифр числа 7893823445 с помощью цикла do while.
Для заметок:

Задание 16

Найти среди чисел от 50 до 70 второе простое число (число называют простым, если оно делится без остатка только на 1 и себя) и завершить цикл с использованием break.

Для заметок:

Задание 17

Для целых чисел, которые делятся на 7 в диапазоне от 1 до 100, вывести на экран строку "Норе!".

Для заметок:

Задание 18

Задать произвольное целое число и вывести его в бухгалтерском формате, то есть, начиная справа, каждые три позиции отделяются пробелом. Например, число 20023143 должно быть выведено как 20 023 143.

Для заметок:

Массивы

Одномерные массивы

Встречаются ситуации, когда использование обычных переменных для хранения данных будет не слишком удобным. Обычно это происходит, когда данных становится много. Например, вам необходимо хранить итоговые оценки по биологии учеников класса. Если класс обычный, то вы будете должны создать около двух десятков переменных, что само по себе неудобно. А если вам понадобится распечатать оценки в виде «отлично», «хорошо» и т. д., вам придется для каждой переменной делать собственные условия. В результате ваша программа будет очень длинной и в ней будет чрезвычайно сложно разбираться.

Такие проблемы можно решить с помощью массивов. Массивы позволят вам создавать не множество отдельных переменных, в каждой из которых хранится одно значение, а одну переменную, в которой можно хранить множество значений.

Таким образом, массив – тип данных, позволяющий хранить в одной переменной сразу несколько значений. Чтобы отличать эти значения их нумеруют. Номер переменной называют индексом, поэтому иногда массивы называются индексированными переменными. Каждое значение в массиве называется элементом массива.

Для создания переменной массива указывается тип элемента, имя массива и пустые квадратные скобки. Скобки могут стоять как после типа, так и после имени.

```
String args[]
```

Данная строка имеется в объявлении main и означает, что входной переменной метода является массив args типа String.

При создании массива следует помнить, что массивы являются особым типом объектов, поэтому приведенная выше запись создает только ссылку на массив, а не сам массив. Сам массив создается с помощью оператора new.

```
int marks[] = new int[20];
```

Мы создали массив marks для хранения оценок двадцати учеников, то есть массив, состоящий из 20 элементов. Как уже было сказано, каждый элемент массива имеет собственный номер. Нумерация начинается с нуля, и идет по порядку, то есть в данном случае в массиве будут элементы с номерами от 0 до 19.

Если вам заранее известно, какие значения должны находиться в элементах вашего массива, их можно перечислить в фигурных скобках через запятую. Например, ваша программа должна выполнять операции с датами, и для этого вам надо учитывать количество дней в каждом из месяцев. Это опять же удобно сделать в виде массива:

```
int daysInMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

В данном случае будет создан массив с двенадцатью элементами. Номера элементам будут заданы автоматически начиная с 0 и далее по порядку. То есть с номером 0 будет иметь значение 31 и соответствовать январю, элемент с номером 1 будет иметь значение 28 и соответствовать февралю, элемент с номером 2 будет иметь значение 31 и так далее.

Как работать с его значениями? Чтобы обратиться к значению, которое хранится в массиве, следует использовать не только имя массива, но и индекс нужного вам значения в квадратных скобках. Если мы работаем с оценками

учеников, то логично, если индексы элементов массива будут соответствовать номеру учеников в списке класса. Только следует учитывать, что нумерация начинается с 0, и чтобы добраться до значения элемента, следует написать имя массива, после которого должны идти квадратные скобки, внутри которых находится индекс требуемого элемента.

```
marks[2] = 4;
```

В результате такой операции в элемент с индексом 2 (то есть ученику, идущему по списку третьим) заносится оценка 4.

При записи индекса элемента можно писать не только число, но и практически любое выражение, результатом которого будет целое число.

```
int i = 6;
marks[i+2] = 10;
```

В результате этих операций в элемент массива индекс 8 попадет значение 10, то есть эту оценку получил девятый по счету ученик.

С элементами массивов можно выполнять любые операции, которые допустимы с обычными переменными, то есть присваивать им значения (как мы видели из предыдущих примеров), использовать их в арифметических, логических и других выражениях.

Например, нам необходимо средняя оценка первого и последнего учеников в списке:

```
int averageMark = (marks[0] + marks[19])/2;
```

Примеры заполнения и вывода массивов

Очень часто значения массива неизвестны в момент, когда программа пишется, а должны заполняться в процессе ее выполнения.

Пример заполнения массива из двадцати элементов, случайными числами:

```
int marks[] = new int[20];
for(int i=0; i < 20; i++){
    marks[i] = (int)(Math.random()*10 + 1);
}
```

Здесь для формирования самой оценки используется стандартный метод `Math.random()`, возвращающий случайное число от 0 до 1

Если следует ввести массив другого размера, нужно поменять числа в объявлении массива и в заголовке цикла.

Важной является также операция вывода массива. Когда нам необходимо посмотреть оценки учеников класса, вывод может выглядеть так:

```
for (int i=0; i < 20; i++) {
    System.out.println("Ученик №"+ i + " = " + marks[i]);
}
```

В результате на экране будут выведены ученики с их номерами и оценками, в виде:

```
Ученик№0 = 6
```

и т. д.

Часто мы не знаем заранее, какого размера будет наш массив, что может создавать проблемы. Предыдущий пример написан исходя из того, что в классе строго 20 учеников, если их больше, то часть из них не будет распечатана, если меньше – будет выведено сообщение об ошибке. Поэтому в большинстве языков программирования есть методы или переменные, позволяющие узнать, сколько переменных в массиве. В нашем случае, чтобы узнать количество элементов в массиве, можно воспользоваться значением поля `length` массива. Она пишется после имени интересующего нас массива через точку. Например:

```
System.out.println(marks.length);
```

При выполнении данной строки будет выведено количество элементов в массиве (то есть учеников в нашем классе) в виде числа. С помощью этой переменной мы можем распечатать массив, сколько бы элементов в нем не было:

```
for (int i=0; i < marks.length; i++) {  
    System.out.println("Ученик №" + i + " = " + marks[i]);  
}
```

В Java нумерация элементов массива начинается с 0 и идет без пропусков. Следует помнить, что `length` содержит количество элементов в массиве, поэтому номер последнего элемента массива равен `marks.length – 1`. Поэтому в заголовке условия содержит знак «меньше», а не «меньше либо равно».

В цикле величина `i` меняется от 0 до `marks.length – 1` и на каждом шаге значение `i` используется в качестве номера для элемента массива.

Примеры работы с массивами

Допустим, возникла следующая задача: для отчета необходимо указать наивысшую оценку, полученную учениками, то есть перед нами стоит задача поиска максимального элемента:

```
int maxMark = marks[0];  
for(int i=0; i < marks.length; i++) {  
    if(maxMark < marks[i]) {  
        maxMark = marks[i];  
    }  
}  
  
System.out.println("максимальная оценка =" + maxMark);
```

Переменная `maxMark` хранит максимальную оценку, обнаруженную в массиве. В самом начале надо присвоить этой переменной начальное значение, которое должно быть больше либо равно максимальному значению массива. Поэтому наиболее надежным действием будет присвоить `m` значение любого элемента массива. Логично для этого использовать нулевой элемент.

В каждом проходе цикла очередное значение массива, то есть оценка очередного ученика, сравнивается с найденной ранее, и если она оказалась больше, то она становится наибольшей. Так как в цикле перебираются оценки всех учеников, после завершения цикла в переменной `maxMark` будет наибольшее значение, то есть самая высокая оценка в классе.

Следующая задача. Следует провести воспитательную работу с учениками, получившими неудовлетворительные оценки (то есть 3 и ниже), для этого необходимо получить список номеров этих учеников:

```
for (int i = 0; i < marks.length; i++) {  
    if(marks[i] <= 3){  
        System.out.println("Ученик №="+ i + " = " + marks[i]);  
    }  
}
```

То есть здесь мы решаем задачу распечатки массива, только не всех его элементов, а тех, которые соответствуют определенным условиям.

Если возникла необходимость сделать статистику оценок учеников, то проще это делать, если оценки будут отсортированы по убыванию или возрастанию.

Существует несколько способов это сделать. Ниже описан один из них.

Улучшенный цикл foreach с массивами

Допустим, что возникла следующая задача: необходимо вывести на экран массив, но при этом нам не важен порядок вывода и текущая позиция элемента. Для этого используется улучшенный цикл for:

```
for(int element: marks) {  
    System.out.println(element);  
}
```

Сортировка массива методом прямого выбора

Метод заключается в том, что ищется максимальный элемент массива и меняется с первым местами, а затем ищется максимальный элемент среди оставшихся и меняется местами со вторым, и так далее.

```
for (int i = 0; i < marks.length; i++) {  
    maxIndex = i;  
    for(int j = i; j < marks.length; j++) {  
        if(marks[maxIndex] < a[j]){  
            maxIndex = j;  
        }  
    }  
    int temp = marks[maxIndex];  
    marks[maxIndex] = marks[i];  
    marks[i] = temp;  
}
```

Внешний цикл предназначен для перебора всех элементов массива. Внутренний служит для поиска наибольшего элемента. Поиск должен начинаться каждый раз с нового элемента, поэтому во втором цикле начальным значением является не 0, а i. В отличие от предыдущего примера, где искомое значение

наибольшего элемента, здесь ищется номер, поэтому условие в if и присваиваемое значение отличаются.

После того как внутренний цикл заканчивается, выполняется обмен найденного наибольшего элемента с текущим. Для этого применяется метод "трех стаканов".

Многомерные массивы

Язык Java допускает создание многомерных массивов. Для этого следует при объявлении указать несколько пар квадратных скобок.

```
int m[][] = new int[5][5];
```

Данная операция создает квадратный двухмерный массив.

Пример заполнения массива единицами:

```
for (int i = 0; i < m.length; i++) {  
    for (int j = 0; j < m[i].length; j++) {  
        m[i][j] = 1;  
    }  
}
```

Следует обратить внимание на то, что свойство length в одном случае вызывается от самого массива, а во втором – от его строки. В первом случае оно возвращает количество строк в массиве, а во втором – количество элементов в строке.

Еще одной особенностью массивов в Java является возможность создания многомерных массивов с переменным количеством элементов в строках (для двухмерного случая). Создание такого массива будет иметь вид:

```
int m[][] = new int[5][];
```

После этого каждую строку следует создать отдельно, например, если нам нужен массив треугольной формы, можно использовать цикл:

```
for (int i = 0; i < m.length; i++) {  
    m[i] = new int[i+1];  
}
```

Если после этого с помощью указанного выше фрагмента заполнить массив единицами и распечатать его, результат будет следующим:

```
1 1 1  
1 1 1  
1 1 1 1  
1 1 1 1 1
```

Задания

Задание 19

Создайте переменную для массива из 10 элементов. Заполните его произвольными значениями целочисленного типа и выведите последний элемент массива на экран.

Для заметок:

Задание 20

Создайте переменную для массива из 10 элементов (другим способом). Заполните его произвольными значениями целочисленного типа и выведите на экран элементы, стоящие на четных позициях.

Для заметок:

Задание 21

Создайте переменную для массива из 10 элементов. Заполните его произвольными значениями целочисленного типа. Найдите максимальный элемент и выведите его индекс.

Для заметок:

Задание 22

Создайте переменную для массива из 10 элементов. Заполните его произвольными значениями целочисленного типа. Определите сумму элементов массива, расположенных между минимальным и максимальным значениями. Если значений максимальных и минимальных несколько, то необходимо взять максимальное значение разницы индексов между максимальным и минимальным значениями.

Для заметок:

Задание 23

Создайте переменную для массива из 10 элементов. Заполните его произвольными значениями целочисленного типа. Выведите на экран, переверните

и снова выведите на экран (при переворачивании нежелательно создавать еще один массив).

Для заметок:

Задание 24

Создать двухмерный квадратный массив и заполнить его «бабочкой», то есть таким образом:

```
1 1 1 1 1
0 1 1 1 0
0 0 1 0 0
0 1 1 1 0
1 1 1 1 1
```

Для заметок:

Классы и объекты

Переменные и методы в классах

Переменные предназначены для того, чтобы хранить данные описываемого классом объекта. Так, если вам необходимо описать геометрическую точку у вас должны быть переменные, описывающие ее координаты. В классе это будет выглядеть следующим образом:

```
public class Point {
    int x;
    int y;
}
```

Таким образом, мы говорим, что в объектах нашего класса точка будет описываться двумя значениями: x и y.

Переменные, созданные в самом классе, называются полями или свойствами класса.

Если мы хотим, чтобы эти значения были доступны всем, перед их объявлением следует поставить слово `public`. Данное слово называют модификатором доступа.

```
public class Point {
```

```
public int x;
public int y;
}
```

Также можно сделать так, чтобы данные переменные были видны только внутри этого класса. В этом случае вместо `public` перед типом переменной ставится слово `private`.

Модификаторы доступа

Существует четыре основных модификатора доступа:

- `private`
- `public`
- `protected`
- по умолчанию или `package-private`

При создании классов рекомендуется, по возможности, закрывать доступ к переменным, то есть делать их `private`, а работу организовывать через методы.

Если переменная имеет тип доступа `private`, чтобы другие классы имели доступ к ее значению, делаются методы для назначения нового значения и его получения.

```
public class Point {
    private int x;
    public void setX(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
}
```

Методы в классах

Метод `setX` получает извне новое значение для `x` и заносит его в свойство класса. Метод `getX` возвращает значение `x`.

Для возвращения значения используется оператор `return`. После него ставится возвращаемое значение и точка с запятой.

Оператор `return` всегда прекращает работу метода. Если метод имеет тип результата `void`, после `return` не должно стоять возвращаемого значения, а сразу должна идти точка с запятой. Также если и результат `void`, оператор `return` вообще может отсутствовать. В этом случае метод завершается, когда выполнены все операции, которые находятся в его теле.

Создание классов и объектов

Ранее мы уже рассматривали общую структуру класса. Рассмотрим пример класса, описывающий лампочку. Наша лампа должна уметь включаться, выключаться, а также у нас должна быть возможность узнать, в каком она сейчас состоянии:

```

public class Lamp {
    private boolean isOn;
    public void on() {
        this.isOn = true;
    }
    public void off() {
        this.isOn = false;
    }
    public boolean getState() {
        return isOn;
    }
}

```

Таким образом, мы имеем свойство, хранящее состояние лампочки, и три метода позволяющие с ним работать. Методы on и off включают и выключают лампочку, а getState позволяет узнать, в каком она состоянии в данный момент.

Для того, чтобы начать использование этого класса, необходимо создать его объект. Например, это можно сделать внутри метода main. Сначала следует создать переменную объектного типа:

```
Lamp first;
```

Созданная переменная пока не содержит объекта. Переменные классов не содержат самих объектов, они служат для хранения адресов объектов в памяти. Изначально в переменной будет пустое значение, то есть null. И если вы попытаетесь у этой переменной вызвать методы класса, вы получите ошибку.

Чтобы создать сам объект, следует воспользоваться оператором new в таком виде:

```
first = new Lamp();
```

После этого объект будет создан и лампочку можно включить, например:

```
first.on();
```

Можно совместить объявление переменной и создание объекта:

```
Lamp first = new Lamp();
```

На один и тот же объект может ссылаться несколько переменных. Так вы можете записать:

```
Lamp first2 = first;
```

В результате у нас будет две переменных: first и first2, но они будут ссылаться на один и тот же объект.

Созданная нами лампочка будет существовать, пока есть хоть одна переменная, в которой хранится адрес этого объекта. После того, как все ссылки на объект исчезли, он будет уничтожен специальной службой, которая называется сборщиком мусора.

Конструкторы. Ссылка this

Для выполнения операций, необходимых для создания класса, в Java используются конструкторы. Конструктор – метод класса, вызываемый при создании объекта. Главные особенности объявления конструктора: он должен иметь имя, совпадающее с именем класса, и не должен иметь возвращаемого значения (даже void).

```
Модификатор_Доступа ИмяКласса() {  
    ...  
}
```

Например, мы хотим, чтобы каждая лампочка из предыдущего примера при создании была включена. Для этого в класс необходимо добавить метод вида:

```
public Lamp() {  
    isOn = true;  
}
```

В результате, когда мы будем использовать оператор new для создания нового объекта, будет вызываться конструктор и включать лампочку.

Конструкторы могут иметь входные параметры. В этом случае при создании объекта можно указать в скобках значения для этих параметров. Конструкторы также могут быть перегружены, то есть может быть несколько конструкторов с разными наборами параметров. Это же относится и к другим методам. Можно создать несколько вариантов методов с одинаковыми именами и разными исходными параметрами. Данный механизм называется перегрузкой метода (overloading). По умолчанию любой класс имеет пустой конструктор, даже если он не объявлен явно, но если мы его перегружаем, то чтобы его использовать, нам необходимо добавить его явно.

Например, нам необходимо создать класс, описывающий геометрическую точку с двумя координатами:

```
public class Point {  
    private int x;  
    private int y;  
    public Point() {  
        x = 1;  
        y = 1;  
    }  
    public Point(int x1, int x2) {  
        x = x1;  
        y = x2;  
    }  
}
```

Такую точку можно создать двумя способами:

```
Point p1 = new Point();  
Point p2 = new Point(15, 6);
```

В первой строке вызывается первый конструктор и создается точка с координатами 1 и 1, во втором – второй конструктор с координатами 15 и 6.

Следует учитывать, что, если объявлены только конструкторы с параметрами, создать объект, не указав параметры, не получится. То есть если в предыдущем примере пропустить описание первого конструктора, строка вида

```
Point p = new Point();
```

вызовет ошибку, так как используется конструктор без параметров.

Если в конструкторе или другом методе возникает конфликт имен, может использоваться ссылка `this`. Она представляет ссылку на объект, вызвавший данный метод. Например, второй конструктор класса `Point` может быть переписан в виде:

```
Point(int x,int y) {  
    this.x = x;  
    this.y = y;  
}
```

Также `this` может использоваться для вызова из конструктора других конструкторов. Например, можно создать еще один конструктор:

```
Point(int x) {  
    this(x,x);  
}
```

Так как в языке Java присвоение начальных значений переменным может быть выполнено несколькими способами, следует учитывать порядок назначения начальных значений. Первыми выполняется установка начальных значений при создании полей, а последним выполняется конструктор и заданные в нем действия. То есть в классе `Point` сначала свойства `x` и `y` получают нулевые значения и только затем в конструкторе, их значения меняются.

Так как освобождением памяти занимается виртуальная машина. В Java нет такой острой необходимости в деструкторах, как в Си++. Тем не менее часто в конце работы объекта необходимо выполнить завершающие действия, такие как закрытие файлов и т.п. Для этого существует специальный метод, вызываемый автоматически при удалении объекта. Этот метод должен иметь имя `finalize`.

