**1. Структура программы. Комментарии. Консольный ввод/вывод в Java.**

**Структура программы**  
Основным строительным блоком программы на языке Java являются инструкции. Каждая инструкция выполняет некоторое действие, например, вызовы методов, объявление переменных и присвоение им значений. После завершени инструкции в Java ставится точка с запятой (;). Данный знак указывает компилятору на конец инструкции. Например: System.out.println("Hello Java!"); Данная строка представляет вызов метода System.out.println, который выводит на консоль строку "Hello Java!". Вызов метода является инструкцией и поэтому завершается точкой с запятой. Кроме отдельных инструкций распространенной конструкцией является блок кода. Блок кода содержит набор инструкций, он заключается в фигурные скобки, а инструкции помещаются между открывающей и закрывающей фигурными скоб ками:  
{  
 System.out.println("Hello!");  
 System.out.println("Welcome to Java!");  
}

**Комментарии**  
Код программы может содержать комментарии. Комментарии позволяют понять смысл программы, что делают те или иные ее части. При компиляции комментарии игнорируются и не оказывают никакого влияния на работу приложения и на его размер. В Java есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //. А многострочный комментарий заключается между символами /\* текст комментария \*/. Онможет размещаться на нескольких строках.

**Консольный ввод/вывод в Java**  
Наиболее простой способ взаимодействия с пользователем представляет консоль: мы можем выводить на консоль некоторую информацию или, наоборот, считывать с консоли некоторые данные. Для взаимодействия с консолью в Java применяется класс System, а его функциональность собственно обеспечивает консольный ввод и вывод.  
**Вывод на консоль**  
Для создания потока вывода в классе System определен объект out. В этом объекте определен метод println, который позволяет вывести на консоль некоторое значение с последующим переводом курсора консоли на следующую строку. Например:  
System.out.println("Hello world!");  
System.out.println("Bye world...");  
В метод println передается любое значение, как правило, строка, которое надо вывести на консоль.   
При необходимости можно и не переводить курсор на следующую строку. В этом случае можно использовать метод System.out.print(), который аналогичен println за тем исключением, что не осуществляет перевода на следующую строкy. Но с помощью метода System.out.print также можно осуществить перевод каретки на следующую строку. Для этого надо использовать escape последовательность \n.

в Java есть также функция для форматированного вывода: System.out.printf().

Ряд спецификаторов для типов данных:  
%x: для вывода шестнадцатеричных чисел  
%f: для вывода чисел с плавающей точкой

%e: для вывода чисел в экспоненциальной форме, например, 1.3e+01  
%c: для вывода одиночного символа

%s: для вывода строковых значений

Например:

{

String name = "Tom";  
 int age = 30;  
 float height = 1.7f;  
 System.out.printf("Name: %s Age: %d Height: %.2f \n", name, age, height);

}

**Ввод с консоли**  
Для получения ввода с консоли в классе System определен объект in. Однако непосредственно через объект System.in не очень удобно работать, поэтому, как правило, используют класс Scanner, который, в свою очередь использует System.in. Так как класс Scanner находится в пакете java.util, то мы вначале его импортируем с помощью инструкции 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 еще ряд методов, которые позволяют получить введенные пользователем значения:  
next(): считывает введенную строку до первого пробела  
nextLine(): считывает всю введенную строку  
nextInt(): считывает введенное число int  
nextDouble(): считывает введенное число double  
nextBoolean(): считывает значение boolean  
nextByte(): считывает введенное число byte  
nextFloat(): считывает введенное число float

nextShort(): считывает введенное число short

**2. Переменные и константы.**

**Переменные**  
Для хранения данных в программе предназначены переменные. Переменная представляет именованную область памяти, которая хранит значение определенного типа. Каждая переменная имеет тип, имя и значение. Тип определяет, какую информацию может хранить переменная или диапазон допустимых значений. Переменные объявляются следующим образом:  
тип\_данных имя\_переменной; Например, определим переменную, которая будет называться x и будет иметь тип int: int x; В этом выражении мы объявляем переменную x типа int. То есть x будет хранить некоторое число не больше 4 байт. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

* имя может содержать любые алфавитно-цифровые символы, а также знак подчеркивания, при этом первый символ в имени не должен быть цифрой;
* в имени не должно быть знаков пунктуации и пробелов;
* имя не может быть ключевым словом языка Java;