```
finalize() {  
    операторы  
}
```

Как и деструктор в Си++ он не имеет возвращаемого значения и входных параметров. При его использовании следует учитывать, что виртуальная машина не гарантирует немедленного удаления объекта, поэтому данный метод может быть вызван в любое время после того, как объект потерял все ссылки на него.

Статические и константные члены класса

Если необходимо, чтобы все объекты этого класса пользовались одним свойством или одним методом, его следует объявить с помощью ключевого слова `static`. Например, необходимо создать переменную, хранящую общее количество объектов:

```
public static int count = 0;
```

Если вы объявили в классе такую переменную, а затем в конструктор добавили строку вида:

```
count++;
```

В результате в этой переменной будет количество созданных в вашей программе объектов данного класса.

Статические свойства создаются при запуске программы. К ним можно обращаться не только от имени переменных, но и от имени их класса. Так если мы создали переменную count в классе Point, мы можем обратиться к ней, записав:

```
Point.count.
```

Статическими могут быть не только свойства, но и методы. Статические методы, также как и свойства могут вызываться без создания объекта данного класса.

В приведенном в начале примере метод main – статический, что позволяет автоматически вызывать его, не создавая объекта класса.

Следует также помнить, что статические методы не могут напрямую получить доступ к не статическим методам и свойствам этого класса. Так, если описать класс следующего вида:

```
public class HelloWorld {  
    private int a = 5;  
    private void print() {  
        System.out.println(a);  
    }  
    public static void main(String[] args) {  
        print();  
    }  
}
```

При компиляции данного примера будет выдана ошибка компиляции, о невозможности вызова print из статического метода, и программа не будет скомпилирована.

Обойти это можно, создав объект класса, и вызывая метод через объект:

```
HelloWorld h1 = new HelloWorld();  
h1.print();
```

Статический метод может быть вызван не от имени объекта, а от имени класса. Если класс A имеет статический метод b, его можно вызвать как:

```
A.b();
```

Иногда возникает необходимость в выполнении достаточно сложных действий для присвоения начальных значений статическим свойствам класса. Для этого внутри класса используются специальные блоки, напоминающие методы, но вместо модификаторов и имени метода стоит слово static, а также отсутствуют скобки и входные параметры:

```
static {  
    операторы  
}
```

Если необходимо, чтобы свойство класса не могло изменить своего значения, его следует объявить с помощью ключевого слова `final`.

```
final double PI = 3.14159;
```

При попытке присвоить переменной другое значение, компилятор выдаст ошибку.

Стандартные классы и объекты Java

Ввод с консоли

Для получения ввода с консоли используют класс `Scanner`, который использует `System.in` внутри себя.

Класс `Scanner` находится в пакете `java.util`, необходимо заимпортировать:

```
import java.util.Scanner.
```

Для создания самого объекта `Scanner` в его конструктор передается объект `System.in`. После этого мы можем получать вводимые значения.

Класс `Scanner` имеет следующие методы:

- `next()` – читает введенную строку до первого пробела;
- `nextLine()` – читает всю введенную строку;
- `nextInt()` – читает введенное число `int`;
- `nextDouble()` – читает введенное число `double`;
- `nextBoolean()` – читает значение `boolean`;
- `nextByte()` – читает введенное число `byte`;
- `nextFloat()` – читает введенное число `float`;
- `nextShort()` – читает введенное число `short`.

Пример:

```
public class Program {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        System.out.print("Name: ");  
        String name = in.nextLine();  
        System.out.print("Age: ");  
        int age = in.nextInt();  
        System.out.print("Height: ");  
        float height = in.nextFloat();  
        System.out.println("Name: " + name);  
        System.out.println("Age: " + age);  
        System.out.println("Height: " + height);  
        in.close();  
    }  
}
```


Математика

Для работы с математическими функциями используется класс `Math`.

`Math` содержит несколько математических констант. Наиболее часто используемыми являются `PI` и `E`. Таким образом, чтобы получить значение `Пи`, следует записать `Math.PI`.

Также имеются следующие методы:

- `Math.abs(x)` – модуль аргумента;
- `Math.ceil(x)` – округление в большую сторону;
- `Math.cos(x)` – косинус;
- `Math.exp(x)` – экспонента;
- `Math.floor(x)` – округление в меньшую сторону;
- `Math.log(x)` – логарифм;
- `Math.max(x,y)` – максимальное значение из данных аргументов;
- `Math.min(x,y)` – минимальное значение;
- `Math.pow(x,y)` – возведение в степень;
- `Math.random()` – случайное число от 0 до 1;
- `Math.round(x)` – округление к ближайшему целому;
- `Math.sin(x)` – синус;
- `Math.sqrt(x)` – квадратный корень;
- `Math.tan(x)` – тангенс.

Все эти методы являются статическими и вызываются через класс `Math`. Например:

```
x = Math.sqrt(15);
```

В результате в `x` будет помещен квадратный корень пятнадцати.

Класс `Arrays`

Класс `Arrays` содержит набор статических методов для работы с массивами. Обычно для данного класса объектов не создается. Он фактически представляет собой контейнер методов для работы с массивами. В классе имеются следующие методы (следует помнить, что типы элементов массивов, используемых в этих методах, могут быть практически любыми: `int`, `char`, `float` и т. д.):

- `static int binarySearch(int[] a, int key)` – производит поиск в массиве указанного значения бинарным способом. Если массив не является отсортированным, то результаты поиска неопределены;
- `static boolean equals(int[] a, int[] a2)` – сравнивает два массива;
- `static void fill(int[] a, int val)` – назначает указанное значение всем элементам массива. После массива могут быть указаны еще два параметра – номер первого и номер последнего элемента, которым присваиваются значения;
- `static void sort(int[] a)` – сортирует массив. Также можно отсортировать фрагмент массива, указав, вторым и третьим входными параметрами номера первого и последнего элемента, сортируемой части.

Классы-обертки

Так как примитивные типы не являются объектами, в ряде случаев их использовать не получается (в частности, нельзя напрямую хранить переменные примитивных типов в коллекциях). Чтобы обойти данное ограничение, были созданы классы-обертки для примитивных типов.

Данные классы могут использоваться не только для создания объектов, они также хранят некоторые статические методы (например, для преобразования в строки и назад). Чаще всего используются следующие классы-обертки: Byte, Double, Float, Integer, Long, Short. Т.о. обертки есть для всех примитивных типов данных. Следует отметить одну особенность объектов данных классов – значения в объектах не могут меняться. То есть если создан объект типа Integer со значением 5, изменить это значение не возможно. Если следует менять значение, необходимо создавать новый объект.

```
Integer intOb = new Integer(5);  
intOb = new Integer(15);
```

Классы-обертки как правило могут возвращать свое значение, приведенное к требуемому типу. Так для класса Integer имеются следующие методы:

- double doubleValue() – возвращает вещественное удвоенной точности;
- float floatValue() – возвращает вещественное;
- int intValue() – возвращает целое число;
- long longValue() – возвращает длинное целое;
- short shortValue() – возвращает короткое целое.

Также для каждого типа имеется метод для перевода из строкового в числовой тип. Метод является статическим и возвращает значение примитивного типа – static int parseInt(String str)

Также у целочисленных типов имеется набор методов для перевода числа в строку с указанной системой счисления:

- static String toBinaryString(int i)
- static String toHexString(int i)
- static String toOctalString(int i)

Задания

Задание 25

Создать класс описывающий промежуток времени. Сам промежуток в классе должен задаваться тремя свойствами: секундами, минутами, часами. Сделать методы для получения полного количества секунд в объекте, сравнения двух объектов (метод должен работать аналогично compareTo в строках). Создать два конструктора:

- получающий общее количество секунд
- получающий часы, минуты и секунды по отдельности.

Сделать метод для вывода данных. Прочее на ваше усмотрение.

Для заметок:

Задание 26

Создать класс описывающие Банкомат. Набор купюр, находящихся в банкомате, должен задаваться тремя свойствами: количеством купюр номиналом 20, 50, 100. Сделать методы для добавления денег в банкомат. Сделать метод, снимающий деньги. С клавиатуры передается сумма денег. На экран – булевское значение (операция удалась или нет). При снятии денег метод должен выводить на экран каким количеством купюр и какого номинала выдается сумма. Создать конструктор с тремя параметрами – количеством купюр. Прочее – на ваше усмотрение.

Для заметок:

Строки и регулярные выражения

Строки

Любая строка в языке Java является объектом класса String. Класс String представляет набор символов и используется для неизменяемых (immutable) строк. Это касается в том числе и неименованных констант. Поэтому вполне допустимой является запись вида:

```
char x = "Hello World".charAt(5);
```

Метод charAt получает символ, находящийся в строке на указанной позиции. В нашем примере в переменную x попадет значение ' ', то есть пробел, так как нумерация начинается с нуля.

Как уже было сказано, если строка не должна меняться, можно использовать класс String, если в строку необходимо вносить изменения, следует использовать класс StringBuffer.

Большинство методов изменяющих строки возвращает новую строку.

Например:

```
String s = "Test String";  
String s1 = s.substring(5);
```

Метод substring отрезает фрагмент строки начиная с указанной позиции. В результате выполнения этих операций строка в переменной s не изменится, а в s1 попадет строка "String".

Можно занести в переменную s результат работы метода:

```
String s = "Test String";  
s = s.substring(5);
```

Но все равно надо помнить, что исходная строка не изменится, а в переменную будет занесен новый объект.

Также всегда следует учитывать, что строки не являются примитивными типами и их нельзя сравнивать обычными операторами сравнения, такими как `==`. Если надо сравнить две строки, следует воспользоваться указанным ниже методом `compareTo`.

Класс `String` имеет следующие основные методы (они приведены в таблице в том виде, в котором объявлены при создании, то есть сначала указывается возвращаемый тип, затем имя, в скобках перечисляются исходные данные с типами.):

- `char charAt(int index)` – возвращает символ, находящийся в указанной позиции;
- `compareTo(String anotherString)` – сравнивает строку с другой строкой (сравнивается именно текст содержащийся в строках);
- `compareToIgnoreCase(String str)` – то же самое, но без учета регистра;
- `String concat(String str)` – возвращает объект строки, содержащий сумму данной строки, и переданной как аргумент;
- `boolean contentEquals(StringBuffer sb)` – сравнивает содержимое объекта `String` и `StringBuffer`;
- `boolean endsWith(String suffix)` – проверяет, завершается ли строка заданным суффиксом (совпадает ли конец строки с переданной);
- `boolean equals(Object anObject)` – сравнивает строку с объектом `boolean`;
- `equalsIgnoreCase(String anotherString)` – сравнивает строки игнорируя регистр;
- `byte[] getBytes()` – возвращает строку в виде массива байтов (в качестве входного параметра указывается название требуемой кодировки);
- `int indexOf(int ch)` – ищет в строке переданный символ и возвращает позицию первого совпадения. Может иметь два параметра, тогда вторым указывается номер, с которого надо начинать поиск. Вместо символа может быть также строка;
- `int lastIndexOf(int ch)` – аналогично предыдущему, но поиск выполняется с конца;
- `int length()` – возвращает длину строки;
- `String replace(char oldChar, char newChar)` – возвращает строку, где все символы, совпадающие с первым, заменены вторым;
- `boolean startsWith(String prefix)` – проверяет начинается ли строка с данного префикса (совпадает ли начало строки с переданной);
- `String substring(int beginIndex)` – возвращает строку, содержащую фрагмент данной строки. Может иметь два параметра: номер первого символа и номер последнего символа (второй может отсутствовать);
- `char[] toCharArray()` – возвращает массив символов, содержащий данную строку;
- `String toLowerCase()` – возвращает строку, содержащую копию данной, приведенную к нижнему регистру;
- `String trim()` – возвращает строку с удаленными начальными и конечными пробелами;
- `boolean isBlank()` – возвращает `true`, если строка пуста или содержит только пробелы, иначе `false`;
- `Stream lines()` – возвращает поток строк, извлеченных из этой строки, разделенных терминаторами строк;
- `String repeat (int)` – возвращает строку, значение которой представляет собой конкатенацию этой строки, повторяющуюся `int` раз;

- `String strip()` – возвращает строку, из которой удалены все пробелы, которые находятся до первого символа, не являющегося пробелом, или после последнего;
- `String stripLeading()` – возвращает строку, из которой удалены все пробелы, которые находятся до первого символа, не являющегося пробелом;
- `String stripTrailing()` – возвращает строку, из которой удалены все пробелы, которые находятся после последнего символа, не являющегося пробелом.

Примеры работы со строками

Имеется задача: ввести строку и подсчитать количество запятых в ней.

Данная задача может быть решена двумя способами: перебором всех элементов строки, либо с использованием метода поиска.

Перебор всех элементов строки будет выглядеть следующим образом:

```
String str = "Тестовая, строка, с несколькими,, запятыми";
int n = 0;
char symbol;
for(int i=0; i < str.length(); i++){
    symbol = str.charAt(i);
    if(symbol == ',') {
        n++;
    }
}
System.out.println("У нас есть " + n + " запятых");
```

Сначала создается строка `str`, `n` – количество запятых в строке, так как изначально запятые могут отсутствовать, начальным значением будет 0. С помощью `charAt` получаем каждый элемент строки и сравниваем его с запятой, увеличивая `n`, если обнаружено совпадение.

Подсчет с помощью поиска будет выглядеть следующим образом:

```
String str = "Тестовая, строка, с несколькими,, запятыми";
int n = 0;
int p = 0;
while(p != -1) {
    p = str.indexOf(',', p);
    if(p != -1) {
        p++;
        n++;
    }
}
System.out.println("У нас есть " + n + " запятых");
```

Подсчет заканчивается, если `indexOf` вернула -1, если возвращено другое число – значит, что в тексте найдена запятая, в этом случае увеличивается счетчик `n` и увеличивается `p` для того, чтобы следующий поиск выполнялся после найденного знака.

Работа со строками `String`, `StringBuffer` и `StringBuilder`

Конструкторы `StringBuffer`:

- `StringBuffer();`
- `StringBuffer(String str).`

Конструкторы `StringBuilder`:

- `StringBuilder ();`
- `StringBuilder(String str).`

Когда использование `String` не является желательным

Базовым классом для работы со строковыми данными является `String`, но его использование не всегда оправдано. Связано это в первую очередь с тем, что строка, содержащаяся в объекте `String`, не может меняться. При изменении содержимого строки создается новый объект. В ряде случаев это может привести к большим потерям в производительности. В частности, операция вида: `str += "добавление строки";` приводит к тому, что создается новый объект, и содержимое обоих исходных строк в него копируется. Если подобные операции используются в больших количествах или в цикле, то это может привести экспоненциальному падению производительности операций.

Для решения этой проблемы следует использовать объект типа `StringBuffer` или `StringBuilder`. Оба эти объекта позволяют менять содержимое находящихся в них строк. С их использованием операция примет вид:

```
strBuilder.append(" добавление строки");
```

В этом случае не создается новый объект и копируется только добавляемая строка.

Сходства и отличия `StringBuffer` и `StringBuilder`

Оба класса имеют одинаковый набор методов и используются в сходных ситуациях, за одним исключением. `StringBuilder` не рассчитан на использование в многопоточных приложениях и может приводить к ошибкам, если используется в нескольких потоках одновременно. С другой стороны, отсутствие синхронизации увеличивает скорость его работы.

Таким образом, если вы уверены, что ваш объект будет использоваться только в одном потоке, желательно использовать `StringBuilder`, в противном случае больше подходит `StringBuffer`.

Регулярные выражения

Основы

Регулярные выражения используются в случае, если необходимо выполнить проверку текста в строке на соответствие определенному шаблону. Например, в

тексте необходимо найти числа, в том числе отрицательные, то есть последовательности цифр, которые могут начинаться со знака минус.

Регулярное выражение, описывающее такое сочетание знаков, можно записать несколькими способами:

```
-?[0123456789]+
```

```
-?[0-9]+
```

```
-?\\d+
```

Все три варианта начинаются с сочетания `-?`. Знак вопроса означает, что стоящий перед ним символ может встретиться один раз или ни разу. Последовательность знаков в квадратных скобках означает, что это может быть любой из перечисленных знаков. Знак плюс после скобок означает, что такой знак должен встречаться один или более раз.

Если знаки идут в кодовой таблице подряд, можно указать первый и второй через тире, что и было использовано во втором примере. В третьем примере используется спецсимвол `\\d`, указывающий, что это должна быть цифра.

Для поиска чисел можно использовать любой из приведенных шаблонов.

Методы класса String

В простейшем случае, можно воспользоваться встроенным в класс String методом `matches`.

```
System.out.println("20934".matches("-?[0-9]+"));
```

Результатом выполнения этого выражения будет вывод `true`.

Метод `matches` получает на вход регулярное выражение и проверяет соответствие всей строки целиком этому выражению. Результатом будет логическое значение истина или ложь.

Еще один полезный метод, использующий регулярные выражения, – метод `split`. Данный метод разрезает строку на части, и возвращает массив. На вход метод получает регулярное выражение, строка разрезается там, где ее фрагменты соответствуют выражению.

```
String s = "Test string for split";  
System.out.println(Arrays.toString(s.split(" +")));
```

Результатом будет вывод массива:

```
[Test, string, for, split]
```

В данном случае строка была разрезана там, где встречаются пробелы. Регулярное выражение позволило исключить влияние нескольких пробелов. Также следует обратить внимание на то, что фрагмент строки, являющийся разделителем (пробелы в нашем случае) вырезается из конечного результата.

Еще два метода, использующие регулярные выражения, – это методы `replaceFirst` и `replaceAll`. Эти методы позволяют выполнить поиск и замену в строке. Первым аргументом будет регулярное выражение:

```
System.out.println("Test multiple spaces".replaceAll(" +", " "));
```

Результатом такого выражения будет вывод:

```
Test multiple spaces
```


Таким образом, метод `replaceAll` находит фрагмент строки, соответствующий регулярному выражению, полученному первым аргументом, и заменяет его строкой, полученной как второй аргумент. В нашем случае произвольное количество пробелов заменяется единичным пробелом. Метод `replaceFirst` работает аналогично, но заменяет только первое найденное совпадение.

Классы `Pattern` и `Matcher`

Более мощные средства для работы с регулярными выражениями предлагают классы `Pattern` и `Matcher`. Класс `Pattern` служит для хранения регулярного выражения, а `Matcher` служит для выполнения операций поиска и сравнения.

Для их использования следует сначала задать регулярное выражение и создать объект `Pattern`. Например, необходимо найти в тексте все тире, окруженные пробелами:

```
Pattern p = Pattern.compile(" +- +");
```

Объект создается статическим методом `compile`, получающим на вход регулярное выражение. Затем следует создать объект класса `Matcher`, указав, строку, с которой будут выполняться все последующие операции поиска:

```
Matcher m = p.matcher("Test - string - test");
```

Объект `Matcher` создается методом `matcher`, получающим на вход строку. После этого можно выполнять различные операции с данной строкой и регулярным выражением.

Одна из наиболее часто используемых операций – поиск. Поиск выполняется с помощью метода `find`:

```
m.find()
```

Метод ищет соответствие заданному регулярному выражению в строке и возвращает истину, если соответствие найдено, и ложь – если нет. При повторном вызове данного метода он продолжает поиск и возвращает истину, если найдено еще одно соответствие.

Местоположение найденного совпадения можно найти с помощью методов `start` и `end`, которые возвращают позиции начала соответствия и конца. Например:

```
while(m.find()) {  
    System.out.println(m.start() + " " + m.end());  
}
```

В данном примере находятся соответствия и выводятся через пробел позиции их начала и конца. Для нашей строки результат будет таким:

```
4 7  
13 16
```

Регулярным выражениям, как правило, могут соответствовать разные строки. Для того, чтобы узнать, как выглядит очередная найденная методом `find` строка, можно использовать метод `m.group()`.

Для объекта класса `Matcher` имеются также:

- метод `matches`, проверяющий соответствие строки регулярному выражению;
- метод `lookingAt`, ищущий соответствие только в самом начале строки, с ее первого знака.

Если все операции с первоначальной строкой закончены и надо работать с новой, то не обязательно создавать новый объект класса `Matcher`, можно воспользоваться методом `reset`:

```
m.reset("New string for search");
```

Синтаксис регулярных выражений

Специальные символы:

- `\xhh` – символ с шестнадцатеричным кодом `0xhh` `\uhhhh` Символ Unicode с шестнадцатеричным кодом `0xhhhh`;
- `\t` – табуляция;
- `\n` – новая строка;
- `\r` – возврат каретки.

Классы символов:

- `.` – любой символ;
- `[abc]` – любой из указанных в скобках знаков `a`, `b` или `c`;
- `^[abc]` – любой символ, кроме `a`, `b` или `c`;
- `[a-zA-Z]` – любой символ начиная от `a` заканчивая `z`, а также от `A` до `Z`;
- `[abc[hij]]` – любой из символов `a,b,c,h,i,j`;
- `\s` – пробельный символ (пробел, перевод строки, табуляция и т.д.);
- `\S` – любой не пробельный символ;
- `\d` – цифра, то же самое, что `[0-9]`;
- `\D` – не цифра, то же самое, что `^[0-9]`;
- `\w` – буква, то же самое, что `[a-zA-Z_0-9]` ;
- `\W` – не буква `^[^w]`.

Обозначения границ:

- `^` – начало строки;
- `$` – конец строки;
- `\b` – граница слова;
- `\B` – не граница слова;
- `\G` – конец предыдущего соответствия.

Группировка:

- `X?` – ни одного или один раз;
- `X*` – любое количество;
- `X+` – раз один или больше;
- `X{n}` – точно `n` раз (`n` – число);
- `X{n,}` – `n` и больше раз;
- `X{n,m}` – `n` или больше, но не больше `m`.

Часть регулярного выражения можно выделить в группу, для этого его следует взять в круглые скобки. При нахождении соответствия, указав методу `group` номер, начиная с единицы, можно получить не все соответствие, а только одну из его групп.

Задания

Задание 27

Введите с клавиатуры строку. Найти в строке не только запятые, но и другие знаки препинания. Подсчитать общее их количество.

Для заметок:

Задание 28

Введите с клавиатуры текст. Подсчитать количество слов в тексте. Учесть, что слова могут разделяться несколькими пробелами, в начале и конце текста также могут быть пробелы, но могут и отсутствовать.

Для заметок:

Задание 29

Введите с клавиатуры текст. Выведите на экран текст, составленный из последних букв всех слов из исходного текста.

Для заметок:

Задание 30

Введите с клавиатуры строку. Напишите метод, выполняющий поиск в строке шестнадцатеричных чисел, записанных по правилам Java, с помощью регулярных выражений. Результат работы метода выведите на экран.

Для заметок:

Задание 31

Введите с клавиатуры строку. Напишите метод, выполняющий поиск в строке всех тегов абзацев, в том числе тех, у которых есть параметры, например, `<p id="p1">`, и замену их на простые теги абзацев `<p>`. Результат работы метода выведите на экран.

Для заметок:

Задание 32

Напишите два цикла выполняющих миллион сложений строк вида “aaabbbccc”, один с помощью оператора сложения и String, а другой с помощью StringBuilder и метода append. Сравните скорость их выполнения. Выведите сравнение на экран.

Для заметок:

Наследование и полиморфизм. Внутренние и анонимные классы

Наследование и полиморфизм

Что такое наследование

Как и большинство объектно-ориентированных языков Java позволяет использовать наследование. При наследовании вы можете создать новый класс не с нуля, а на основе существующего класса. При этом новый класс получает те же свойства и методы, которые были объявлены в старом классе, кроме приватных. Новый класс при этом называют дочерним классом, а старый – родительским. В дочернем классе обычно описываются новые свойства и методы, которые необходимо добавить к тому, что есть в родительском. Кроме того, в ряде случаев дочерний класс может заменять полученные от родительского класса методы своими вариантами. По умолчанию любой класс наследуется от класса Object.

Чтобы сделать класс дочерним от другого класса, следует при его создании после его имени указать слово `extends` и имя родительского класса. Например, возьмем класс:

```
public class Point {  
    public int x;  
    public int y;  
}
```

И нам надо добавить цвет:

```
public class ColorPoint extends Point {  
    private int color;  
}
```

Таким образом, класс ColorPoint получает все свойства и методы, которые были в Point, и при этом добавляет к ним собственное свойство color.

Надо учитывать, что, если какие-то свойства или методы в предке имели модификатор `private`, дочерний класс напрямую видеть их не может. То есть его методы не могут обращаться к закрытым свойствам и методам, унаследованным от родителя.

Если вы хотите в классе закрыть свойства или методы от посторонних, но при этом дать доступ дочерним классам, следует использовать модификатор не **private**, а **protected**. Тогда ваш дочерний класс сможет работать с ними напрямую.

При этом, если необходимо запретить наследование от создаваемого класса, это можно сделать, указав при его создании модификатор **final**. Если заголовок класса **Point** выглядел как:

```
public final class Point {  
    }
```

Попытка создать класс **ColorPoint** вызовет ошибку.

Последовательность вызовов конструкторов родительского и дочернего классов следующая: сначала вызывается конструктор родительского класса, а затем дочернего.

Если возникает необходимость вызвать конструктор родительского класса, он вызывается с помощью ключевого слова **super** и входных параметров.

```
super (входные параметры);
```

Если родительский класс не имеет конструктора без параметров, явный вызов родительского конструктора с помощью **super** является обязательным, причем вызов **super** должен стоять в дочернем конструкторе самым первым.

```
class A {  
    private int y;  
    A(int x){  
        this.y = x;  
    }  
}  
  
class B extends A {  
    private int x;  
    B(){  
        super(1);  
        this.x = 0;  
    }  
}
```

Следует также отметить, что в Java отсутствует множественное наследование классов.

Переопределение методов

Вы можете в дочернем классе создавать методы с таким же именем, типом возвращаемого значения и набором параметров, как и в родительском классе. Тем самым вы переопределяете метод. Этот процесс называется по-английски **overriding**.

Любой класс, создаваемый в вашей программе, даже если вы явно не использовали наследование будет иметь предка: класс **Object**. И даже пустой класс будет содержать методы, унаследованные от него. Один из самых полезных методов, наследуемых из этого класса **toString**. Данный метод автоматически

вызывается, когда надо преобразовать ваш класс в строку. Например, если вы написали:

```
Point p = new Point();  
System.out.println(p);
```

В этом случае будет вызван `toString` (из класса `Object`, который наследуют все классы) и распечатан результат его работы. Вы можете переопределить `toString`, чтобы при распечатке выводилась нужная вам информация.

```
public String toString() {  
    return "А это мой класс!";  
}
```

Если добавить к методу модификатор `final`, то это запретит его переопределение.

Перегрузка методов

Вы можете в одном классе создавать методы с таким же именем, типом возвращаемого значения, но разным набором параметров. Тем самым вы перегружаете метод, этот процесс называется по-английски `overload`.

Например, если вы написали:

```
public class A {  
    int average(int x, int y){  
        return (x + y)/2;  
    }  
    int average(int x, int y, int z){  
        return (x + y + z)/3;  
    }  
}
```

В этом случае метод `average` будет перегружен и при его вызове в зависимости от количества входных параметров JVM будет вызван соответствующий метод. На перегрузку влияет как количество, так и типы параметров. Если JVM не сможет понять, какой конкретно метод нужно вызвать, то программа не скомпилируется.

Пакеты

Когда в программе классов становится много, то их имеет смысл сгруппировать. Для этого используются пакеты. Физически пакет – каталог на диске, в котором лежат исходные и скомпилированные файлы класса.

Если класс не лежит в корневом каталоге, а положен в пакет, то в самом начале исходного текста следует указать имя пакета при помощи ключевого слова `package`.

```
package packet;
```

Если это записано первой строкой исходника, значит он должен лежать не в корневом каталоге проекта, а в каталоге с именем `packet`.

Пакеты, как и каталоги могут вкладываться друга. В этом случае записывается вся цепочка пакетов, разделяемая точками.

```
package com.packet;
```

Это означает, что класс лежит в каталоге packet, который лежит в каталоге com.

Если в вашей программе используются классы из других пакетов, вы обязаны их подключить с помощью команды import.

```
import com.packet.MyClass;
```

Это означает, что мы будем использовать класс MyClass в пакете com.packet.

Для множественного импортирования классов используется:

```
import com.packet.*;
```

Все классы из стандартной библиотеки java разложены по пакетам и большинство из них требуют подключения с помощью import. Исключением из этого правила являются классы, расположенные в пакете java.lang. Классы типа String, System, Math являются исключениями.

При объявлении переменных и методов класса может не быть никакой тип доступа, это значит, что доступ для этого класса и других классов в этом же пакете.

Доступ protected также дает доступ классам из этого же пакета.

Для импортирования статических констант используется следующее выражение:

```
import static java.lang.Math.abs;
```

Абстрактные классы

Java позволяет создавать классы, предназначенные только для наследования, то есть классы, которые не позволяют создавать объекты, а только позволяют их унаследовать при создании других классов. Такие классы называются абстрактными. Для того, чтобы указать, что класс будет абстрактным, в заголовке класса указывается ключевое слово abstract. Абстрактный класс может иметь и полностью описанные методы, и абстрактные методы, у которых отсутствует тело, и которые должны быть реализованы в дочерних классах. Такие методы должны быть описаны с ключевым словом abstract, а вместо тела метода ставится знак точки с запятой. Например, описание абстрактного класса геометрической фигуры:

```
abstract class Fig {  
    protected float x;  
    protected float y;  
    float getX(){  
        return x;  
    }  
    float getY(){  
        return y;  
    }  
    void move(float dx, float dy){  
        x += dx;  
        y += dy;  
    }  
}
```



```

    }
    abstract float calcArea();
    abstract void getType();
}

public class Rectangle extends Fig {
    private float h;
    private float w;
    public Rectangle() {
        x = 1;
        y = 1;
        w = 1;
        h = 1;
    }
    public float calcArea() {
        return h*w;
    }
    public String getType() {
        return "Прямоугольник";
    }
}

public class MyFirst {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        System.out.println(r1.getType());
        System.out.println("Area = " + r1.calcArea());
    }
}

```

В данном примере создан абстрактный класс figure, в котором имеются абстрактные методы calcArea и getType, а также обычные свойства и методы x, y, move, getX, getY. Затем был создан дочерний класс Rectangle, описывающий прямоугольник. В нем были добавлены свойства w и h, а также перегружены абстрактные методы родительского класса.

Следует обратить внимание, что для создания обычного класса на основе абстрактного следует перегрузить все абстрактные методы. В противном случае компилятор выдаст ошибку, сообщающую, какие методы еще не перегружены.

Интерфейсы

Существует еще один особый тип, который является еще более абстрактным, чем абстрактные классы. Это интерфейсы. Если абстрактный класс может хотя бы частично содержать реализацию класса (в нем могут присутствовать обычные свойства и методы), то интерфейс описывает только то, как класс будет взаимодействовать с другими классами. Поэтому в нем могут содержаться абстрактные методы, а также `default` и `private` методы. Описание обычных методов в интерфейсе не допустимо.

Общий вид описания интерфейса аналогичен описанию класса. В начале указывается ключевое слово `interface` (вместо `class`), затем имя интерфейса и в фигурных скобках его содержимое. Например, интерфейс, описывающий лампочку:

```
interface Lamp {
    void on();
    void off();
    void printState();
    public default void light() {
        light1();
        light2();
        System.out.println("default method");
    }
    private void light1(){
        System.out.println("private method");
    }

    private static void light2(){
        System.out.println("private static method");
    }
}
```

Несмотря на то, что все методы являются абстрактными, ключевое слово `abstract` здесь может не использоваться. Оно используется по умолчанию, также следует помнить, что так как интерфейс описывает взаимодействие с другими классами, то методы по умолчанию имеют тип доступа не пакетный, а открытый (`public`).

Интерфейсы могут содержать не только методы, но и поля. Но при этом на поля налагаются существенные ограничения. В частности, свойства могут быть только статическими и константными. Даже если это не указано, свойства являются `static` и `final`. При объявлении свойства ему обязательно присвоить начальное значение. Например, если все типы лампочек имеют одинаковый срок годности, то в интерфейс можно добавить свойство следующего вида:

```
int lifeTime = 12;
```

Данное свойство будет константным, то есть изменить значение, заданное в интерфейсе, невозможно. И будет доступно через сам интерфейс, без создания дочерних классов и их объектов.

```
System.out.println(Lamp.lifeTime);
```

Это позволяет иногда использовать интерфейсы, как контейнеры для набора связанных между собой констант.

Для интерфейсов возможно использование наследования. Наследование производится аналогично наследованию между классами.

```
interface VolumeLamp extends Lamp {  
    void setVolume();  
}
```

То есть интерфейс VolumeLamp будет описывать те же свойства и методы, что и Lamp, но при этом добавит свой метод setVolume.

Все отличия интерфейсов от абстрактных классов, рассмотренные до этого, не являются существенными. Главным отличием интерфейсов от классов является возможность множественного наследования. Допустим, что кроме описанного выше интерфейса Lamp, имеется интерфейс Color:

```
interface Color {  
    void setColor();  
}
```

На основе этих интерфейсов можно создать интерфейс ColorLamp, следующим образом:

```
interface ColorLamp extends Lamp, Color {  
}
```

То есть, чтобы итоговый интерфейс содержал все свойства и методы из нескольких родительских, ему следует указать имена этих интерфейсов через запятую. В данном примере интерфейс не имеет собственных свойств и методов, а только унаследованные, но собственные могут быть также добавлены обычным образом.

После создания интерфейса на его основе может быть создан класс. Создание класса из интерфейса аналогично созданию обычного дочернего класса, но вместо слова extends используется слово implements.

```
public class OrdLamp implements Lamp {  
    protected boolean state = false;  
    public void on(){  
        this.state = true;  
    }  
    public void off(){  
        this.state = false;  
    }  
    public void printState() {  
        if(state) {  
            System.out.println("Лампа включена!");  
        } else {  
            System.out.println("Лампа выключена!");  
        }  
    }  
}
```

Следует обратить внимание, что при переопределении методов, описанных в интерфейсе, в классе обязательно явным образом указывать правильный тип доступа (в данном случае `public`), хотя в интерфейсе он использовался по умолчанию. При создании класса может одновременно применяться наследование от другого класса и интерфейсов:

```
public class ColorLamp extends OrdLamp implements Color {
    protected int color;
    public void setColor(){
        this.color = 1;
    }
}
```

Внутренние классы

Если объекты класса могут быть использованы только внутри другого класса, иногда имеет смысл создавать сам класс внутри другого.

В Java существуют 4 типа вложенных (nested) классов:

- статические вложенные классы;
- внутренние классы;
- локальные классы;
- анонимные (безымянные) классы.

Как пример рассмотрим внутренний класс:

```
class ClassA {
    class ClassB {
        int x = 1;
    }
    void printData() {
        ClassB b = new ClassB();
        System.out.println("inner"+b.x);
    }
}
```

В данном примере создан класс `ClassA`, внутри которого создан класс `ClassB`. Другие классы не могут, минуя `ClassA` создавать объекты класса `ClassB`. Чтобы внешний класс мог создать объект внутреннего класса, он должен создать объект основного класса. Например:

```
ClassA a = new ClassA();
ClassA.ClassB b = a.new ClassB();
```

Следует обратить внимание, что оператор `new` вызывается как метод объекта основного класса.

Такая запись возможна, только если внутренний класс имеет тип доступа, открытый или пакетный. Если внутренний класс создан с ключевым словом `private`, создать объект внутреннего класса таким образом невозможно.

Другой класс может получить доступ к внутреннему классу с типом доступа `private`, только если внутренний класс был создан на основе интерфейса, и в основном классе есть метод, возвращающий объект внутреннего класса.

```
interface IntB {
    int getX();
}

class ClassA {
    private class ClassB implements IntB {
        int x=1;
        public int getX(){
            return x;
        }
    }
    void printData() {
        ClassB b = new ClassB();
        System.out.println("inner"+b.x);
    }
    IntB getB(){
        return new ClassB();
    }
}

...

ClassA a = new ClassA();
IntB b = a.getB();
System.out.println(b.getX());
```

Как видно, внутренний класс является закрытым, но тем не менее имеется возможность через интерфейс получить доступ к его интерфейсным методам. Подобный подход использован в итераторах классов-коллекций.

Анонимные классы

В случае, если объект данного класса создается исключительно в одном месте, можно не создавать отдельный внутренний класс, а сделать так называемый анонимный класс, который описывается прямо на месте создания объекта.

Анонимный класс на основе интерфейса или другого класса создается следующим образом:

```
Интерфейс имя_переменной = new Интерфейс() {
    Здесь располагается описание класса.
}
```

Таким образом, если рассматривать приведенный ранее пример, для итератора можно сделать анонимный класс. В этом случае метод интерфейса будет иметь вид:

```
interface IntB {  
    int getX();  
}  
  
...  
IntB intB = new IntB {  
    int getX() {  
        return 5;  
    }  
}
```

И специально объявлять класс, имплементирующий интерфейс IntB, уже нет необходимости. Анонимные классы достаточно широко используются в программах с графическим интерфейсом пользователя.

Задания

Задание 33

Создать цепочку наследования (минимум 3 звена) классов, описывающих бытовую технику. Создать несколько объектов описанных классов, часть из них включить в розетку. Иерархия должна иметь хотя бы три уровня.

Для заметок:

Задание 34

Создать цепочку наследования (минимум 3 звена) классов, описывающих банковские карточки. Иерархия должна иметь хотя бы три уровня.

Для заметок:

Задание 35

Создать цепочку наследования (минимум 3 звена) классов, описывающих должностную структуру на заводе. Реализовать методы по начислению зарплаты в зависимости от должности (почасовая, процентная, смешанная). Иерархия должна иметь хотя бы три уровня.

Для заметок:

Дженерики. Перечисляемые типы

Дженерики

В ряде задач внутри класса могут использоваться ссылки на объекты разных классов. Причем в разных обстоятельствах используемые классы могут быть разными. Допустим, что в одной части задачи нам необходимо иметь ссылку на объект класса `String`, а в другой – `Integer`. Без использования дженериков единственное решение – хранить объект в виде переменной общего родительского класса. Самым универсальным решением является использование типа `Object` для переменной. В этом случае переменная может хранить ссылку на объекты любого класса, так как `Object` является предком всех классов.

Но если вы храните объект в переменной `Object`, то при его получении возникает проблема приведения типов и определения типа объекта. Если вы положили объект одного типа, а пытаетесь извлечь другого, то у вас будут ошибки времени выполнения. Например, у нас имеется класс:

```
public class Box {  
    private Object item;  
    public Object getItem() {  
        return item;  
    }  
    public void setItem(Object item){  
        this.item = item;  
    }  
}
```

Если мы попытаемся им воспользоваться таким образом:

```
Box box = new Box();  
box.setItem("Test");  
.....  
Object item = box.getItem();  
Integer itemInt = (Integer)item;
```


Во время компиляции никаких ошибок возникать не будет, а во время выполнения вы получите ошибку выполнения `ClassCastException`, так как в объекте хранится `String`, а вы пытаетесь его занести в переменную `Integer`.

Решить подобную проблему можно, изменив класс `Box` следующим образом:

```
public class Box<T> {  
    private T item;  
    public T getItem() {  
        return item;  
    }  
    public void setItem(T item){  
        this.item = item;  
    }  
}
```

Это и называется дженериком. При использовании этого класса его объект надо создавать с указанием в угловых скобках класса хранимого элемента. Причем делается это обычно два раза – при объявлении переменной и при создании объекта с помощью `new`.