Кроме того, при объявлении и последующем использовании надо учитывать, что Java - регистрозависимый язык, поэтому следующие объявления int num; и int NUM; будут представлять две разных переменных. Объявив переменную, мы можем присвоить ей значение. Также можно присвоить значение переменной при ее объявлении. Этот процесс называется инициализацией.  
Если мы не присвоим переменной значение до ее использования, то мы можем получить ошибку.  
Через запятую можно объявить сразу несколько переменных одного типа. Также можно их сразу инициализировать. Отличительной особенностью переменных является то, что мы можем в процессе работы программы изменять их значение.

**Ключевое слово var**  
Начиная с Java 10 в язык было добавлено ключевое слово var, которое также позволяет определять переменную:

var x = 10;  
System.out.println(x); // 10  
Слово var ставится вместо типа данных, а сам тип переменной выводится из того значения, которое ей присваивается. Например, переменной x присваивается число 10, значит, переменная будет представлять тип int. Но если переменная объявляется с помощью var, то мы обязательно должны инициализировать ее.  
**Константы**  
Кроме переменных, в Java для хранения данных можно использовать константы. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется также, как и переменная, только вначале идет ключевое слово final:  
final int LIMIT = 5;  
System.out.println(LIMIT); // 5  
// LIMIT=57; // так мы уже не можем написать, так как LIMIT - константа  
Как правило, константы имеют имена в верхнем регистре. Константы позволяют задать такие переменные, которые не должны больше изменяться. Например, если у нас есть переменная для хранения числа pi, то мы можем объявить ее константой, так как ее значение постоянно.

**3. Базовые типы данных.**

**Типы данных**  
Одной из основных особенностей Java является то, что данный язык является строго типизированным. А это значит, что каждая переменная и константа представляет определенный тип и данный тип строго определен. Тип данных определяет диапазон значений, которые может хранить переменная или константа. Рассмотрим систему встроенных базовых типов данных, которая используется для создания переменных в Java. Она представлена следующими типами.  
**boolean**: хранит значение true или false  
**byte**: хранит целое число от -128 до 127 и занимает 1 байт  
**short**: хранит целое число от -32 768 до 32 767 и занимает 2 байта  
**int**: хранит целое число от -2 147 483 648 до 2 147 483 647 и занимает 4 байта  
**long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт  
**double**: хранит число с плавающей точкой от ±4.9\*10 -324 до ±1.8\*10 308 и занимает 8 байт  
**float**: хранит число с плавающей точкой от -3.4\*10 38 до 3.4\*10 38 и занимает 4 байта  
**char**: хранит одиночный символ в кодировке UTF-16 и занимает 2 байта, поэтому диапазон хранимых значений от 0 до 65535  
При этом переменная может принимать только те значения, которые соответствуют ее типу. Если переменная представляет целочисленный тип, то она не может хранить дробные числа.  
**Целые числа**  
Все целочисленные литералы, например, числа 10, 4, -5, воспринимаются как значения типа int, однако мы можем присваивать целочисленные литералы другим целочисленным типам: byte, long, short. В этом случае Java автоматически осуществляет соответствующие преобразования.  
Однако если мы захотим присвоить переменной типа long число, которое выходит за пределы допустимых значений для типа int, то мы столкнемся с ошибкой во время компиляции.  
Как правило, значения для целочисленных переменных задаются в десятичной системе счисления, однако мы можем применять и другие системы счисления. Также целые числа поддерживают разделение разрядов числа с помощью знака подчеркивания.  
**Числа с плавающей точкой**  
При присвоении переменной типа float дробного литерала с плавающей точкой, например, 3.1, 4.5 и т.д., Java автоматически рассматривает этот литерал как значение типа double. И чтобы указать, что данное значение должно рассматриваться как float, нам надо использовать суффикс f.  
**Символы и строки**  
В качестве значения переменная символьного типа получает одиночный символ, заключенный в одинарные кавычки: char ch='e'. Кроме того, переменной символьного типа также можно присвоить целочисленное значение от 0 до 65535. В этом случае переменная будет хранить символ, а целочисленное значение будет указывать на номер символа в таблице символов Unicode (UTF-16).   
Еще одной формой задания символьных переменных является шестнадцатеричная форма: переменная получает значение в шестнадцатеричной форме, которое следует после символов '\u'. Например, char ch='\u0066'; опять же будет хранить символ 'f'. Символьные переменные не стоит путать со строковыми, 'a' не идентично "a". Строковые переменные представляют объ ект String, который в отличие от char или int не является примитивным типом в Java.