```
Box<String> myBox = new Box<>();  
myBox.setItem("Test");  
....  
String s = myBox.getItem();
```

Чем эта ситуация отличается от предыдущей. Во-первых, вам не позволят сделать присвоение переменной неправильного типа. Если вы здесь напишете `Integer s` вместо `String s`, ошибку вы получите еще на стадии компиляции. Во-вторых, сразу возвращается значение правильного типа и вам не требуется лишний раз делать приведение типов в вашей программе.

Перечисляемые типы

Перечисление (`enum`) – это тип, значения которого ограничены конечным набором констант.

Ранее в рамках класса мы рассматривали константы в таком виде:

```
public final class SomeConstants {  
    public static final int ORDER_ONE = 1;  
    public static final int ORDER_TWO = 2;  
    public static final int ORDER_THREE = 3;  
}
```

В Java предоставлена реализация абстрактного класса `Enum` (`java.lang.Enum`), так же для простоты существует ключевое слово `enum`.

Вернемся к нашему примеру:

```
public enum SomeConstants {  
    ORDER_ONE, ORDER_TWO, ORDER_THREE  
}
```

Запись "public enum SomeConstants" равноценна "abstract class SomeConstants extends java.lang.Enum". Хотя класс объявлен без модификатора final, дальше расширять класс нельзя (запрет реализован на уровне компилятора).

Также в этот класс мы можем добавить:

- поля;
- конструкторы;
- методы;
- статические методы.

Пример:

```
public enum SomeConstants {  
    ORDER_ONE(1), ORDER_TWO(2), ORDER_THREE(3);  
  
    SomeConstants(int id) {  
        this.id = id;  
    }  
  
    private int id;  
  
    public int getId() {  
        return id;  
    }  
}
```

Задания

Задание 36

Создать enum, который описывает сезоны года. Добавить поле description в этот enum. Добавить поле countOfDays в этот enum. Вывести на экран все константы сезоны года.

Для заметок:

Задание 37

Написать метод, который выводит следующий сезон от заданного во входном параметре. Входной параметр ввести с клавиатуры.

Для заметок:

Задание 38

Написать метод, который в зависимости от сезона, выводит на экран сумму дней в этом сезоне. Входной параметр ввести с клавиатуры.

Для заметок:

Задание 39

Создать классы Car и Motorcycle, которые наследуются от общего класса Vehicle. Создать поле name в классе Vehicle и проинициализировать его через конструктор. Создать generic класс Garage, который может хранить только объекты типа наследуемого от Vehicle. Создать 2 объекта класса Garage (все поля ввести с клавиатуры) и вывести на экран имя хранимого транспортного средства.

Для заметок:

Коллекции. List, Set, Map

Для динамических структур данных, в Java используются контейнерные классы. Обычно они называются коллекциями.

Java Collections Framework состоит из следующих интерфейсов:

- List – это упорядоченный список объектов (иногда еще называют последовательностью объектов). Элементы списка могут вставляться или извлекаться по их индексу в списке (индексация начинается с 0). В эту группу входят: ArrayList, LinkedList;
- Set – это неупорядоченная коллекция. Главная особенность множеств – уникальность элементов, то есть один и тот же элемент не может содержаться в множестве дважды. Есть такие имплементации Set интерфейса: HashSet, TreeSet, LinkedHashSet, EnumSet. HashSet хранит элементы в хэш-таблице, что предполагает эффективную реализацию, но не гарантирует порядок итерации элементов. TreeSet хранит элементы в красно-чёрном дереве и упорядочивает элементы по их значениям, эта коллекция существенно медленнее, чем HashSet. LinkedHashSet реализована как хэш-таблица и хранит элементы в связанном списке, в порядке, котором они были вставлены;
- Map (иногда можно встретить названия: отображения, словарь, ассоциативный массив, карты) – отображает ключи в значения и не может иметь дубликаты ключей. Есть три основных имплементации Map интерфейса: HashMap, TreeMap и LinkedHashMap. HashMap не

гарантирует воспроизводимость порядка вставки элементов. TreeMap хранит элементы в красно-чёрном дереве и упорядочивает элементы по их ключам и медленнее, чем HashMap. LinkedHashMap упорядочивает элементы в порядке их вставки.

Коллекции типа List

ArrayList

Для создания массивов изменяющейся длины используют класс ArrayList. Конструктор данного класса может быть без параметров, в этом случае создается массив без элементов. Он может иметь на входе другой массив (коллекцию), тогда он заполняется элементами из переданной ему коллекции. Третий вариант конструктора имеет один входной параметр – число, задающее под сколько элементов, должна быть выделена память, исходно (сами элементы при этом не создаются).

Пример:

```
List arrayList = new ArrayList();
```

Следует обратить внимание, что контейнерные классы не являются строго типизированными. Как видно из примера, тип элементов, хранящихся в коллекции, никак не задается. Но тем не менее, некоторые ограничения существуют: контейнерные классы не могут хранить объекты примитивных типов. Поэтому для их хранения следует пользоваться типами – обертками.

Для работы с ArrayList можно использовать следующие методы:

- void add(int index, Object element) – добавляет в массив элемент на заданную позицию;
- boolean add(Object o) – добавляет элемент в конец списка;
- boolean addAll(Collection c) – добавляет элементы из указанной коллекции к концу массива;
- boolean addAll(int index, Collection c) – вставляет элементы из указанной коллекции в точку массива с указанным номером;
- void clear() – удаляет все элементы из массива;
- boolean contains(Object elem) – возвращает истину, если список такой же элемент;
- Object get(int index) – возвращает ссылку на элемент с указанным номером;
- int indexOf(Object elem) – возвращает номер первого элемента, который равен указанному;
- boolean isEmpty() – возвращает истину, если массив не содержит элементов;
- int lastIndexOf(Object elem) – возвращает последний элемент, равный указанному;
- Object remove(int index) – удаляет элемент с заданным номером;
- protected void removeRange(int fromIndex, int toIndex) – удаляет набор элементов, начиная с первого номера (включительно) и до второго (он не удаляется);
- Object set(int index, Object element) – заменяет элемент с указанным номером на переданный;

- `int size()` – возвращает количество элементов в массиве;
- `Object[] toArray()` – возвращает обычный массив, хранящий все элементы в правильном порядке.

LinkedList

`LinkedList` представляет структуру данных в виде связанного списка.

Класс `LinkedList` имеет следующие конструкторы:

- `LinkedList()` – создает пустой список;
- `LinkedList(Collection<? extends E> collection)` – создает список, в который добавляет все элементы коллекции `collection`.

Пример:

```
List linkedList = new LinkedList();
```

Данный класс имеет практически такой же набор методов, как и `ArrayList`. Разница в основном заключается в скорости выполнения разных операций, но имеется также и несколько своих методов:

- `Object getFirst()` – возвращает первый элемент списка;
- `Object getLast()` – возвращает последний элемент списка;
- `ListIterator listIterator(int index)` – возвращает итератор списка, указывающий на элемент с данным номером;
- `Object removeFirst()` – удаляет первый элемент списка;
- `Object removeLast()` – удаляет последний элемент списка.

Коллекции типа Set

Для реализации данного типа коллекции чаще всего используется класс `HashSet`. Класс `HashSet` имеет следующие конструкторы:

- `HashSet()` – создает пустой список;
- `HashSet(Collection<? extends E> collection)` – создает хеш-таблицу, в которую добавляет все элементы коллекции `collection`;
- `HashSet(int capacity)` – параметр `capacity` указывает начальную емкость таблицы, которая по умолчанию равна 16;
- `HashSet(int capacity, float coef)` – параметр `coef` или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами, прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение.

Особенности `HashSet`:

- `HashSet` не поддерживает порядок своих элементов, а это значит, что элементы будут возвращены в любом порядке;
- `HashSet` не разрешает хранить дубликаты. Если вы добавите существующий элемент, то старое значение будет переписано;
- `HashSet` разрешает добавить в коллекцию `null` значение, но только одно.

Пример:

```
Set set = new HashSet();
```

Данные коллекции позволяют выполнять очень ограниченное количество операций, в частности – добавлять элемент, проверять, имеется ли в коллекции такой элемент, и удалять элемент:

- `add(Object o)` – добавляет элемент в коллекцию;
- `boolean contains(Object o)` – проверяет, есть ли элемент в коллекции `boolean`;
- `remove(Object o)` – удаляет элемент из коллекции.

Коллекции типа Queue

Интерфейс `java.util.Queue` представляет собой структуру данных, предназначенную для вставки элементов в конец очереди и удаления элементов из начала очереди.

Пример:

```
Queue<String> queue = new PriorityQueue<>();
```

Основные методы:

- `E element()` – возвращает, но не удаляет элемент из начала очереди, если очередь пуста, генерирует исключение `NoSuchElementException`;
- `boolean offer(E obj)` – добавляет элемент `obj` в конец очереди, если элемент удачно добавлен, возвращает `true`, иначе – `false`;
- `E peek()` – возвращает без удаления элемент из начала очереди, если очередь пуста, возвращает значение `null`;
- `E poll()` – возвращает с удалением элемент из начала очереди, если очередь пуста, возвращает значение `null`;
- `E remove()` – возвращает с удалением элемент из начала очереди, если очередь пуста, генерирует исключение `NoSuchElementException`.

Карты (Map)

Для реализации ассоциативного массива используется класс `HashMap`. Данный класс имеет следующие конструкторы:

- конструктор без параметров создает пустой массив;
- конструктор с одним целочисленным параметром создает пустой массив выделяя при этом память под указанное количество элементов;
- конструктор с одним входным параметром в виде `HashMap` заносит в массив элементы из переданного массива.

Пример:

```
Map map = new HashMap();
```

Класс имеет следующие методы:

- `void clear()` – удаление всех элементов из массива;
- `Object clone()` – возвращает копию данного массива;
- `boolean containsKey(Object key)` – возвращает истину, если массив содержит элемент с указанным ключом;
- `boolean containsValue(Object value)` – возвращает истину, если массив содержит один или несколько элементов равных указанному;
- `Set entrySet()` – возвращает коллекцию из элементов, содержащихся в массиве;

- `Object get(Object key)` – возвращает объект с указанным ключом, если объекта с таким ключом нет, то возвращается `null`;
- `boolean isEmpty()` – возвращает истину, если массив пустой;
- `Set keySet()` – возвращает коллекцию, содержащую ключи из данного массива;
- `Object put(Object key, Object value)` – заносит в массив элемент с указанным ключом;
- `void putAll(Map m)` – заносит в массив все элементы из указанного массива, если в массиве уже имеются элементы с совпадающими ключами, они будут заменены новыми;
- `Object remove(Object key)` – удаляет объект с указанным ключом;
- `int size()` – возвращает количество элементов в массиве.

Использование дженериков в коллекциях

Если коллекция должна хранить объекты определенного класса и его потомков, переменную коллекции следует создавать следующим образом:

```
List<Point> points = new ArrayList<Point>();
```

Таким образом, мы получаем переменную коллекции, в которой хранятся точки. И нам компилятор не позволит занести туда что-то кроме `Point` и его потомков. Кроме того, при получении данных с помощью `get` будет получена ссылка правильного типа.

```
Point p = points.get(3);
```

А в первоначальном виде мы должны были бы писать:

```
Point p = (Point)points.get(3);
```

Причем в этом случае есть шансы получить ошибку из-за несоответствия типа хранимого объекта.

Если вы используете коллекцию типа `Map` в угловых скобках, надо указывать два типа – тип ключа и тип значения.

```
Map<String,Point> pointmap = new HashMap<String,Point>();
```

В данном случае мы задаем, что здесь ключом будут строковые объекты, а значениями объекты класса `Point`.

Использование циклов в коллекциях

Допустим у нас есть коллекция:

```
Set<String> stringSet = new HashSet<>() {{
    add("1");
    add("2");
    add("3");
    add("4");
}};
```

Код, чтобы ее перебрать и вывести на экран с помощью улучшенного цикла `for` будет выглядеть так:


```
for (String element: stringSet) {
    System.out.println(element);
}
```

Итераторы

Итератор представляет собой интерфейс, позволяющий перебирать коллекции, элемент за элементом. Так как все коллекции, кроме Map, наследуют и имплементируют метод `iterator()` от интерфейса `Iterable`, то они имеют возможность итерироваться. Метод `iterator()` создаёт и возвращает ссылку на объект, который реализует интерфейс `Iterator` и позволяет итерировать данную коллекцию. Если имеется коллекция `Points`, то создать итератор можно следующим образом:

```
Iterator iterator = points.iterator();
```

Затем можно перебирать все элементы коллекции с помощью методов итератора.

- `boolean hasNext()` – возвращает истину, если у итератора еще остались элементы для перебора;
- `Object next()` – переходит на следующий объект и возвращает пройденный объект;
- `void remove()` – удаляет из коллекции элемент, который только что был пройден.

Пример перебора коллекции с помощью итератора:

```
while(iterator.hasNext()){
    ((Point)iterator.next()).print();
}
```

Как и в самих коллекциях, итератор может использовать угловые скобки с указанием типа.

```
Iterator<Point> iterator = points.iterator(); while(iterator.hasNext()){
    iterator.next().print();
}
```

В данной ситуации мы получили некоторое упрощение записи, так как метод `next` итератора сразу возвращает объекты класса `Point` и нам не потребовалось использовать приведение типов.

Итераторы существуют только для коллекции `Set` или `List`. Коллекции типа `Map` напрямую перебираться итераторами не могут. Можно извлечь набор ключей в виде `Set` и перебирая его итератором перебрать `Map`. Например:

```
Map<String, Integer> m = new HashMap<String, Integer>();
...
Set<String> keys = m.keySet();
Iterator<String> iterator = keys.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    System.out.println(s + " " + m.get(s));
}
```

Следует помнить, что для итераторов не существует понятия “текущий объект”, итератор указывает не на объект, а между объектами.

Данный вариант итератора будет работать практически с любыми коллекциями. Для коллекций типа список, таких как ArrayList или LinkedList существует специальный итератор ListIterator. Для его получения данные коллекции используют метод listIterator. Данный вариант итератора обладает значительно более широкими возможностями, соответственно, большим количеством методов:

- void add(Object o) – вставляет элемент в список;
- boolean hasPrevious() – возвращает истину, если еще есть элементы для перебора, при прохождении в направлении начала списка;
- int nextIndex() – возвращает номер следующего элемента;
- Object previous() – сдвигается на один элемент в направлении начала списка и возвращает пройденный объект;
- int previousIndex() – возвращает номер предыдущего элемента;
- void remove() – удаляет элемент который был пройден последним;
- void set(Object o) – присваивает новое значение элементу, который был пройден последним.

Задания

Задание 40

Создать список оценок учеников с помощью ArrayList, заполнить случайными оценками. Удалить неудовлетворительные оценки из списка.

Для заметок:

Задание 41

Создать коллекцию, заполнить ее случайными целыми числами. Удалить повторяющиеся числа.

Для заметок:

Задание 42

Создать список оценок учеников с помощью ArrayList, заполнить случайными оценками. Найти самую высокую оценку с использованием итератора.

Для заметок:

Задание 43

Ввести текст с клавиатуры. Для текста создать Map, где ключом будет слово, а значение – количество повторений этого слова в тексте.

Для заметок:

Исключения

Основы

Для облегчения работы с ошибками большинство современных языков имеют механизм исключений. Исключительным состоянием называется состояние программы, когда она не может продолжать выполнение данного фрагмента кода и необходимо передать управление фрагменту, который может выполнить необходимые в данном случае действия.

При возникновении исключительной ситуации создается объект специального класса, который также называют исключением. Все эти объекты связаны между собой наследованием. У всех них есть общий предок Throwable, наследниками которого являются два класса: Exception и Error. Все остальные классы являются наследниками этих двух. У исключительных ситуаций, описываемых Error и его наследниками, есть общая особенность – они означают ошибки настолько серьезные, что пользователь ничего не может с ними поделать. Например, если закончилась память, возникает OutOfMemoryError. Продолжение работы программы при возникновении этой и других подобных ошибок невозможно. В случае, если возникает исключение основанное на Exception, пользователь может прописать действия, необходимые в случае его возникновения, и программа сможет продолжить работу. Среди исключений, основанных на Exception, также есть особая группа: класс RuntimeException и его наследники. Такие исключения могут создаваться как программистом явно, так и виртуальной машиной самостоятельно. Типичный пример: NullPointerException. Данное исключение может создаваться самой виртуальной машиной, если пользователь попытался вызвать метод или свойство переменной, в которой хранится null.

Все исключения разделяют на checked(обрабатываемые) и unchecked(необрабатываемые). Обрабатываемые исключения – это исключения, которые Java требует программиста обрабатывать явно. К ним относятся классы Throwable, Exception и их потомки, за исключением класса RuntimeException и его наследников, класса Error и его наследников.

Ключевое слово throw

Для создания исключения используется ключевое слово throw. За ним должен следовать объект исключения. Обычно его создают тут же на месте с помощью new:

```
throw new NullPointerException();
```

Здесь NullPointerException – класс исключения, объект которого создается. Как видно, основой для исключений является объект исключения. Классы исключений являются дочерними от класса Throwable. В простейших случаях можно не создавать свой класс, а использовать непосредственно этот класс, хотя подобная практика не рекомендуется. Обычно все же следует создавать свой класс, унаследовав его от Exception.

```
class CustomException extends Throwable {  
}
```

Обработка исключений

Следует помнить, что вызов исключения используются не только, чтобы их создавать, но и обрабатывать. Для этого используется конструкция следующего вида:

```
try {  
    операторы1;  
} catch (Тип имя) {  
    операторы2;  
} finally {  
    операторы3;  
}
```

В блоке try содержатся операторы1, которые могут вызвать ошибку и создать исключения. В блоке catch содержатся операторы, которые вызываются при возникновении исключения. Блок finally содержит операторы, которые должны выполняться в любых условиях, то есть если исключение было или его не было.

Если в блоке try может возникнуть несколько видов исключений, можно создать несколько идущих подряд блоков catch. Будет вызван тот из них, который соответствует создаваемому исключению. Также можно обойтись одним блоком catch, но в этом случае в нем следует создавать переменную родительского класса исключений. В этом случае ловятся не только исключения именно этого класса, но и исключения его потомков. Таким образом, если в catch в качестве типа указано Exception, будут ловиться все исключения.

Следует помнить, что блок try может не содержать создание исключения, а только вызывать метод, в котором это исключение возникает.

Если у нас несколько видов исключений, но способ их обработки одинаковый, можно писать так:

```
try {  
    операторы1;  
} catch (Тип1 | Тип2 | Тип3 имя) {  
    операторы2;  
}
```

Ключевое слово throws

Если в методе возникает исключение и непосредственно в нем оно не обрабатывается, то есть вышеприведенная конструкция отсутствует, следует указать в заголовке, что этот метод может вызывать исключение, и что его обработкой должен заниматься вызывающий метод. Для этого при объявлении метода указывается throws и имя класса исключения.

```
public static void main(String[] args) throws IOException
```

Задания

Задание 44

Написать код, который выбрасывает NullPointerException. Написать обработчик этого исключения и вывести на экран сообщение, которое будет содержать описание данного исключения.

Для заметок:

Задание 45

Написать собственное исключение от Exception. Сгенерировать код, который будет выбрасывать его и обрабатывать. Результат работы программы вывести на экран.

Для заметок:

Задание 46

Написать метод, который будет возбуждать исключение и обработать это исключение на уровне выше. Результат работы программы вывести на экран.

Для заметок:

Работа с файлами

Чтение текстовых файлов

В языке Java ввод вывод организован с помощью потоков данных (байтов или символов). Для каждого вида потока существует свой класс и/или интерфейс.

Основными классами для чтения информации из файла являются классы `FileInputStream` и `FileReader`. Объекты данных классов выполняют базовые методы файлового ввода вывода, такие как открытие и закрытие файла, чтение информации из файла. Отличие между этими двумя классами заключается в способе чтения информации. Класс `FileInputStream` читает из файла отдельные байты, в то время как `FileReader` читает из файла символы. Так как все символы файла могут быть записаны в виде одного байта или в виде много байтовых символов, то в первом случае разницы между этими классами в чтении информации нет, во втором случае результат работы объектов этих классов будет разным. Для более сложных операций чтения совместно с ним используются объекты других классов.

Файл открывается при создании объекта. Для этого конструктору может быть передано имя файла в виде строки или объект типа `File`, содержащий описание требуемого файла.

```
FileInputStream fileInputStream = new FileInputStream("hello.txt");
```

Если файл не открывается, вызывается исключение `FileNotFoundException`, созданное на основе более общего `IOException`.

Соответственно, если необходимо выполнить действия, если требуемый файл не найден, следует создать обработчик этого исключения.

```
FileInputStream fileInputStream;  
try {  
    fileInputStream = new FileInputStream("hello.txt");  
} catch (FileNotFoundException e) {  
    System.out.print("File not found");  
}
```

Если обработчик исключения в данном методе не создается, следует передать исключение “наверх”, указав в заголовке `throws FileNotFoundException` (либо `IOException`, так как обработчик родительского исключения перехватывает все дочерние).

Когда работа с файлом завершена, следует закрыть файл, используя метод `close()`.

Чтение из потока выполнятся с помощью метода `read()`. Если этот метод вызван без параметров, он возвращает значение типа `int`, содержащее величину очередного прочитанного байта(символа) либо, если файл закончился, возвращается -1.

Пример чтения текстового файла и вывода его на экран. Следует обратить внимание, что в данном примере используется класс `FileReader`, поэтому он будет работать корректно и с русским текстом.

```
import java.io.*;  
public class IOMain {  
    public static void main(String[] args) throws IOException {
```



```

FileReader reader = new FileReader("hello.txt");
int res = reader.read();
while( res != -1) {
    System.out.print((char)res);
    res = reader.read();
}
}
}

```

Объекты классов `FileInputStream` и `FileReader` имеют некоторые недостатки. Во-первых, это исключительно побайтовое (посимвольное) чтение информации, а во-вторых, это достаточно низкое быстродействие, так как для чтения каждого байта выполняется отдельная операция доступа к диску.

Чтобы повысить быстродействие, используется класс `BufferedInputStream` для `FileInputStream` или `BufferedReader` для `FileReader`. Главным их отличием является то, что данные читаются сразу блоками во внутренний буфер и выдаются по мере необходимости. Таким образом, значительно увеличивается быстродействие.

Чтобы не реализовывать повторно файловые операции, уже созданные в `FileInputStream` и `FileReader`, данные классы сделали в виде обертки. То есть в `BufferedInputStream` имеется поле, хранящее ссылку на объект `FileInputStream`, который непосредственно выполняет ввод-вывод (аналогично и для другой пары). Поэтому при создании объекта `BufferedInputStream` его конструктору надо указать объект `FileInputStream`. Обычно создание встроенного объекта выполняется тут же:

```

BufferedInputStream bufferedInputStream = new BufferedInputStream( new
FileInputStream("hello.txt"));

```

аналогично:

```

BufferedReader bufferedReader = new BufferedReader(new
FileReader("hello.txt"));

```

Большинство методов `BufferedInputStream` и `BufferedReader` работают аналогично методам внутренних классов, то есть можно использовать те же методы `read`, `close` и т.д.

Но у класса `BufferedReader` есть метод, который может существенно облегчить работу с текстовыми файлами: метод для чтения строки из файла `readLine()`. Данный метод возвращает объект типа `String`, содержащий очередную прочитанную строку файла. Если достигнут конец файла, возвращается значение `null`. Таким образом, блок чтения файла из предыдущего примера может принять вид:

```

String res = bufferedReader.readLine();
while( res != null){
    System.out.println(res);
    res = bufferedReader.readLine();
}

```

Следует обратить внимание, что при чтении строки символы завершения строки обрезаются, поэтому при выводе используется `println`.

Чтение двоичных файлов

Если файл является двоичным и хранит набор вещественных или целочисленных значений, использование приведенных выше классов для ввода данных очень неудобно. Классы `FileReader` и `BufferedReader` вообще не пригодны для данной задачи, а использование `FileInputStream` потребует дополнительных операций, чтобы собрать значения из последовательности байтов. Поэтому был создан класс `DataInputStream`, умеющий выполнять требуемые действия. Данный класс также является оберткой и может содержать объект типа `FileInputStream` или `BufferedInputStream` (обычно, для повышения быстродействия используют второй). Точно также конструктор данного класса требует объект, который непосредственно будет заниматься вводом-выводом. В результате при использовании `BufferedInputStream`, создание объекта будет иметь вид:

```
DataInputStream in5 = new DataInputStream( new BufferedInputStream( new
FileInputStream("Data.txt")) );
```

Кроме методов, полученных от предыдущих классов, он имеет следующие:

- `boolean readBoolean()` – возвращает прочитанное из файла булевское значение;
- `byte readByte()` – возвращает прочитанный из файла байт;
- `char readChar()` – возвращает символ;
- `double readDouble()` – возвращает вещественное число удвоенной точности;
- `float readFloat()` – возвращает вещественное число;
- `int readInt()` – возвращает целое число;
- `String readLine()` – возвращает текстовую строку (не выполняет корректного преобразования символов, поэтому не рекомендуется к использованию);
- `long readLong()` – возвращает длинное целое;
- `short readShort()` – возвращает короткое целое;
- `int readUnsignedByte()` – возвращает беззнаковое значение байта;
- `int readUnsignedShort()` – возвращает беззнаковое значение короткого целого;
- `int skipBytes(int n)` – пропускает в файле указанное количество байтов.

Следует обратить внимание, что у данных методов отсутствует возможность сообщить о достижении конца файла с помощью возвращаемого значения. Поэтому, если достигнут конец файла, они создают исключение `EOFException`. Это значит, что для корректной обработки файла его следует перехватывать.

Например, вывод на экран файла, содержащего набор целочисленных значений, будет выглядеть:

```
import java.io.*;
public class IOMain {
    public static void main(String[] args) throws IOException {
        DataInputStream dataInputStream = null;
        try{
            dataInputStream = new DataInputStream(new BufferedInputStream( new
FileInputStream("hello.dat")) );
        } catch (FileNotFoundException e) {
```

```
        System.out.print("File not found");
        return;
    }
    int res = dataInputStream.readInt();
    while(true) {
        System.out.println(res);
        try {
            res = dataInputStream.readInt();
        } catch (EOFException e) {
            break;
        }
    }
    if(dataInputStream != null ){
        dataInputStream.close();
    }
}
```

Запись в файл

Запись информации в файл выполняется аналогично чтению. Для нее также имеются специальные классы. Базовыми классами являются `FileOutputStream` и `FileWriter`. Создание объектов этих классов происходит точно также, как и для классов чтения:

```
FileOutputStream f = new FileOutputStream("hello.dat");
```

Если файл не может быть открыт, происходит создание исключения `IOException`. Кроме того, у конструкторов может быть второй параметр булевского типа. Если он равен истине, происходит открытие файла на добавление, то есть содержимое файла не удаляется.

Для записи в файл у данных классов используется метод `write`. Он выполняет побайтовую запись у `FileOutputStream` и посимвольную у `FileWriter`. В качестве параметра может быть передан один байт (символ) либо массив. Если передан массив, можно также указать еще два числовых параметра: с какого и сколько элементов массива следует записать в файл. У класса `FileWriter` передаваемые для записи данные могут быть в виде объекта `String`.

При невозможности записи информации создается исключение `IOException`. Точно также для увеличения быстродействия, используя объекты `BufferedWriter` или `BufferedOutputStream`. Так как эти классы могут не сразу записывать информацию на диск, для того чтобы сделать это принудительно, в них имеется метод `flush()`.

Для записи в файл непосредственно значений переменных используются классы `PrintWriter` и `DataOutputStream`. Первый используется для записи данных в символьном виде, второй – в двоичном.

Как и для чтения при создании объектов данных классов, конструктору следует указать объект, который непосредственно будет заниматься выводом информации.

```
DataOutputStream a = new DataOutputStream(new BufferedOutputStream(new
FileOutputStream("hello.dat")));
```

Для текстового вывода используются методы `print` и `println` класса `PrintWriter`. Отличие этих методов очевидно: второй завершает вывод переводом строки. В качестве параметра этим методам может быть передана величина практически любого примитивного типа, а также строка или массив символов.

Особенность класса `PrintWriter` в том, что методы `println` и `print` не отслеживают многие ошибки, поэтому был введен специальный метод `checkError()`, который занимается проверкой, возникали ли ошибочные состояния и возвращает `true`, если ошибка произошла.

Для двоичного вывода используются методы класса `DataOutputStream`:

- `void writeBoolean(boolean v)` – записывает в файл логическое значение;
- `void writeByte(int v)` – записывает в файл байт;
- `void writeBytes(String s)` – записывает в файл строку как последовательность байтов;
- `void writeChar(int v)` – записывает в файл символ;
- `void writeChars(String s)` – записывает в файл строку как последовательность символов;
- `void writeDouble(double v)` – записывает в файл вещественное число удвоенной точности;
- `void writeFloat(float v)` – записывает вещественное число;
- `void writeInt(int v)` – записывает в файл целое число;
- `void writeLong(long v)` – записывает в файл длинное целое;
- `void writeShort(int v)` – записывает в файл короткое целое.

Пример записи в файл чисел от 0 до 19 в двоичном виде:

```
DataOutputStream a;
try {
    a = new DataOutputStream(new BufferedOutputStream(new
FileOutputStream("hello.dat")));
} catch (FileNotFoundException e) {
    System.out.print("File not found"); return;
}
for(int i=0; i<20;i++) {
    a.writeInt(i);
}
```

Конструкция `try-with-resources`

В процессе работы с файлами может возникнуть ситуация неправильного освобождения ресурсов. Если вы открываете много файлов, но не закрываете их, то они остаются в памяти, что может привести к утечке. В результате программа может перестать работать, когда закончится оперативная память.

Чтобы этого не произошло, используется конструкция `try-with-resources`. Так как потоки с файлами реализуют интерфейс `java.io.Closeable`, то в эта конструкция позволяет автоматически закрывать ресурсы без необходимости ручного вмешательства.

Заккрытие ресурса вручную:

```
import java.io.*;

public class IOMain {

    public static void main(String[] args) throws IOException {
        FileReader reader;

        try{
            reader = new FileReader("hello.txt");
            int res = reader.read();
            while( res != -1) {
                System.out.print((char)res);
                res = reader.read();
            }
        } catch (FileNotFoundException e) {
            System.out.print("File not found");
        } finally {
            reader.close();
        }
    }
}
```

Заккрытие ресурса с помощью конструкции try-with-resources:

```
public class IOMain {

    public static void main(String[] args){
        try(FileReader reader = new FileReader("hello.txt")){
            int res = reader.read();
            while( res != -1) {
                System.out.print((char)res);
                res = reader.read();
            }
        } catch (IOException e) {
            System.out.print("Something wrong with file");
        }
    }
}
```

Класс File

Для некоторых базовых операций с файлами используется класс File. Несмотря на свое название этот класс позволяет работать не только с файлами, но и с каталогами. При создании объекта данного класса ему в соответствие ставится имя файла или каталога. Для этого конструктору передается строка с именем. Имя может быть как абсолютным (то есть с полным указанием пути), так и относительным (то есть относительно текущего каталога). Если необходимо в File занести текущий каталог, его обозначают точкой, например:

```
File path = new File(".");
```

Так как в разных операционных системах используются различные обозначения для разделителей каталогов, в классе имеется специальное свойство `pathSeparator`, хранящее в виде строки знак, разделяющий имена каталогов в пути. Для работы с файлами и каталогами имеются следующие методы:

- `boolean canRead()` – проверяет, может ли файл, на который указывает объект, быть прочитан;
- `boolean canWrite()` – проверяет, может ли выполняться запись в файл, на который указывает объект;
- `int compareTo(File pathname)` – сравнивает путь к данному файлу и путь переданного ему файла как строки;
- `boolean createNewFile()` – создает новый файл, имя которого задано данным объектом. Если файл уже существует, операция не выполняется, и возвращается ложь, если файл создан, возвращается истина;
- `static File createTempFile(String prefix, String suffix)` – создает пустой файл в каталоге для временных файлов. При задании имени используются переданные строки префикса и суффикса. Третьим параметром может быть передан объект `File`, задающий каталог, где надо создать файл;
- `boolean delete()` – удаляет файл или каталог, заданный данным объектом;
- `void deleteOnExit()` – требует от виртуальной машины удаления этого файла, когда работа будет завершена;
- `boolean exists()` – проверяет существует ли файл или каталог, заданный этим объектом;
- `File getAbsolutePath()` – возвращает объект, содержащий абсолютный путь к данному файлу;
- `String getAbsolutePath()` – возвращает абсолютный путь к данному файлу в виде строки;
- `File getCanonicalFile()` – предыдущие методы, хоть и возвращают абсолютный путь к файлу, но могут содержать ненужные знаки, как например, `.` или `..`. Поэтому данный метод создает объект, хранящий путь в каноническом виде;
- `String getCanonicalPath()` – возвращает путь к файлу в каноническом виде как строку.

Пример:

```
File f1 = new File("./hello.txt");  
System.out.println(f1.getAbsolutePath());  
System.out.println(f1.getCanonicalPath());
```

Результатом работы этого фрагмента программы могут быть строки (пример для Linux):

```
/home/academy/install/eclipse/workspace/ioproj/./hello.txt  
/home/academy/install/eclipse/workspace/ioproj/hello.txt
```

- `String getName()` – возвращает имя файла или каталога, содержащегося в этом объекте;
- `String getParent()` – возвращает путь к родительскому каталогу;
- `File getParentFile()` – возвращает объект, содержащий родительский каталог;
- `String getPath()` – возвращает путь к файлу в виде строки;

- `boolean isAbsolute()` – проверяет, является ли путь, задающий данный файл, абсолютным;
- `boolean isDirectory()` – проверяет, указывает ли данный объект на каталог;
- `boolean isFile()` – проверяет, указывает ли данный объект на файл;
- `boolean isHidden()` – проверяет, является ли данный файл или каталог скрытым;
- `long lastModified()` – возвращает время, когда данный файл был изменен последний раз;
- `long length()` – возвращает длину файла;
- `String[] list()` – если объект указывает на каталог, возвращает массив строк с именами файлов и каталогов, находящихся в нем;
- `File[] listFiles()` – если объект указывает на каталог, возвращает массив объектов, связанных с файлами и каталогами, находящимися в нем;
- `static File[] listRoots()` – возвращает допустимые корни файловой системы;
- `boolean mkdir()` – создает каталог, на который указывает данный объект;
- `boolean mkdirs()` – аналогично предыдущему, но также создает, если необходимо родительские каталоги;
- `boolean renameTo(File dest)` – переименовывает файл;
- `boolean setLastModified(long time)` – устанавливает время, когда файл последний раз изменялся;
- `boolean setReadOnly()` – устанавливает для файла атрибут только для чтения.

Задания

Задание 47

Вывести список файлов и каталогов выбранного каталога на диске. Файлы и каталоги должны быть распечатаны отдельно.

Для заметок:

Задание 48

Создать файл с текстом, прочитать, подсчитать в тексте количество знаков препинания и слов. Вывести результат на экран.

Для заметок:

Задание 49

Создать файл с текстом, в котором присутствуют числа. Найти все числа, вывести на экран, посчитать сумму, убрать все повторяющиеся числа и снова вывести на экран.

Для заметок:

Задание 50

Записать с помощью Java в двоичный файл 20 случайных чисел. Прочитать записанный файл, вывести на экран числа и их среднее арифметическое.

Для заметок:

Задание 51

Создать цепочку из трех папок. В конечном каталоге создать 5 произвольных текстовых файлов. Заполнить их 10 произвольными целыми числами. Содержимое файлов записать в один файл в том же каталоге. Создать файл, который будет содержать список файлов данного каталога.

Для заметок:

Задание 52

Создать объект Person с полями name, surname, age. Сгенерировать 10 объектов класса Person со случайными полями соответствующего типа. С помощью Java создать файл, в который запишется информация о этих людях.

Для заметок:

Класс Thread и интерфейс Runnable

Создание потоков

Язык программирования Java имеет встроенные средства для создания многопоточных приложений. Для создания потока следует создать класс потока. Это можно сделать двумя способами. В простейшем случае класс потока создается на основе стандартного класса Thread. В данном классе имеется метод run, который содержит все операции, выполняемые данным потоком.

```
class NewThread extends Thread {  
    public void run() {  
        int counter=0;  
        while (counter < 100) {  
            counter++;  
            System.out.println(counter);  
        }  
    }  
}
```

В данном примере имеется класс на основе Thread, в нем в методе run выполняется увеличение счетчика на 1 и распечатка этого значения.

Для запуска потока на выполнение используется метод start класса Thread.

```
NewThread t1 = new NewThread();  
t1.start();
```

У программ, использующих потоки, имеется несколько особенностей. Так как при вызове метода main тоже создается поток с именем main, то обычная программа будет работать, пока не завершится выполнение метода main. В то время как программа, работающая с потоками, завершится, когда завершится выполнение всех потоков.

При выполнении потоков каждому из них система будет выделять некоторый интервал времени для выполнения, а затем переключаться на другой поток. И таким образом по кругу будут перебираться имеющиеся в системе потоки. Время, выделяемое каждому из потоков, зависит от его приоритета.

Если создать несколько объектов класса NewThread, очень вероятно они отработают последовательно, так как на современных компьютерах интервал времени, выделяемый потоку, достаточен для перебора чисел от 1 до 100 и вывода их. Если есть необходимость в более частом переключении между потоками, можно воспользоваться методом yield. Тогда метод run примет вид:

```
public void run() {  
    int counter=0;  
    while (counter < 100){  
        counter++;  
        System.out.println(counter);  
        yield();  
    }  
}
```

Метод `yield` сигнализирует системе, что можно начать выполнять следующий поток.

Достаточно часто встречаются ситуации, когда выполнение потока следует приостановить на определенный промежуток времени. Для этого используется метод `sleep`, который обеспечивает приостановку потока на указанное количество миллисекунд.

```
sleep(1000);
```

Таким образом, выполнение текущего потока будет приостановлено на 1 секунду. Следует учитывать, что данный метод является статическим и не существует разницы от имени какого объекта потока он вызван. Лучше всего вызывать данный метод от имени класса:

```
Thread.sleep(1000);
```

Он всегда приостанавливает только тот поток, который выполняется в данный момент.

Метод `sleep` может создавать `InterruptedException` – исключение срабатывающее, если кто-то прерывает ожидание процесса. Поэтому при использовании данного метода необходимо либо ловить данное исключение, либо добавлять `throws` в метод.

Поток можно приостановить до того момента, пока другой поток не закончит свою работу. Для этого от объекта потока, завершения которого следует дожидаться, надо вызвать метод `join`.

Например:

```
MyThread t1 = new MyThread();
MyThread t2 = new MyThread();
t1.start();
t2.start();
t1.join();
t2.join();
System.out.println("Treads competed!!!!");
```

В данном примере в основном потоке создаются и запускаются два дополнительных. После их запуска вызывается `join`, основной поток останавливается и ждет, пока оба дополнительных не завершат свою работу. Таким образом, распечатка в последней строке примера будет выполнена только после того, как оба потока полностью отработают.

Каждый поток имеет свое имя. Оно может быть дано автоматически или программистом. Что бы установить имя потока, можно при его создании вызывать конструктор класса `Thread`, указав в качестве параметра строку с именем.

```
public NewThread() {
    super("mythread");
}
```

В данном примере создается поток с именем `mythread`. Имя потока также можно установить с помощью метода `setName`.

```
setName("mythread");
```

Чтобы узнать имя текущего потока, можно воспользоваться методом `getName`, который возвращает имя потока в виде объекта `String`.

Так, чтобы в нашем примере распечатывалось не только значение счетчика, но и имя потока, можно написать:

```
System.out.println(getName()+" "+ Counter);
```

Имеется два метода, позволяющие установить приоритет потока, или также получить текущее значение приоритета. Метод `setPriority` получает в качестве входного параметра целое значение. Чем больше значение и, соответственно, выше приоритет, тем большая доля времени выделяется данному потоку на выполнение. Значение приоритета должно находиться в промежутке, задаваемом константами `MAX_PRIORITY` `MIN_PRIORITY`.

У создания объекта потока путем наследования от `Thread` есть один большой недостаток: его невозможно использовать, если класс должен быть дочерним от другого класса, так как множественное наследование классов в Java запрещено.

В этом случае следует использовать второй способ создания класса потока: его следует создавать на основе интерфейса `Runnable` (это интерфейс, на основе которого сделан сам `Thread`).

Так как самостоятельно создавать методы `start`, `yield` и тому подобное не представляется возможным, все равно приходится использовать класс `Thread`, но в данной ситуации он является не родительским классом, а используется для создания объекта, свойства создаваемого класса потока.