Кроме собственно символов, которые представляют буквы, цифры, знаки препинания, прочие символы, есть специальные наборы символов, которые называют управляющими последовательностями. Например, самая популярная последовательность \n. Она выполняет перенос на следующую строку.

**4. Арифметические операции. Условные выражения. Операции присваивания и приоритет операций.**

**Арифметические операции**  
Большинство операций в Java аналогичны тем, которые применяются в других си-подобных языках. Есть унарные операции (выполняются над одним операндом), бинарные - над двумя операндами, а также тернарные - выполняются над тремя операндами. Операндом является переменная или значение (например, число), участвующее в операции. Рассмотрим все виды операций. В К бинарным операциям относят следующие:  
+ - операция сложения двух чисел

- - операция вычитания двух чисел

\* - операция умножения двух чисел

/ - операция деления двух чисел

% - получение остатка от деления двух чисел

При делении стоит учитывать, что если в операции участвуют два целых числа, то результат деления будет округляться до целого числа, даже если результат присваивается переменной float или double. Чтобы результат представлял число с плавающей точкой, один из операндов также должен представлять число с плавающей точкой.

Также есть две унарные арифметические операции, которые производятся над одним числом: ++ (инкремент) и -- (декремент). Каждая из операций имеет две разновидности: префиксная и постфиксная:  
++ (префиксный инкремент) - увеличение переменной на единицу, например, z=++y (вначале значение переменной y увеличивается на 1, а затем ее значение присваивается переменной z)

++ (постфиксный инкремент) - увеличение переменной на единицу, например, z=y++ (вначале значение переменной y присваивается переменной z, а потом значение переменной y увеличивается на 1)

-- (префиксный декремент) - уменьшение переменной на единицу, например, z=--y (вначале значение переменной y уменьшается на 1, а потом ее значение присваивается переменной z)

-- (постфиксный декремент) - z=y-- (сначала значение переменной y присваивается переменной z, а затем значение переменной y уменьшается на 1)

Операции в порядке уменьшения приоритета:  
++ (инкремент), -- (декремент)  
\* (умножение), / (деление), % (остаток от деления)  
+ (сложение), - (вычитание)

**Условные выражения**  
Условные выражения представляют собой некоторое условие и возвращают значение типа boolean, то есть значение true(если условие истинно), или значение false (если условие ложно). К условным выражениям относятся операции сравнения и логические операции.  
**Операции сравнения**  
В операциях сравнения сравниваются два операнда, и возвращается значение типа boolean - true, если выражение верно, и false, если выражение неверно.

«==» - сравнивает два операнда на равенство и возвращает true (если операнды равны) и false (если операнды не равны)  
«!=» - сравнивает два операнда и возвращает true, если операнды НЕ равны, и false, если операнды равны  
«<» - (меньше чем) Возвращает true, если первый операнд меньше второго, иначе возвращает false  
«>» - (больше чем)  
«>=» - (больше или равно)Возвращает true, если первый операнд больше или равен второму, иначе возвращает false  
«<=» - (меньше или равно)Возвращает true, если первый операнд меньше или равен второму, иначе возвращает false  
**Логические операции**  
В Java есть логические операции, которые представляют условие и возвращают true или false и обычно объединяют несколько операций сравнения. К логическим операциям относят следующие:

| - c=a|b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

& - c=a&b; (c равно true, если и a, и b равны true, иначе c будет равно false)  
! - c=!b; (c равно true, если b равно false, иначе c будет равно false)  
^ - c=a^b; (c равно true, если либо a, либо b (но не одновременно) равны true, иначе c будет равно false)

|| - c=a||b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false)

&& - c=a&&b; (c равно true, если и a, и b равны true, иначе c будет равно false)  
Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны. Выражение c=a|b; будет вычислять сначала оба значения - a и b и на их основе выводить результат. выражении же c=a||b; вначале будет вычисляться значение a, и если оно равно true, то вычисление значения b уже смысла не имеет, так как у нас в любом случае уже c будет равно true. Значение b будет вычисляться только в том случае, если a равно false. То же самое касается пары операций &/&&. В выражении c=a&b; будут вычисляться оба значения - a и b. В выражении же c=a&&b; сначала будет вычисляться значение a, и если оно равно false, то вычисление значения b уже не имеет смысла, так как значение c в любом случае равно false. Значение b будет вычисляться только в том случае, если a равно true.