```
class NewThread implements Runnable {
    Thread th;
    public NewThread() {
        th = new Thread(this);
    }
    public void run() {
        int count = 0;
        while(count<100) {
            count++;
            System.out.println(count);
        }
    }
    void start() {
        th.start();
    }
}

class ThreadsTest {
    public static void main(String[] args) {
        NewThread t1 = new NewThread();
        NewThread t2 = new NewThread();
        t1.start();
        t2.start();
    }
}
```

Следует обратить внимание, что в конструкторе создаваемого класса нужно создать объект класса Thread и передать ему ссылку на текущий объект. В этом случае методы, вызываемые от объекта Thread, будут применяться к нашему объекту. Так чтобы узнать имя потока, следует написать:

```
t1.th.getName();
```

Следует помнить, что основная часть программы также представляет собой поток и можно получить нить этого потока. Для этого можно использовать метод `currentThread` класса Thread. Данный метод является статическим и поэтому его следует вызывать от имени класса:

```
Thread currentThread = Thread.currentThread();
```

После выполнения данной операции ссылка `currentThread` будет содержать объект текущего потока. Соответственно, если эта операция будет выполнена, скажем в `main`, это будет объект основного потока программы.

Задания

Задание 53

Создать 10 потоков, каждый из которых вычисляет среднее арифметическое коллекции из 10 случайных целых чисел и выводит на экран.

Для заметок:

Задание 54

Создать класс поток, который генерирует массив случайных целых чисел из 10 элементов, затем находит максимальный элемент, в этом массиве, и выводит на экран в формате имя потока, максимальный элемент. Запустить 10 потоков.

Для заметок:

Задание 55

Создать класс поток, который создает средствами Java файл и записать в него произвольно сгенерированный массив из 10 случайных целых чисел. Запустить 5 потоков.

Для заметок:

Взаимодействие потоков. Producer – Consumer

Синхронизация

Достаточно часто доступ нескольких потоков к одному методу или к одним данным объекта одновременно является нежелательным. Например, какой-либо метод выполняет вывод блока информации на экран. При этом если его вызовут одновременно (или почти одновременно) несколько потоков, выводимая информация может перемешаться. Чтобы этого не происходило, используется синхронизация.

Существуют два основных типа синхронизации: синхронизация по методу, синхронизация по объекту (синхронизация блоков).

Для того чтобы потоки не могли вызвать метод одного и того же объекта одновременно, следует в заголовке метода указать ключевое слово `synchronized`. В этом случае один из потоков будет ждать, пока другой не завершит работу с методом данного объекта.

При синхронизации блоков перед блоком, взятым в фигурные скобки, ставится ключевое слово `synchronized`, а после него в круглых скобках указывается объектная переменная.

```
synchronized(obj){  
    ...  
}
```

Когда поток подходит к синхронизированному блоку, происходит проверка, выполняет ли кто-нибудь другой синхронизированный блок с таким же объектом в заголовке. Если да, то пришедший поток ждет, пока ранее занявший не завершит работу с блоком.

В качестве объекта синхронизации может использоваться объект любого класса. Иногда для этого создают объекты класса `Object`, не выполняющие никаких других задач кроме синхронизации.

Например, у нас имеется класс потока, выполняющий суммирование чисел от 1 до указанного значения, и заносящий результаты работы в `StringBuffer`:

```
public class CountClass extends Thread {  
    private StringBuffer text;  
    private int countTo;  
    public CountClass(StringBuffer s, int c) {  
        text = s;  
        countTo = c;  
    }  
    @Override  
    public void run() {  
        synchronized (text) {  
            int sum = 0;  
            for(int i=1;i<=countTo;i++){  
                sum += i;  
                text.append("Next value="+i);  
            }  
        }  
    }  
}
```

```

    }
    text.append("\nsum="+sum+"\n");
}
}
}

```

Далее мы создаем два подобных потока и выводим результат их работы:

```

StringBuffer text = new StringBuffer();
CountClass c1 = new CountClass(text, 200);
CountClass c2 = new CountClass(text, 100);
c1.start();
c2.start();
c1.join();
c2.join();
System.out.println("Result:"+text);

```

Эти потоки используют одну и ту же переменную для занесения результата. Если не использовать синхронизацию, то результаты работы будут перемешаны. Синхронизация по объекту позволяет разделить вывод данных потоков. Следует также обратить внимание, что именно эта переменная используется как объект синхронизации.

Взаимодействие потоков

Возможны и другие способы взаимодействия потоков. В частности, встречаются ситуации, когда один поток должен выполнять некоторые операции только после того, как другой поток дойдет до определенной точки. В этом случае первый поток можно приостановить, вызвав метод `wait`. Данный метод приостанавливает выполнение данного потока, пока другой поток не вызовет метод `notify`, как сигнал того, что можно продолжать выполнение дальше.

Данный способ, как и синхронизация использует объект. Именно от объекта (любого класса) должен выполняться метод `wait`, от этого же объекта должен вызываться метод `notify`, чтобы приостановленный поток продолжил свою работу.

Еще есть требование – вызов метода должен находиться в синхронизованном блоке. Также должно ловиться исключение `InterruptedException`, таким образом вызов может выглядеть так:

```

synchronized (obj) {
    try {
        obj.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

Метод `notify` также должен вызываться в синхронизованном блоке. И чтобы приостановленный поток продолжил свое выполнение, другой поток должен выполнить код вида:

```
synchronized (obj) {
    obj.notify();
}
```

В данных примерах obj должен быть ссылкой на один и тот же объект, доступный обоим потокам.

Метод wait может не иметь параметров или иметь параметр в виде целого числа. В этом случае, он может продолжить свое выполнение не только после сообщения notify, но и по истечении промежутка времени, заданного данным числом.

Если существует несколько ожидающих потоков, только один из них продолжит свое выполнение при вызове notify. Если необходимо продолжить выполнение всех потоков, следует вызывать метод notifyAll.

Вместо синхронизации и методов wait notify, есть возможность использования специальных объектов блокировки. Классы этих объектов наследуются от интерфейса Lock и самым часто используемым является ReentrantLock. Чтобы использовать такую блокировку, оба потока должны иметь доступ к одному и тому же объекту. Когда поток дошел до синхронизируемого фрагмента кода, он вызывает метод lock(). Если никто в данный момент не занял эту блокировку, выполнение продолжается, если другой поток ранее вызвал lock() у этого же объекта, вновь пришедший будет ждать освобождения. Блокировка освобождается с помощью метода unlock().

Таким образом, поток, использующий блокировку для синхронизации, может выглядеть так:

```
public class NewCountClass extends Thread {
    private StringBuffer text;
    private int countTo;
    private Lock lock;
    public NewCountClass(StringBuffer s, int c, Lock lock) {
        text = s;
        countTo = c;
        this.lock = lock;
    }
    @Override
    public void run() {
        lock.lock();
        int sum = 0;
        for(int i=1;i<=countTo;i++){
            sum += i;
            text.append("Next value="+i);
        }
        text.append("\nsum="+sum+"\n");
        lock.unlock();
    }
}
```


Важной особенностью использования объектов блокировки является то, что можно не останавливать поток, дожидаясь, пока другие закончат, а выполнять какие-нибудь другие действия. В этом случае для получения блокировки следует воспользоваться методом `tryLock()`, который возвращает булевское значение. Значение `true`, если блокировка получена, и `false`, если нет. Получение блокировки в этой ситуации может выглядеть как-то так:

```
while(!lock.tryLock()){  
    Действия не требующие блокировки  
}
```

Deadlock

Deadlock – это взаимная блокировка двух процессов, которые работают одновременно с двумя синхронизированными ресурсами. Его сложно отладить, потому что возникает очень редко, когда исполнение двух потоков совпадает точно по времени, а также когда может участвовать более двух процессов одновременно.

Простейший deadlock:

```
class A {  
    synchronized void foo(B b) {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + "    A.foo()");  
        try {  
            Thread.sleep(1000);  
        } catch (Exception e) {  
            System.out.println("Exception at A ");  
        }  
        System.out.println(name + "    SecondClass.last()");  
        b.last();  
    }  
  
    synchronized void last() {  
        System.out.println("    A.last()");  
    }  
}  
  
class B {  
    synchronized void bar(A a) {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + "    B.bar()");  
        try {  
            Thread.sleep(1000);  
        }  
    }  
}
```

```

        } catch (Exception e) {
            System.out.println(" B ");
        }
        System.out.println(name + "    A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("  B.last()");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    Deadlock() {
        Thread.currentThread().setName(" ");
        Thread t = new Thread(this, " ");
        t.start();
        a.foo(b);
    }

    public void run() {
        b.bar(a);
    }

    public static void main(String args[]) {
        new Deadlock();
    }
}

```

Livelock

Livelock – другая проблема параллелизма, похожая на deadlock. В режиме livelock два или более потоков продолжают передавать состояния друг другу, а не ждать бесконечно. Следовательно, потоки не могут выполнять свои задачи.

Отличным примером livelock является система обмена сообщениями, в которой, когда возникает исключение, потребитель сообщения откатывает транзакцию и помещает сообщение обратно в начало очереди. Затем одно и то же сообщение многократно читается из очереди только для того, чтобы вызвать другое

исключение и быть помещенным обратно в очередь. Потребитель никогда не получит никакого другого сообщения из очереди.

Starvation

Есть несколько потоков, и все должны получить блокировку для определенного объекта. Первый поток удерживает блокировку на общем объекте в течение длительного времени. Все это время другие потоки ожидают, но длительный поток может вызываться очень часто. Таким образом, другим потокам будет заблокирован доступ к общему объекту. Эта ситуация называется starvation в многопоточности.

Задания

Задание 56

Создать метод, который печатает название потока и засыпает на 2 секунды. Запустить одновременно 10 потоков. Реализовать механизм синхронизации, чтобы все потоки выполнились последовательно.

Для заметок:

Задание 57

Создать программу, которая реализует deadlock между тремя потоками.

Для заметок:

Задание 58

Есть 3 производителя и 2 потребителя, все разные потоки и работают все одновременно. Есть очередь с 200 элементами. Производители добавляют случайное число от 1..100, а потребители берут эти числа. Если в очереди элементов ≥ 100 производители спят, если нет элементов в очереди – потребители спят. Если элементов стало ≤ 80 , производители просыпаются. Все это работает до тех пор, пока обработанных элементов не станет 10000, только потом программа завершается.

Для заметок:

Пулы потоков. Асинхронные вычисления

Пулы потоков

Наиболее часто используемым вариантом являются пулы потоков – объекты, которые занимаются запуском потоков и прочими операциями с ними. Они реализуют интерфейс `ExecutorService`, и создаются с помощью методов-фабрик класса `Executors`. Основные три фабрики:

- `static ExecutorService newCachedThreadPool()` – объект, позволяющий создавать любое количество работающих параллельно потоков;
- `static ExecutorService newFixedThreadPool(int nThreads)` – объект, позволяющий создавать определенное количество параллельно работающих потоков, остальные добавляемые потоки становятся в очередь;
- `static ExecutorService newSingleThreadExecutor()` – объект, позволяющий создавать один поток.

Например, пул, который может выполнять одновременно до 5 потоков, создается следующим образом:

```
ExecutorService pool = Executors.newFixedThreadPool(5);
```

Сами потоки создаются с помощью метода `execute`. При этом поток должен быть наследником интерфейса `Runnable`.

В качестве примера рассмотрим создание потока в пуле с помощью анонимного класса:

```
pool.execute(new Runnable() {  
    @Override  
    public void run() {  
        int i=0;  
        while(i<50){  
            System.out.print("num:"+i);  
            i++;  
        }  
    }  
});
```

Одним из важных достоинств пулов потоков является то, что при создании потока, если ранее такой поток уже выполнялся, используется уже готовый, а не создается новый, в то время как при запуске с помощью `Thread` каждый раз создается новый поток. Это обеспечивает существенную экономию ресурсов.

С помощью пулов можно также завершить выполнение ранее запущенных потоков. Для этого используется методы `shutdown` и `shutdownNow`. Первый позволяет уже запущенным потокам завершиться в нормальном режиме, но не позволяет создавать новые. Второй принудительно прекращает работу потоков. Следует учитывать, что данные методы не ожидают, пока запущенные потоки остановятся. Для этого следует использовать метод `awaitTermination`.

Интерфейс Callable

Интерфейс Callable<V> очень похож на интерфейс Runnable. Объекты, реализующие данные интерфейсы, исполняются другим потоком. Однако, в отличие от Runnable интерфейс Callable использует Generic', для определения типа возвращаемого объекта. Runnable содержит метод run(), описывающий действие потока во время выполнения, а Callable – метод call().

Интерфейс Future

Объект Future содержит в себе результат асинхронного вычисления. При запуске потока в пуле потока возвращается объект Future. В любой момент работы программы из объекта Future можно получить результат, когда он будет готов.

Основные методы:

- cancel (boolean mayInterruptIfRunning) – попытка завершения задачи;
- V get() – ожидание (при необходимости) завершения задачи, после чего можно будет получить результат;
- V get(long timeout, TimeUnit unit) – ожидание (при необходимости) завершения задачи в течение определенного времени, после чего можно будет получить результат;
- isCancelled() – вернет true, если выполнение задачи будет прервано прежде завершения;
- isDone() – вернет true, если задача завершена.

Задания

Задание 59

Создать задачу Callable, которая генерирует 10 файлов с 10 произвольными строками -> засыпает произвольно на 1-3 секунды, результат выполнения – коллекция имен файлов. Запустить 10 задач параллельно в пуле из 3 потоков. Вывести ход программы на экран с указанием имени потока, который выполняет работу.

Для заметок:

Задание 60

Создать задачу Callable, которая генерирует коллекцию из 10 рандомных целых чисел -> засыпает произвольно на 1-10 секунд, результат выполнения – сумму этих чисел в виде строки. Запустить 10 задач параллельно в пуле из 3 потоков. Вывести ход программы на экран с указанием имени потока, который выполняет работу.

Для заметок:

Задание 61

Создать задачу Callable, которая берет сообщение “Hello World” + текущее время и записывает его в файл. Запись в файл должна производиться последовательно через синхронизированный метод. Запустить 10 задач параллельно в пуле из 3 потоков. Вывести ход программы на экран с указанием имени потока, который выполняет работу.

Для заметок:

Lambda, Streams API

Основные понятия

Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную, а затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая, собственно, представляет тело лямбда-выражения, где выполняются все действия. Общий вид lambda выражения:

(спецификация параметров) -> {операторы}

Рассмотрим создание объекта потока:

```
Runnable r = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello World!");  
    }  
};
```

Чтобы описать тоже самое только с использованием лямбда-выражения, можно использовать такую конструкцию:

```
Runnable r = () -> System.out.println("Hello World!");
```

Функциональный интерфейс

Функциональные интерфейсы – это интерфейсы, которые содержат в себе только один абстрактный метод. Функциональные интерфейсы имеют тесную связь с

лямбда-выражениями и служат основой для применения лямбда-выражений в функциональном программировании на Java.

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

Имейте в виду, что этот код останется корректным даже если убрать аннотацию `@FunctionalInterface`.

Ссылки на методы

Java позволяет передавать ссылки на метод. Рассмотрим код:

```
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);
```

Чтобы преобразовать его с использованием ссылки на метод, можно использовать ключевое слово `::`, которое используется следующим образом:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);
```

Ссылки на конструкторы

Как и со ссылками на методы, можно использовать ссылку на конструктор. Рассмотрим класс с конструктором:

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

Создадим интерфейс, который создает объект `Person` по имени и фамилии:

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

В результате использования ссылки на конструктор процесс создания объекта с помощью этого конструктора выглядит так:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```


При использовании ссылок на конструкторы, методы функциональных интерфейсов должны принимать тот же список параметров, что и конструкторы класса, а также должны возвращать объект данного класса.

Optional<T>

Контейнерный объект, который может содержать или не содержать ненулевое значение. Если значение присутствует, метод `isPresent()` вернет `true`, а `get()` вернет значение.

У данного объекта предоставляются дополнительные методы, которые зависят от наличия или отсутствия содержащегося в нем значения. Например, `orElse()` (возвращает значение по умолчанию, если значение отсутствует) и `ifPresent()` (выполняет блок кода, если значение присутствует).

Stream API

Stream API – это способ работать со структурами данных в функциональном стиле. Чаще всего с помощью stream работают с коллекциями, но на самом деле этот механизм может использоваться для самых различных данных.

Существует несколько способов создания стримов из коллекции:

- создание стрима из коллекции

```
Collection<String> collection = Arrays.asList("a1", "a2", "a3");  
Stream<String> streamFromCollection = collection.stream();
```

- создание стрима из значений `Stream.of(значение1,... значениеN)`

```
Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");
```

- диапазон значений

```
IntStream streamFromValues = IntStream.range(1, 10);
```

Java Stream API предлагает два основных метода:

- конвейерные – возвращают другой stream, то есть работают как builder;
- терминальные – возвращают другой объект, такой как коллекция, примитивы, объекты, Optional и т.д.

Терминальные методы

- `findFirst` – возвращает первый элемент из стрима (возвращает Optional)

```
collection.stream().findFirst().orElse(<1>)
```

- `findAny` – возвращает любой подходящий элемент из стрима (возвращает Optional)

```
collection.stream().findAny().orElse(<1>)
```

- `collect` – представление результатов в виде коллекций и других структур данных

```
collection.stream().filter((s) ->  
s.contains(<1>)).collect(Collectors.toList())
```

- `count` – возвращает количество элементов в стриме

```
collection.stream().filter(<a1>::equals).count()
```

- **anyMatch** – возвращает true, если условие выполняется хотя бы для одного элемента

```
collection.stream().anyMatch («a1»::equals)
```

- **noneMatch** – возвращает true, если условие не выполняется ни для одного элемента

```
collection.stream().noneMatch («a8»::equals)
```

- **allMatch** – возвращает true, если условие выполняется для всех элементов

```
collection.stream().allMatch((s) -> s.contains («1»))
```

- **min** – возвращает минимальный элемент, в качестве условия использует компаратор

```
collection.stream().min(String::compareTo).get()
```

- **max** – возвращает максимальный элемент, в качестве условия использует компаратор

```
collection.stream().max(String::compareTo).get()
```

- **forEach** – применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется

```
set.stream().forEach((p) -> p.append("_1"));
```

- **forEachOrdered** – применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантируется

```
list.stream().forEachOrdered((p) -> p.append("_new"));
```

- **toArray** – возвращает массив значений стрима

```
collection.stream().map(String::toUpperCase).toArray(String[]::new);
```

- **reduce** – позволяет выполнять агрегатные функции на всей коллекции и возвращать один результат

```
collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)
```

Конвейерные методы

- **filter** – отфильтровывает записи, возвращает только записи, соответствующие условию

```
collection.stream().filter («a1»::equals).count()
```

- **skip** – позволяет пропустить N первых элементов

```
collection.stream().skip(collection.size() - 1).findFirst().orElse («1»)
```

- **distinct** – возвращает стрим без дубликатов (для метода equals)

```
collection.stream().distinct().collect(Collectors.toList())
```

- **map** – преобразует каждый элемент стрима

```
collection.stream().map((s) -> s + "_1").collect(Collectors.toList())
```

- **peek** – возвращает тот же стрим, но применяет функцию к каждому элементу стрима

```
collection.stream().map(String::toUpperCase).peek((e) ->
System.out.print(", " + e)).
collect(Collectors.toList())
```

- **limit** – позволяет ограничить выборку определенным количеством первых элементов

```
collection.stream().limit(2).collect(Collectors.toList())
```

- **sorted** – позволяет сортировать значения либо в натуральном порядке, либо задавая **Comparator**

```
collection.stream().sorted().collect(Collectors.toList())
collection.stream().sorted(Comparator.comparing(Student::getAge)).collect(
Collectors.toList())
```

Задания

Задание 62

Напишите программу, которая для всех четных значений длиной от 1 до 20 дюймов:

- печатает на экран значения в дюймах;
- переводит значения в сантиметры;
- печатает на экран сумму в сантиметрах.

Для заметок:

Задание 63

Создайте класс **Person** с полями **name**, **surname**, **age**. Сгенерируйте группу из 100 человек в возрасте от 15 до 30. Напишите ОДНУ НЕПРЕРЫВНУЮ цепочку **stream** вызовов, которая:

- 1) выбирает объекты, возраст которых меньше 21;
- 2) распечатывает их на экран;
- 3) сортирует по фамилии, а потом по имени (использовать **thenComparing** у объекта **Comparator**);
- 4) берет 4 первых объекта;
- 5) формирует коллекцию из фамилий объектов;
- 6) агрегирует все в коллекцию.

Для заметок:

Задание 64

Сгенерируйте List коллекцию целых чисел из n элементов от minValue до maxValue. Определить, содержатся ли в данной коллекции числа, которые делятся и на 3, и на 5 с помощью stream.

Для заметок:

Date Time API

Основные пакеты

Для работы со временем в Java реализованы следующие библиотеки:

- пакет `java.time` – это базовый пакет нового Date Time API. Все основные базовые классы являются частью этого пакета: `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration` и другие. Все эти классы являются неизменными и потокобезопасными. В большинстве случаев этих классов будет достаточно для большинства задач;
- пакет `java.time.chrono` – пакет с общими интерфейсами для не календарных систем ISO. Мы можем наследовать класс `AbstractChronology` для создания собственной календарной системы;
- пакет `java.time.format` – пакет с классами форматирования и парсинга времени и даты. В большинстве случаев мы не будем использовать их напрямую, потому что классы в пакете `java.time` предоставляют удобные методы для форматирования и парсинга;
- пакет `java.time.temporal` используется для работы с временными объектами, например, с помощью него мы можем узнать первый или последний день месяца. Методы таких классов сразу заметны на фоне других, потому что всегда имеют формат `'withXXX'`;
- пакет `java.time.zone` – классы для поддержки различных часовых поясов и правила их изменения.

Каждый из данных пакетов содержит классы для работы с датой и временем, а также с периодами времени.

Класс `java.time.LocalDate`

`java.time.LocalDate` – неизменяемый класс, который представляет объекты Date в формате по умолчанию yyyy-MM-dd. Мы можем использовать метод `now()`, чтобы получить текущую дату. Мы также можем предоставить в качестве аргументов год, месяц и день, чтобы создать экземпляр `LocalDate`. Этот класс предоставляет перегруженный метод `now()`, которому мы можем предоставить `ZoneId` для получения даты в конкретном часовом поясе. Этот класс предоставляет такую же функциональность, как `java.sql.Date`.

Пример:

```
// Получаем текущую дату
LocalDate today = LocalDate.now();
System.out.println("Текущая дата : " + today);
//Создадим LocalDate и в качестве аргументов
//укажем год месяц и день
LocalDate specificDate = LocalDate.of(2017, Month.NOVEMBER, 30);
System.out.println("Дата с указанием года, месяца и дня : " +
specificDate);
```

Класс java.time.LocalTime

Класс **LocalTime** является неизменяемым и представляет собой время в читабельном виде. По умолчанию он предоставляет формат hh:mm:ss.zzz. Так же, как и **LocalDate**, этот класс обеспечивает поддержку часовых поясов и создание даты, передавая в качестве аргументов часы, минуты и секунды.

Пример:

```
// получаем текущее время
LocalTime time = LocalTime.now();
System.out.println("Получаем текущее время : " + time);
//Создаем LocalTime с использованием своих данных в качестве параметров
LocalTime specificTime = LocalTime.of(23, 15, 11, 22);
System.out.println("Какое-то время дня : " + specificTime);
```

Вспомогательные методы Date API

Пример:

```
LocalDate today = LocalDate.now();
//Получаем год, проверяем его на високосность
System.out.println("Год " + today.getYear() + " - високосный? : " +
today.isLeapYear());
//Сравниваем два LocalDate: до и после
System.out.println("Сегодня - это до 02.03.2017? : " +
today.isBefore(LocalDate.of(2017,3,2)));
//Создаем LocalDateTime с LocalDate
System.out.println("Текущее время : " + today.atTime(LocalTime.now()));
//Операции + и - с датами
System.out.println("9 дней после сегодняшнего дня будет: " +
today.plusDays(9));
System.out.println("3 недели после сегодняшнего дня будет: " +
today.plusWeeks(3));
System.out.println("20 месяцев после сегодняшнего дня будет: " +
today.plusMonths(20));
```

```

System.out.println("9 дней до сегодняшнего дня будет: " +
today.minusDays(9));

System.out.println("3 недели до сегодняшнего дня будет: " +
today.minusWeeks(3));

System.out.println("20 месяцев до сегодняшнего дня будет: " +
today.minusMonths(20));

// А теперь поиграемся с датой

System.out.println("Первый день этого месяца : " +
today.with(TemporalAdjusters.firstDayOfMonth()));

LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());

System.out.println("Последний день этой года : " + lastDayOfYear);

Period period = today.until(lastDayOfYear);

System.out.println("Находим время между двумя датами : « + period);

System.out.println("В этом году осталось " + period.getMonths() + "
месяц(ев)");

```

Парсинг и форматирование даты

Пример:

```

LocalDate date = LocalDate.now();
// стандартный формат даты
System.out.println("стандартный формат даты для LocalDate : " + date);
// применяем свой формат даты
System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")
));
System.out.println(date.format(DateTimeFormatter.BASIC_ISO_DATE));
LocalDateTime dateTime = LocalDateTime.now();
//стандартный формат даты
System.out.println("стандартный формат даты LocalDateTime : " + dateTime);
//применяем свой формат даты
System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MMM::uuu
u HH::mm::ss")));
System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));
Instant timestamp = Instant.now();
//стандартный формат даты
System.out.println("стандартный формат : " + timestamp);

```

Задания

Задание 65

Ввести с клавиатуры номер месяца текущего года. Вывести на экран все его даты в формате d::MMM::uuuu.

Для заметок:

Задание 66

От текущей даты вывести расписание всех встреч, которые происходят еженедельно в 13:00 в течение 2 месяцев.

Для заметок:

Задание 67

От текущей даты вывести на экран дату, которая была 60 дней назад.

Для заметок:

Принципы дизайна ПО. SOLID. Паттерны

Понятие шаблона проектирования

Шаблон проектирования (design pattern) – образец решения задачи, который можно использовать в различных ситуациях. При проектировании программы некоторые задачи встречаются достаточно регулярно. Шаблоны проектирования описывают то, как могут решаться такие часто встречаемые задачи. Основой для современного понимания шаблонов проектирования стала книга Design patterns – Elements of reusable Object-oriented software. В данной книге были описаны 23 шаблона проектирования. Кроме базовых шаблонов проектирования, существуют шаблоны относящиеся к конкретным языкам программирования и технологиям.

SOLID (сокр. от англ. single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion) в программировании – акроним, введенный Майклом Фэзерсом (Michael Feathers) для первых пяти принципов, названных Робертом Мартином в начале 2000-х, которые означали 5 основных принципов объектно-ориентированного программирования и проектирования. Принципы объектно-ориентированного программирования:

- **принцип единственной ответственности** (The Single Responsibility Principle) – каждый класс должен иметь одну и только одну причину для изменений;

- **принцип открытости/закрытости** (The Open Closed Principle) – «программные сущности ... должны быть открыты для расширения, но закрыты для модификации»;
- **принцип подстановки Барбары Лисков** (The Liskov Substitution Principle) – «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Наследующий класс должен дополнять, а не изменять базовый;
- **принцип разделения интерфейса** (The Interface Segregation Principle) – «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»;
- **принцип инверсии зависимостей** (The Dependency Inversion Principle) – «Зависимость на Абстракциях. Нет зависимости на что-то конкретное».

Singleton

Singleton или Одиночка – объект, который может существовать в программе в единственном экземпляре. Такие объекты встречаются достаточно часто. Например, объект, содержащий настройки приложения, объект диалогового окна и т.д.

Если создание второго объекта класса может привести к проблемам в работе программы, имеет смысл сделать его одиночкой.

Для этого класс должен иметь ряд особенностей. Во-первых, класс должен иметь приватный конструктор.

```
public class MySingle {
    private MySingle() {
    }
}
```

Это приводит к тому, что создать объект данного класса становится невозможным. Например, попытка сделать:

```
MySingle ms = new MySingle();
```

Приведет к появлению ошибки "The constructor MySingle() is not visible".

Каким образом в таком случае можно создать объект? Для этого к классу добавляются два элемента: статическая ссылка на этот же класс, статический метод, возвращающий объект. Выглядит это следующим образом:

```
private static MySingle instance;
public static MySingle getMySingle() {
    if(inst == null) {
        inst = new MySingle();
    }
    return inst;
}
```

Данный метод возвращает ссылку на экземпляр объекта. Если он не существует, метод создает его, а если он уже есть, то есть не равен null, возвращается ссылка на объект. В результате объект получается в единственном экземпляре.

Таким образом, создание объекта будет иметь вид:

```
MySingle ms = MySingle.getMySingle();
```

Вот такое создание объекта ошибок не вызовет и будет работать корректно.

Единственная возможная проблема с такой реализацией, возможность появления ошибки, если объект создается почти одновременно двумя разными потоками. В этом случае может возникнуть ситуация, когда они практически одновременно пройдут условие и new будет выполнен два раза.

Чтобы обойти эту проблему, существует несколько решений. Самым простым из которых является синхронизация метода:

```
public static synchronized MySingle GetMySingle()
```

Правда, это решение может вызывать проблемы в случае, если метод вызывается часто, так как синхронизация несколько замедляет работу метода.

Метод Фабрика

Метод фабрика – метод предназначен для создания объектов.

Фабрика используется в тех случаях, когда в зависимости от условия надо создавать разные объекты с одним и тем же интерфейсом.

Также он может использоваться, если требуются дополнительные операции при создании объекта.

Например, программа должна создавать объекты для работы с изображениями. Для разных типов изображений созданы разные классы GifImage JpegImage и так далее, но при этом у них есть общий интерфейс Image. При создании объекта класс выбирается по строке с расширением файла. В такой ситуации имеет смысл создать метод фабрику примерно такого вида:

```
public class Images{
    public static Image createImage(String ext) {
        if(ext.equals("gif"))
            return new GifImage();
        else if(ext.equals("bmp"))
            return new BmpImage();
        ...
    }
    ...
}
```

После это создание объекта будет иметь вид:

```
Image firstImage = Images.createImage("bmp")
```

Такой подход избавляет от необходимости повторять операторы выбора в нескольких местах программы. При добавлении нового типа картинки будут вноситься изменения только в метод createImage, не затрагивая остальную часть программы.

Строитель

Паттерн Строитель (Builder) применяется в тех случаях, когда необходимо создать на основе исходных данных сложный составной объект.

Как правило, состоит из двух частей: объекта управляющего созданием Director, объекта-строителя Builder.

Часто получаемый в результате объект может иметь очень разное наполнение. Например, если стоит задача конвертации документа (это может быть файл .doc или .rtf) в другие форматы, например, HTML, чистый текст, LaTeX и т.д.

В этом случае объект Director разбирает исходный файл и вызывает методы объекта builder, для того чтобы создавать элементы другого формата. Если надо получать разные типы документов на выходе, сначала создается интерфейс или абстрактный класс строителя, в котором объявляются все методы для создания документов, а затем для каждого типа выходного документа создается строитель. Мы получаем столько строителей сколько надо типов выходных документов. Вот пример абстрактного строителя и строителя для HTML:

```
public abstract class DocumentBuilder {
    StringBuilder text = new StringBuilder();
    public abstract void createHeader(String headerText);
    public abstract void createParagraph(String parText);
    public String getText() {
        return text.toString();
    }
}

public class HTMLBuilder extends DocumentBuilder {
    @Override
    public void createHeader(String headerText) {
        text.append("<h1>");
        text.append(headerText);
        text.append("</h1>");
    }
    @Override
    public void createParagraph(String parText) {
        text.append("<p>");
        text.append(parText);
        text.append("</p>");
    }
}
```

Для других типов документов создаются аналогичные строители. В директоре мы можем выполнить следующий код:

```
DocumentBuilder builder = new HTMLBuilder();
builder.createHeader("Заголовок");
builder.createParagraph("Первый абзац");
builder.createHeader("Второй Заголовок");
builder.createParagraph("Второй Абзац");
String result = builder.getText();
```

В переменной `result` у нас окажется требуемый HTML код. Если надо другой тип документа, следует просто после `new` указать другого строителя.

Команда

Шаблон проектирования Команда (`Command`) используется в том случае, если необходимо отделить запуск какого-то метода от выбора, какой же метод мы должны выполнять. Например, нам необходимо выполнить распечатку названия дня недели по его номеру. Причем выбор дня удобнее сделать в одном методе, а саму распечатку выполнить в другом.

Для этого создается абстрактный класс `Command`, содержащий метод, который в будущем должен выполнять нужную задачу. Как правило, метод называется `execute` и объявляется абстрактным. Далее от `Command` создаются дочерние классы, в которых метод `execute` переопределяется с нужным наполнением.

```
public abstract class Command {
    public abstract void execute();
}

public class MonCommand extends Command {
    @Override
    public void execute() {
        System.out.println("Понедельник");
    }
}

public class TuCommand extends Command {
    @Override
    public void execute() {
        System.out.println("Вторник");
    }
}
```

Дальше в классе, который обычно называют клиентом (`Client`), выполняется выбор, какое из созданных действий будет выполняться. Например:

```
Command command;

if (n==1)
    command = new MonCommand();
else if (n==2)
    command = new TuCommand();
```

Затем класс, в котором должно выполняться действие, получает выбранный объект и запускает метод `execute`.

```
command.execute();
```

Класс, запускающий выполнение, называется `Invoker`. В самых простых случаях роль `Invoker` и `Client` может выполняться одним и тем же классом.

Кроме Command, Invoker, Client может присутствовать четвертый тип класса – Receiver. Его объект или объекты находятся внутри Command и непосредственно выполняют полезную работу.

Шаблон Command часто используют, если необходимо хранить историю выполняемых действий (в этом случае, объекты команд можно заносить в коллекцию или массив для хранения). Также данный шаблон может использоваться, если необходимо организовать очередь для выполнения действий и т.д.

Задания

Задание 68

Создайте простейший сервис с методом, который выводит на экран текущую дату. Сделайте его Singleton и используйте в основном теле программы.

Для заметок:

Задание 69

Создайте класс Person с полями: имя, фамилия, год рождения. Реализуйте у этого класса паттерн Строитель. Введите поля с клавиатуры и заполните объект класса Person с помощью паттерна Строитель.

Для заметок:

Задание 70

Создайте простейший логгер, выводящий сообщения об ошибках в текстовый файл. Объект логгера должен быть создан с помощью ШП Singleton. У объекта должен быть обязательным один метод, получающий на вход текст сообщения об ошибке и записывающий его в файл вместе с информацией о дате и времени происшествия.

Для заметок:

Reflection API. Аннотации

Reflection API

Рефлексия в Java – это механизм, с помощью которого можно вносить изменения и получать информацию о классах, интерфейсах, полях и методах во время выполнения, при этом не зная имен этих классов, методов и полей. Кроме того, Reflection API дает возможность создавать новые экземпляры классов, вызывать методы, а также получать или устанавливать значения полей.

Ограничения по работе:

- низкая производительность – поскольку рефлексия в Java определяет типы динамически, то она сканирует classpath, чтобы найти класс для загрузки. В результате чего снижается производительность программы;
- ограничения системы безопасности – рефлексия требует разрешения времени выполнения, которые не могут быть доступны для систем, работающих под управлением менеджера безопасности (Java Security Manager);
- нарушения безопасности приложения – с помощью рефлексии мы можем получить доступ к части кода, к которой мы не должны получать доступ. Например, мы можем получить доступ к закрытым полям класса и менять их значения. Это может быть серьезной угрозой безопасности;
- сложность в поддержке – код, написанный с помощью рефлексии, трудно читать и отлаживать, что делает его менее гибким и трудно поддерживаемым.

Используя рефлексия, мы можем работать с полями – переменными-членами класса. В этом нам помогает Java класс `java.lang.reflect.Field`. С помощью него в рантайме мы можем устанавливать значения и получать данные с полей.

Получить поля класса с помощью рефлексии можно с помощью следующей строки кода:

```
Class mClassObject = SomeObject.class  
Field[] fields = mClassObject.getFields();
```

Каждый элемент массива содержит экземпляр `public` поля, объявленного в классе.

Если вы знаете имя поля, к которому вы хотите получить доступ, достаточно вызвать следующую строку:

```
Class mClassObject = SomeObject.class  
Field field = mClassObject.getField("fieldName");
```

Название поля можно получить следующим способом:

```
Field field = mClassObject.getField("fieldName");  
String fieldName = field.getName();
```

Тип поля можно получить следующим способом:

```
Field field = mClassObject.getField("fieldName");  
Object fieldType = field.getType();
```

Получить и установить значения полей можно:

```
Class mClassObject = SomeObject.class
```

```
Field field = mClassObject.getField("fieldName");
SomeObject instance = new SomeObject();
Object value = field.get(instance);
field.set(instance, value);
```

Используя рефлексия в Java, мы можем получать информацию о методах и вызывать их в рантайме. Для этого используется класс `java.lang.reflect.Method`:

```
Class mClassObject = SomeObject.class
Method[] methods = mClassObject.getMethods();
```

Нам не нужно получать массив со всеми методами, если нам известны точные типы параметров метода, который мы хотим использовать. Например, у нас есть метод под названием "sayHello", который принимает String в качестве параметра. Получить объект Method для него можно так:

```
Class mClassObject = SomeObject.class
Method method = mClassObject.getMethod("sayHello", new
Class[]{String.class});
```

Если такого метода нет, то будет выброшен `NoSuchMethodException`.

Если метод `sayHello()` без параметров, то нужно передать `null` в методе `getMethod()`:

```
Method method = mClassObject.getMethod("sayHello", null);
```

Получить параметры метода можно так:

```
Class[] parameterTypes = method.getParameterTypes();
```

В строчке ниже мы можем получить тип возвращаемого значения:

```
Class returnType = method.getReturnType();
```

Вызов метода с помощью Java рефлексии:

```
Class mClassObject = SomeObject.class
Method method = mClassObject.getDeclaredMethod("sayHello", parameterTypes)
method.invoke(objectToInvokeOn, params)
```

Аннотации

Аннотация (`@Annotation`) – специальная форма метаданных, которая может быть добавлена в исходный код.

Аннотированы могут быть пакеты, классы, методы, поля класса и параметры, локальные переменные. Для этого аннотация ставится перед заголовком аннотируемого элемента.

Пример:

```
@Override
public void go() {
    ...
}
```


Назначение аннотаций:

1. Информация для компилятора – аннотации могут использоваться компилятором для определения ошибки, подавления сообщений с предупреждениями.

2. Обработка на этапе развертывания и компиляции – инструменты разработки могут анализировать аннотации и на их основе генерировать информацию (например, XML).

3. Обработка на этапе выполнения – некоторые аннотации могут быть доступны на этапе выполнения.

Наиболее часто используемые аннотации:

@Deprecated – аннотация **@Deprecated** помечает элемент как устаревший, это означает, что его не желательно использовать. Компилятор создает предупреждение, если программа использует методы, поля или классы, помеченные как устаревшие. Пример:

```
@Deprecated
public void setDate(int date)
```

@Override – аннотация **@Override** сообщает компилятору, что мы собираемся переопределить метод родительского класса.

Пример:

```
@Override
public void go() {
    ...
}
```

При использовании данной аннотации компилятор будет проверять, соответствует ли заголовок вашего метода заголовку родительского. Если будет обнаружено несоответствие, вы получите ошибку компиляции. Например, если попытаетесь сделать метод `String toString(int x)`. Без аннотации ошибки не возникает, так как сам по себе синтаксис не нарушен. С аннотацией возникает ошибка, поскольку в родительском методе входных параметров нет.

@SuppressWarnings – аннотация **@SuppressWarnings** используется для устранения предупреждений, создаваемых компилятором.

Пример:

```
@SuppressWarnings("deprecation")
public void useDeprecatedMethod() {
    Date date = new Date();
    date.setDate(...);
}
```

Создание собственных аннотаций

Для создания своей аннотации используется слово **@interface**. После него ставятся фигурные скобки и в них объявляются свойства аннотации.

Пример:

```
public @interface Version {
    String info();
}
```

Данная аннотация содержит одно свойство строкового типа. Причем свойства в аннотациях объявляются с круглыми скобками, как методы.

В дальнейшем данную аннотацию можно использовать вот таким образом:

Пример:

```
@Version(info = "3.5.3")
public void checkAnnotation() {
    ...
}
```

Если свойство единственное, писать имя и знак равно не обязательно `@Version("3.5.3")`. Если свойств несколько, их значения надо перечислять через запятую с именами и знаками равно.

При создании аннотации можно задать, на какой стадии данная аннотация будет доступна. Делается это с помощью аннотации `@Retention`. У нее могут быть следующие значения:

- `RetentionPolicy.SOURCE` – аннотация используется на этапе компиляции и должна отбрасываться компилятором;
- `RetentionPolicy.CLASS` – аннотация будет записана в class-файл компилятором, но не должна быть доступна во время выполнения (runtime). Значение по умолчанию;
- `RetentionPolicy.RUNTIME` – аннотация будет записана в class-файл и доступна во время выполнения через reflection.

Пример

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String stringValue();
    int intValue();
}
```

Также можно задать, для какого типа элементов данная аннотация может использоваться. Для этого существует аннотация `@Target`:

- `@Target(ElementType.PACKAGE)` – только для пакетов;
- `@Target(ElementType.TYPE)` – только для классов;
- `@Target(ElementType.CONSTRUCTOR)` – только для конструкторов;
- `@Target(ElementType.METHOD)` – только для методов;
- `@Target(ElementType.FIELD)` – только для атрибутов (переменных) класса;
- `@Target(ElementType.PARAMETER)` – только для параметров метода;
- `@Target(ElementType.LOCAL_VARIABLE)` – только для локальных переменных.