**Операции присваивания**  
Рассмотрим операции присваивания:

= просто приравнивает одно значение другому: c=b;

+= c+=b; (переменной c присваивается результат сложения c и b)  
-= c-=b; (переменной c присваивается результат вычитания b из c)  
\*= c\*=b; (переменной c присваивается результат произведения c и b)

/= c/=b; (переменной c присваивается результат деления c на b)  
%= c%=b; (переменной c присваивается остаток от деления c на b)  
&= c&=b; (переменной c присваивается значение c&b)  
|= c|=b; (переменной c присваивается значение c|b)  
^= c^=b; (переменной c присваивается значение c^b)  
<<= c<<=b; (переменной c присваивается значение c<<b)  
>>= c>>=b; (переменной c присваивается значение c>>b)  
>>>= c>>>=b; (переменной c присваивается значение c>>>b)

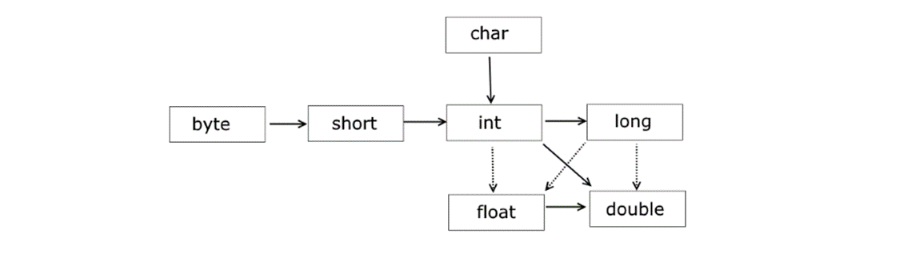
**Приоритет операций**  
При работе с операциями важно понимать их приоритет:

|  |
| --- |
| expr++ expr-- |
| ++expr --expr +expr -expr ~ ! |
| / % |
| + - |
| << >> >>> |
| < > <= >= instanceof |
| == != |
| & |
| ^ |
| | |
| && |
| || |
| ? : (тернарный оператор) |
| = += -= \*= /= %= &= ^= |= <<= >>= >>>= (операторы присваивания) |

Чем выше оператор в этой таблице, тем больше его приоритет. При этом скобки повышают приоритет операции, используемой в выражении.

5. Преобразования базовых типов данных.

Каждый базовый тип данных занимает определенное количество байт памяти. Это накладывает ограничение на операции, в которые вовлечены различные типы данных.  
**Явные и неявные преобразования**  
Когда в одной операции вовлечены данные разных типов, не всегда необходимо использовать операцию преобразования типов. Некоторые виды преобразований выполняются неявно, автоматически.  
**Автоматические преобразования**

  
Стрелками на рисунке показано, какие преобразования типов могут выполняться автоматически. Пунктирными стрелками показаны автоматические преобразования с потерей точности.  
Автоматически без каких-либо проблем производятся расширяющие преобразования (widening) - они расширяют представление объекта в памяти. Например:  
byte b = 7;  
int d = b; // преобразование от byte к int  
В данном случае значение типа byte, которое занимает в памяти 1 байт, расширяется до типа int, которое занимает 4 байта.  
Расширяющие автоматические преобразования представлены следующими цепочками:  
byte -> short -> int -> long  
int -> double  
short -> float -> double  
char -> int  
**Автоматические преобразования с потерей точности**  
Некоторые преобразования могут производиться автоматически между типами данных одинаковой разрядности или даже от типа данных с большей разрядностью к типу с меньшей разрядностью. Это следующие цепочки преобразований:

int -> float,  
long -> float и long -> double. Они производятся без ошибок, но при преобразовании мы можем столкнуться с потерей информации.   
**Явные преобразования**  
Во всех остальных преобразованиях примитивных типов явным образом применяется операция преобразования типов. Обычно это сужающие преобразования (narrowing) от типа с большей разрядностью к типу с меньшей разрядностью.  
**Потеря данных при преобразовании**  
При применении явных преобразований мы можем столкнуться с потерей данных.   
int a = 258;  
byte b = (byte) a;  
System.out.println(b); // 2  
Результатом будет число 2. В данном случае число 258 вне диапазона для типа byte (от -128 до 127), поэтому произойдет усечение значения.   
**Усечение рациональных чисел до целых**  
При преобразовании значений с плавающей точкой к целочисленным значениям, происходит усечение дробной части:  
double a = 56.9898;  
int b = (int)a;  
Здесь значение числа b будет равно 56, несмотря на то, что число 57 было бы ближе к 56.9898. Чтобы избежать подобныхказусов, надо применять функцию округления, которая есть в математической библиотеке Java:  
double a = 56.9898;  
int b = (int)Math.round(a);  
**Преобразования при операциях**  
Нередки ситуации, когда приходится применять различные операции, например, сложение и произведение, над значениями разных типов. Здесь также действуют некоторые правила:

если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double

если предыдущее условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float  
если предыдущие условия не соблюдены, один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long  
иначе все операнды операции преобразуются к типу int  
Примеры преобразований:  
int a = 3;  
double b = 4.6;  
double c = a+b;  
Так как в операции участвует значение типа double, то и другое значение приводится к типу double и сумма двух значений a+b будет представлять тип double.  
Другой пример:  
byte a = 3;  
short b = 4;  
byte c = (byte)(a+b);  
Две переменных типа byte и short (не double, float или long), поэтому при сложении они преобразуются к типу int, и их сумма a+b представляет значение типа int. Поэтому если затем мы присваиваем эту сумму переменной типа byte, то нам опять надо сделать преобразование типов к byte.

**6. Условные конструкции.**

Одним из фундаментальных элементов многих языков программирования являются условные конструкции. Данные конструкции позволяют направить работу программы по одному из путей в зависимости от определенных условий. В языке Java используются следующие условные конструкции: if..else и switch..case  
Условный оператор if

Ели логическое выражение в скобках правдиво, то выполняется , блок кода в фигурных скобках {} после if. Если логическое выражение принимает значение false, то ничего не происходит.

if (a == b) {

System.out.println("a и b равны!"); //Если a равно b - выводим сообщение

}

Условный оператор if-else

Конструкция if-else отличается от предыдущей тем, что если логическое выражение в круглых скобках принимает значение false, то выполняется блок кода, находящийся в фигурных скобках после ключевого слова else

if (a == b) {

System.out.println("a и b равны!");//Если a равно b - выводим сообщение

}

else{

System.out.println("a и b не равны!"); //Если a не равно b - выводим сообщение

}

Если блоки кода  if, else  содержат лишь одну строку, то можно обойтись без использования фигурных скобок.  
**Конструкция switch**  
Конструкция switch/case аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:  
switch (выражение) {

case x:

// фрагмент кода

break;

case y:

// фрагмент кода

break;

default:

// фрагмент кода

}  
После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно срав нивается со значениями, помещенными после операторов сase. И если совпадение найдено, то будет выполняет соответствующий блок сase. В конце блока сase ставится оператор break, чтобы избежать выполнения других блоков. Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default. Также мы можем определить одно действие сразу для нескольких блоков case подряд.  
**Тернарная операция**  
Тернарная операция имеет следующий синтаксис: [первый операнд (условие)] ? [второй операнд] : [третий операнд]. Таким образом, в этой операции участвуют сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:  
int x=3, y=2;  
int z = x<y? (x+y) : (x-y);  
System.out.println(z);  
Здесь результатом тернарной операции является переменная z. Сначала проверяется условие x<y. И если оно соблюдается,то z будет равно второму операнду (x+y), иначе z будет равно третьему операнду.