Например, аннотация, которая может применяться только к методам:

```
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value();
}
```

Задания

Задание 71

Создать класс Map с произвольным набором полей и методов (не менее 3). Создать метод, который распечатает информацию о классе с помощью рефлексии. Вызвать метод с помощью рефлексии из основной программы.

Для заметок:

Задание 72

Создайте класс с методом printHelloWorld(). Вызвать метод с помощью рефлексии из основной программы.

Для заметок:

Задание 73

Создать собственную аннотацию @AcademyInfo с полем year. Создать метод, помеченный этой аннотацией с заданным year, и метод без нее. С помощью рефлексии проверить наличие данной аннотации у этих методов из основной программы.

Для заметок:

Основы XML/JSON

Понятие XML, достоинства и недостатки

XML представляет собой язык разметки, позволяющий достаточно легко хранить в текстовом виде сложные иерархические данные. Название расшифровывается как eXtensible Markup Language, что переводится как расширяемый язык разметки.

Главное достоинство XML по сравнению, например, с простыми конфигурационными файлами – возможность представления сложных иерархически

построенных данных практически любой глубины и сложности. С помощью XML можно без особых проблем записывать такие структуры данных, как списки, деревья и т.д.

Еще одно неоспоримое достоинство XML – его широчайшая распространенность, средства для работы с XML имеются в большинстве современных языков программирования.

К недостаткам можно отнести некоторую избыточность XML. Документ на XML, как правило, значительно больше по объему, чем специализированный двоичный документ такого же формата.

Из-за распространенности его часто используют там, где его мощь является избыточной, а также где можно применять более простые решения.

Несмотря на то, что XML – текстовый формат, сложные документы могут быть трудными для восприятия человеком.

В связи с большой гибкостью языка, одна и та же структура может быть записана большим количеством способов.

XML не имеет встроенных типов данных.

Но, несмотря на перечисленные недостатки, XML чрезвычайно удобен для работы со многими видами данных и является одним из наиболее распространенных средств хранения данных.

XML имеет много общего с HTML, так как они произошли от одного общего предка, языка SGML, который оказался слишком сложным, и поэтому не получил такого широкого распространения. HTML и XML являются фактически его подмножествами.

Основные элементы документа XML, понятие тега

Документ на XML представляет собой текстовый документ, в котором кроме текста расположены так называемые теги – команды, которые задают структуру документа. Как отличить тег от обычного текста? Тег всегда начинается с символа < и заканчивается символом >. Сразу после символа < идет имя тега. После имени тега он может заканчиваться символом >, либо иметь через пробел один или несколько параметров вида имя_параметра="значение", пробельными символами. Параметры также называют атрибутами тега. Таким образом, общий вид тега:

```
<имя_тега имя_параметра="" имя_параметра="" ...>
```

Например, тег описывающий двухмерную точку:

```
<point x="2" y="3">
```

Здесь имя тега point, два атрибута x и y, с соответствующими значениями. Следует обратить внимание, что значения атрибутов всегда берутся в кавычки.

Обычно теги XML являются двойными, кроме первого тега, есть также второй так называемый закрывающий тег. Он имеет такое же имя, как и первый, но перед ним стоит знак /.

```
<point x="2" y="3"></point>
```

Обычно тег является двойным, если между первым и вторым находится еще информация. Если как в данном примере ее нет, тег может быть одинарным и тогда знак / надо поставить перед закрывающейся угловой скобкой.

```
<point x="2" y="3" />
```

Между тегами может располагаться какая-то текстовая информация, либо другие теги. В качестве иллюстрации того, что одна и та же информация может быть записана несколькими разными способами, второй вариант описания такой же точки:

```
<point>
  <x>2</x>
  <y>3</y>
</point>
```

В данном примере для записи координат использованы не параметры тега, а новые теги, вложенные в первый. В такой записи все теги должны быть строго двойными.

Структура документа XML

Самой первой строкой XML документа рекомендуется вставлять строку вида:

```
<?xml version = "1.0" ?>
```

Эта строка дает понять, что это XML документ. Она не является обязательной. Также в ней часто указывается кодировка документа.

Дальше обычно следует тег вида `<!DOCTYPE>`, в котором находится ссылка на описание структуры данного документа. Такой тег также является необязательным, но рекомендованным.

После этих двух строк следует уже содержимое документа. Стандарт XML требует, чтобы содержимое документа находилось внутри одного тега, называемого корневым тегом. Все остальные теги и текст документа находятся внутри него. Например, документ содержит список точек. В этом случае он будет выглядеть так:

```
<pointsList>
  <point>
    <x>2</x>
    <y>3</y>
  </point>
  <point>
    <x>9</x>
    <y>3</y>
  </point>
</pointsList>
```

Таким образом, мы имеем корневой тег `pointsList` и вложенные теги `point`, в которые вложены теги `x` и `y`. Теги `point` называются дочерними к тегу `pointsList`, а теги `x` и `y` – дочерними к тегу `point`.

Иногда необходимо вставить в текст знак `<`, чтобы он не воспринимался как начало тега. В таких ситуациях используются специальные знаки `<` и `>` для `<` и `>` соответственно.

Если во вставляемом тексте подобных знаков много, можно воспользоваться еще одним механизмом:

```
<![CDATA[< & > are my favorite delimiters]]>
```

Фрагмент текста, начинающийся с `<![CDATA[` и заканчивающийся `]]>` воспринимается только как текст, какие бы знаки и теги в нем не находились.

Также в документ можно вставлять комментарий. Он должен начинаться с `<!--` и заканчиваться `-->`.

DTD, схема XML и обзор XSD

Как правило, при создании XML документа он должен иметь определенную структуру, определенный набор тегов и правила, какие теги могут быть дочерними, какие родительскими и т.д. Для этого существует механизм DTD (document type definition – определение типа документа). Описание структуры может располагаться либо внутри тега DOCTYPE, либо в отдельном файле, и тогда DOCTYPE должен содержать ссылку на него. Например:

```
<!DOCTYPE pointsList SYSTEM "pl.dtd">
```

В данном случае указывается ссылка на файл pl.dtd.

DOM и SAX и StAX parser

Общие понятия

Для работы с XML документом необходимо провести его анализ, разбить на отдельные составные элементы и получить к ним доступ.

Программа, выполняющая данные задачи, называется парсером (parser). Существует три основных подхода к разбору XML документа: DOM, SAX и StAX анализаторы.

DOM при анализе документа создает из него древовидную структуру.

SAX проходит по документу, генерируя события.

StAX, как и SAX, последовательно проходит по документу, но использует не событийный подход, а итератор.

DOM – анализатор, как правило, прост в использовании и что самое главное, обеспечивает произвольный доступ к элементам документа.

SAX и StAX чаще используется, когда документ слишком велик и будет занимать много памяти. Кроме того, его имеет смысл использовать, если нас интересуют несколько отдельных элементов документа и не интересует все остальное.

Использование DOM

Итак, каким образом происходит разбор документа в DOM? Разбор документа осуществляется специальным объектом класса DocumentBuilder. Он создается специальным объектом-фабрикой класса DocumentBuilderFactory. Таким образом, сначала создается объект-фабрика, затем с помощью метода newDocumentBuilder создается объект builder. При его создании, возможно, появление исключения.

После того как парсер создан, он может выполнять разбор документов с помощью метода parse. Данный метод на вход может получать либо объект класса File, либо URL, либо файловый поток. При операции разбора может возникать исключение ввода-вывода или исключение разбора файла. В результате выполнения метода parse возвращается объект документа. Таким образом, разбор документа может выглядеть следующим образом:


```

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = null;
Document doc= null;
try {
    builder = factory.newDocumentBuilder();
} catch (ParserConfigurationException e) {
    e.printStackTrace();
}
File f = new File("pl.xml");
try {
    doc = builder.parse(f);
} catch (SAXException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}

```

После выполнения данного кода, в объекте doc будет полное представление документа pl.xml.

Следующий шаг – получение доступа к элементам документа. Чтобы получить доступ к корневому элементу, можно воспользоваться методом `getDocumentElement`. Сам элемент при этом получается в виде объекта класса `Element`.

```
Element root = doc.getDocumentElement();
```

Когда элемент получен, мы можем с ним работать. Например, чтобы получить имя тега элемента, следует воспользоваться методом `getTagName`.

```
System.out.println(root.getTagName());
```

В нашем случае будет распечатано имя корневого тега `pointsList`.

Кроме имени тега, можно получить значение любого из параметров тега с помощью метода `getAttribute("имя атрибута")`.

Чтобы получить дочерние элементы, можно воспользоваться методом `getChildNodes`. Этот метод возвращает коллекцию класса `NodeList`, содержащую дочерние элементы в виде объектов класса `Node`. Здесь используется класс `Node`, а не элемент, потому что элементы могут быть не только тегами, но и простым текстом. Теперь, чтобы получить имя объекта класса `Node`, можно воспользоваться методом `getNodeName`.

```

NodeList nList = root.getChildNodes();
for(int i=0;i < nList.getLength();i++){
    System.out.println(nList.item(i).getNodeName());
}

```

В данном примере распечатываются имена всех дочерних элементов. Причем выводятся все элементы. Текст между тегами воспринимается как отдельные текстовые элементы. Если нас интересуют только теги, вывод можно написать следующим образом:

```
if(nList.item(i) instanceof Element)
```



```
System.out.println(nList.item(i).getNodeName());
```

Мы рассмотрели получение дочерних элементов, но существует возможность аналогично получить атрибуты элемента с помощью метода `getAttributes`. Данный метод возвращает коллекцию типа `NamedNodeMap`.

```
NamedNodeMap attributes = element.getAttributes();
```

Кроме получения дочерних элементов, узел дает возможность получить своих соседей с помощью методов `getNextSibling()` и `getPreviousSibling()`. Эти методы возвращают ссылки на следующий и предыдущий равноправный элемент соответственно. Для текстовых узлов и атрибутов можно также узнать значение с помощью `getNodeValue()`.

Использование StAX

Для использования StAX также в начале создается фабрика `XMLInputFactory`:

```
XMLInputFactory factory = XMLInputFactory.newFactory();
```

Следующим шагом следует создать объект итератора класса `XMLStreamReader`:

```
XMLStreamReader reader = factory.createXMLStreamReader(new  
FileInputStream("config.xml"));
```

После этого происходит последовательное чтение документа с помощью методов аналогичных итератору: `next` и `hasNext`. Метод `hasNext` проверяет, дошли ли мы до конца, и возвращает истину, если еще остались элементы, и ложь, если мы уперлись в конец документа. Метод `next` читает очередной элемент документа, но, в отличие от итераторов в коллекциях, возвращает не объект, а целое число, означающее, какой элемент прочитан.

Могут быть следующие значения:

- `START_DOCUMENT` – начало документа;
- `START_ELEMENT` – открывающий тег;
- `CDATA` – блок CDATA;
- `CHARACTERS` – текстовый элемент между тегами;
- `COMMENT` – комментарий;
- `END_DOCUMENT` – конец документа;
- `END_ELEMENT` – закрывающий тег.

Все эти константы объявлены в интерфейсе `XMLStreamConstants`. После того как элемент прочитан с помощью `next`, можно извлекать из него данные другими методами итератора. Вот некоторые из них:

- **`String getLocalName()`** – получить имя тега;
- **`int getAttributeCount()`** – получить количество параметров в теге;
- **`String getAttributeLocalName(int index)`** – получить имя параметра по его номеру;
- **`String getAttributeValue(int index)`** – получить значение параметра по номеру;
- **`String getText()`** – получить текст у текстового элемента.

Рассмотрим следующий пример. Допустим, что у нас имеется xml файл, в котором хранятся настройки подключения к базе данных:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <url>jdbc:mysql://localhost:3306/ListExpenses</url>
  <user>root</user>
  <pass>1234</pass>
</config>
```

Наша задача: распечатать эти настройки.

```
boolean isUrl = false;
boolean isName = false;
boolean isPass = false;
XMLInputFactory factory = XMLInputFactory.newFactory();
try {
    XMLStreamReader reader = factory.createXMLStreamReader(new
FileInputStream("config.xml"));
    while(reader.hasNext()) {
        int res = reader.next();
        if(res == reader.START_ELEMENT){
            if(reader.getLocalName().equals("url"))
                isUrl = true;
            else if(reader.getLocalName().equals("user"))
                isName = true;
            else if(reader.getLocalName().equals("pass"))
                isPass = true;
        } else if(res == reader.CHARACTERS){
            if(isUrl){
                System.out.println("Url:" + reader.getText());
                isUrl = false;
            } else if(isName){
                System.out.println("Username:" + reader.getText()); isName =
                false;
            } else if(isPass){
                System.out.println("Password:" + reader.getText());
                isPass = false;
            }
        }
    }
} catch (FileNotFoundException | XMLStreamException e) {
    e.printStackTrace();
}
```

В данном примере мы создаем фабрику, затем создаем итератор reader и последовательно читаем документ. Если очередной обнаруженный элемент

является тегом с нужным нам именем, устанавливается переменная флаг, а затем, когда обнаруживается текстовое содержимое элемента, оно выводится, а флаг сбрасывается.

В случае, если читаемый документ содержит ошибки, возникает `XMLStreamException`.

XSL/XSLT

Технология XSL/XSLT позволяет преобразовывать XML файлы в другой XML или любой другой формат (например: HTML, чистый текст, rtf и т.д.).

Понятие JSON

JSON – это текстовый формат обмена данными, который легко читается людьми. Считается более подходящим для сериализации сложных структур. Часто применяется в веб-приложениях для обмена данными между браузером и сервером.

Синтаксис JSON

Синтаксис представлен набором пар ключ:значение. В качестве ключа используется только строка. В качестве значений могут использоваться следующие типы:

- объект – это неупорядоченное множество пар ключ:значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ:значение отделяются друг от друга запятыми;
- массив (одномерный) – это упорядоченное множество значений. Массив заключается в квадратные скобки «[]». Значения разделяются запятыми;
- число;
- литералы true, false и null;
- строка – это упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \', \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке Unicode в виде \uFFFF.

Пример

Создадим Person JSON в файле person.txt:

```
{
  "firstName": "Иван",
  "lastName": "Иванов",
  "address": {
    "streetAddress": "Московское ш., 101, кв.101",
    "city": "Ленинград",
```

```
        "postalCode": "101101"
    },
    "phoneNumbers": [
        "812 123-1234",
        "916 123-4567"
    ]
}
```

Создадим класс Person:

```
public class Person {
    private String firstName;
    private String lastName;
    private Address address;
    private List<String> phoneNumbers;
    public String getFirstName() {
        return name;
    }
    public void setFirstName(String name) {
        this.name = name;
    }
    public String getLastName() {
        return name;
    }
    public void setLastName(String name) {
        this.name = name;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
    public List<String> getPhoneNumbers () {
        return phoneNumbers;
    }
    public void setPhoneNumbers (List<String> phoneNumbers) {
        this. phoneNumbers = phoneNumbers;
    }
}
```

Напишем код, который преобразует JSON в наш объект класса Person и обратно в строку:

```

public class PersonMapperExample {
    public static void main(String[] args) throws IOException {
        byte[] jsonData = Files.readAllBytes(Paths.get("person.txt"));
        ObjectMapper objectMapper = new ObjectMapper();
        Person person = objectMapper.readValue(jsonData, Person.class);
        System.out.println("Person Object " + person);
        objectMapper.configure(SerializationFeature.INDENT_OUTPUT, true);
        StringWriter stringPerson = new StringWriter();
        objectMapper.writeValue(stringPerson, person);
        System.out.println("Person JSON is\n"+ stringPerson);
    }
}

```

Задание

Задание 74

Напишите программу, которая будет разбирать xml файл:

```

<pointsList>
  <point>
    <x>2</x>
    <y>3</y>
  </point>
  <point>
    <x>9</x>
    <y>3</y>
  </point>
</pointsList>

```

и выводит его на экран в текстовом виде. Каждая точка должна выводиться на отдельной строке в виде двух чисел, разделенных запятой, при этом должна выводиться единица измерения. Например: 10px, 30px.

Для заметок:

Задание 75

Напишите программу, которая будет разбирать xml файл, сделанный в задании 74 с помощью StAX, и выводить его на экран в текстовом виде. Каждая точка должна выводиться на отдельной строке в виде двух чисел, разделенных запятой, при этом должна выводиться единица измерения. Например: 10px, 30px.

Для заметок:

Задание 76

Напишите программу, которая будет разбирать json файл:


```
{
  "id": 123,
  "name": "Tester",
  "permanent": true,
  "address": {
    "street": "Skryganova",
    "city": "Minsk",
    "zipcode": 220000
  },
  "phoneNumbers": [
    123456,
    987654
  ],
  "role": "Manager",
  "cities": [
    "Minsk",
    "Grodno"
  ]
}
```


и выводить его на экран в текстовом виде.


Для заметок:


IT-Academy




 +375 (29) 222-24-60

 +375 (44) 570-22-22

 +375 (25) 760-24-60

 info@it-academy.by

 Минск, ул. Скрыганова, 14,
5-й этаж, каб. 59