**7. Операторы цикла. Операторы break и continue.**

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз. В языке Java есть следующие виды циклов: for; while; do...while;  
**Цикл for**  
Цикл for имеет следующее формальное определение:  
for ([инициализация счетчика]; [условие]; [изменение счетчика])  
{  
// действия  
}  
Рассмотрим стандартный цикл for:  
for (int i = 1; i < 9; i++){  
System.out.printf("Квадрат числа %d равен %d \n", i, i \* i);  
}  
Первая часть объявления цикла - int i = 1 создает и инициализирует счетчик i (Счетчик необязательно должен представлять тип int. Это может быть и любой другой числовой тип, например, float.). Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной.  
Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9. И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--. В итоге блок цикла сработает 8 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.  
Нам необязательно указывать все условия при объявлении цикла, можно опустить ряд блоков.  
int i = 1;  
for (; i<9;){  
System.out.printf("Квадрат числа %d равен %d \n", i, i \* i);  
i++;  
}  
Этот пример эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for. Цикл for может определять сразу несколько переменных и управлять ими.  
**Цикл do**  
Цикл do сначала выполняет код цикла, а потом проверяет условие в инструкции while. И пока это условие истинно, цикл повторяется. Например:  
int j = 7;  
do{  
System.out.println(j);  
j--;  
}  
while (j > 0);  
В данном случае код цикла сработает 7 раз, пока j не окажется равным нулю. Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно.   
**Цикл while**  
Цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:  
int j = 6;  
while (j > 0){  
System.out.println(j);  
j--;  
}  
**Операторы continue и break**  
Оператор break осуществляет выход из блока фигурных скобок { } оператора цикла или оператора switch и передает управление следующему оператору в программе. Если используются вложенные операторы, то оператор break осуществляет выход из самого внутреннего оператора. Общая форма оператора break следующая

break;

Оператор break можно использовать во всех [операторах цикла](https://www.bestprog.net/ru/2018/04/22/loops-for-while-do-while-examples-of-use-solving-of-tasks-using-loops_ru/) и операторе [switch](https://www.bestprog.net/ru/2017/05/11/оператор-выбора-switch-2/). Другими словами, оператор break не может быть использован за пределами цикла или оператора switch.

Оператор continue применяется в [циклах](https://www.bestprog.net/ru/2018/04/22/loops-for-while-do-while-examples-of-use-solving-of-tasks-using-loops_ru/). Оператор continue обеспечивает переход к следующей итерации в цикле. Общая форма оператора continue имеет вид:

continue;

Чаще всего оператор continue вызывается при достижении некоторого условия в теле цикла.

Оператор continue можно применять только в циклах. Другими словами, оператор continue нельзя применять за пределами цикла.

**8. Массивы. Объявление массивов. Инициализация массивов. Одномерные массивы. Многомерные массивы.**

**Массивы**

Массив представляет набор однотипных значений. Объявление массива похоже на объявление обычной переменной, которая хранит одиночное значение, причем есть два способа объявления массива:  
тип\_данных название\_массива[];  
тип\_данных[] название\_массива;  
Например, определим массив чисел:  
int nums[];  
int[] nums2;  
После объявления массива мы можем инициализовать его:  
int nums[];  
nums = new int[4]; // массив из 4 чисел  
Создание массива производится с помощью следующей конструкции:

new тип\_данных[количество\_элементов], где new - ключевое слово, выделяющее память для указанного в скобках количества элементов. Например, nums = new int[4]; - в этом  
выражении создается массив из четырех элементов int, и каждый элемент будет иметь значение по умолчанию - число 0.  
Также можно сразу при объявлении массива инициализировать его:  
int nums[] = new int[4]; // массив из 4 чисел  
При подобной инициализации все элементы массива имеют значение по умолчанию. Для числовых типов (в том числе для типа char) это число 0, для типа boolean это значение false, а для остальных объектов это значение null. Поскольку для типа int значением по умолчанию является число 0, то выше определенный массив nums будет состоять из четырех нулей.  
Однако также можно задать конкретные значения для элементов массива при его создании:  
int[] nums = new int[] { 1, 2, 3, 5 }; // эти два способа равноценны  
int[] nums2 = { 1, 2, 3, 5 };  
Стоит отметить, что в этом случае в квадратных скобках не указывается размер массива, так как он вычисляется по количеству элементов в фигурных скобках.  
После создания массива мы можем обратиться к любому его элементу по индексу, который передается в квадратных скобках после названия переменной массива.  
int[] nums = new int[4]; // устанавливаем значения элементов массива  
nums[0] = 1;  
nums[1] = 2;  
nums[2] = 4;  
nums[3] = 100;  
System.out.println(nums[2]); // 4 // получаем значение третьего элемента массива  
Индексация элементов массива начинается с 0, поэтому в данном случае, чтобы обратиться к четвертому элементу в массиве, нам надо использовать выражение nums[3].  
И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу. Если мы так попытаемся сделать, то мы получим ошибку.

**Многомерные массивы**

Одномерные массивы можно представить как цепочку или строку однотипных значений. Но кроме одномерных массивов  
также бывают и многомерные. Наиболее известный многомерный массив - таблица, представляющая двухмерный массив:  
  
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };  
int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };  
Визуально оба массива можно представить следующим образом:  
Одномерный массив nums1  
0 1 2 3 4 5  
Двухмерный массив nums2  
0 1 2  
3 4 5  
Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Его также можно было создать следующим образом: int[][] nums2 = new int[2][3];. Количество квадратных скобок указывает на размерность массива. А числа в скобках - на количество строк и столбцов. И также, используя индексы, мы можем использовать элементы массива в программе:  
nums2[1][0]=44;// установим элемент первого столбца второй строки  
System.out.println(nums2[1][0]);  
Объявление трехмерного массива могло бы выглядеть так:  
int[][][] nums3 = new int[2][3][4];

**Зубчатый массив**  
Многомерные массивы могут быть также представлены как "зубчатые массивы". В вышеприведенном примере двухмерный массив имел 3 строчки и три столбца, поэтому у нас получалась ровная таблица. Но мы можем каждому элементу в двух-  
мерном массиве присвоить отдельный массив с различным количеством элементов:  
int[][] nums = new int[3][];  
nums[0] = new int[2];  
nums[1] = new int[3];  
nums[2] = new int[5];

**9. Методы. Параметры методов. Результат метода. Перегрузка методов.**

**Методы**  
Если переменные и константы хранят некоторые значения, то методы содержат набор операторов, которые выполняют определенные действия.Общее определение методов выглядит следующим образом:  
[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры]){  
// тело метода  
}  
Модификаторы и параметры необязательны. Создадим несколько методов:

void hello(){  
 System.out.println("Hello");  
}  
void welcome(){  
 System.out.println("Welcome to Java 17");  
}  
Здесь определены два дополнительных метода: hello и welcome, каждый из которых выводит некоторую строку на консоль. Методы определяются внутри класса . По умолчанию в программе Java выполняется только метод main и все его содержимое. Поэтому, если мы хотим, чтобы другие методы тоже выполнялись, их надо вызвать в методе main. Вызов метода осуществляется в форме:  
имя\_метода(аргументы);  
После имени метода указываются скобки, в которых перечисляются аргументы - значения для параметров метода.  
Например, определим и выполним несколько методов:  
public class Program{  
public static void main (String[] args){  
hello();  
welcome();  
welcome();  
}  
static void hello(){  
System.out.println("Hello");  
}  
static void welcome(){  
System.out.println("Welcome to Java 17");  
}  
}  
В методе main вызывается один раз метод hello и два раза метод welcome. В этом и заключается одно из преимуществ методов: мы можем вынести некоторые общие действия в отдельный метод и затем вызывать многократно их в различных местах программы. Также следует отметить, что чтобы вызвать в методе main другие методы, которые определены в одном классе с методом main, они должны иметь модификатор static.  
**Параметры методов**  
С помощью параметров мы можем передать в методы различные данные, которые будут использоваться для вычислений. Например:  
static void sum(int x, int y){  
int z = x + y;  
System.out.println(z);  
}  
Данный метод принимает два параметра - два числа, складывает их и выводит их сумму на консоль. А при вызове этого метода в программе нам необходимо передать на место параметров значения, которые соответствуют типу параметра. Поскольку метод sum принимает два значения типа int, то на место параметров надо передать два значения типа int. Это могут быть и числовые литералы, и переменные типов данных, которые представляют тип int или могут быть автоматически преобразованы в тип int. Значения, которые передаются на место параметров, еще называются аргументами. Значения передаются параметрам по позиции, то есть первый аргумент первому параметру, второй аргумент - второму параметру и так далее.  
**Параметры переменной длины**  
Метод может принимать параметры переменной длины одного типа. Например, нам надо передать в метод набор чисел и вычислить их сумму, но мы точно не знаем, сколько именно чисел будет передано - 3, 4, 5 или больше. Параметры переменнойдлины позволяют решить эту задачу:  
public class Program{  
public static void main (String[] args){  
sum(1, 2, 3); // 6  
sum(1, 2, 3, 4, 5); // 15  
sum(); // 0  
}  
static void sum(int ...nums){  
int result =0;  
for(int n: nums)  
result += n;  
System.out.println(result);  
}  
}  
Троеточие перед названием параметра int ...nums указывает на то, что он будет необязательным и будет представлять массив. Мы можем передать в метод sum одно число, несколько чисел, а можем вообще не передавать никаких параметров. Причем, если мы хотим передать несколько параметров, то необязательный параметр должен указываться в конце.  
**Оператор return**  
Методы могут возвращать некоторое значение. Для этого применяется оператор return.  
return возвращаемое\_значение;  
После оператора return указывается возвращаемое значение, которое является результатом метода. Это может быть литеральное значение, значение переменной или какого-то сложного выражения. Если в качестве возвращаемого типа для метода определен любой другой, отличный от void, то метод обязательно должен использовать оператор return для возвращения значения. При этом возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции. И если функция возвращает значение типа int, то после оператора return стоит целочисленное значение, которое является объектом типа int. Метод может использовать несколько вызовов оператора return для возвращения разных значений в зависимости от некоторых условий.

**Выход из метода**  
Оператор return применяется для возвращения значения из метода, и для выхода из метода. В подобном качестве оператор return применяется в методах, которые ничего не возвращают, то есть имеют тип void.  
**Перегрузка методов**  
В программе мы можем использовать методы с одним и тем же именем, но с разными типами и/или количеством параметров. Такой механизм называется перегрузкой методов. Стоит отметить, что на перегрузку методов влияют количество и типы параметров. Однако различие в типе возвращаемого значения для перегрузки не имеют никакого значения.

**10. Java ООП. Классы и объекты. Конструкторы. Ключевое слово this. Инициализаторы.**

**Классы и объекты**  
Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов. Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести  
следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Это есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса. Класс определяется с помощью ключевого слова сlass:  
class Person{  
}  
В данном случае класс называется Person. После названия класса идут фигурные скобки, между которыми помещается тело класса - то есть его поля и методы. Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.  
Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы.   
Как правило, классы определяются в разных файлах. Класс представляет новый тип, поэтому мы можем определять переменные, которые представляют данный тип.  
**Конструкторы**  
Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается   
Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения). Если необходимо, чтобы при создании объекта производилась какая-то логика, например, чтобы поля класса получали какие-то определенные значения, то можно определить в классе свои конструкторы. Например:  
class Person{  
String name; // имя  
int age; // возраст  
Person(){  
name = "Undefined";  
age = 18;  
}  
Person(String n){  
name = n;  
age = 18;  
}  
Person(String n, int a){  
name = n;  
age = a;  
}  
void displayInfo(){ System.out.printf("Name: %s \t Age: %d\n", name, age); }  
}

**Ключевое слово this**  
Ключевое слово this представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы. Как правило, применять this нужно в двух случаях:  
Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;  
Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора. Вызов this() может находиться только в первой строчке конструтора.

**Инициализаторы**  
Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Инициализатор выполняется до любого конструктора. То есть в инициализатор мы можем поместить код, общий для всех конструкторов:  
class Person{  
String name; // имя  
int age; // возраст  
/\*начало блока инициализатора\*/  
{  
name = "Undefined";  
age = 18;  
}  
/\*конец блока инициализатора\*/

**11. Модификаторы доступа и инкапсуляция.**

**Модификаторы доступа**  
Все члены класса в языке Java - поля и методы - имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.  
В Java используются следующие модификаторы доступа:  
public: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.  
private: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.  
protected: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах,даже если они находятся в других пакетах  
Модификатор по умолчанию. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.  
Рассмотрим модификаторы доступа на примере следующей программы:  
public class Program{  
public static void main(String[] args) {  
Person kate = new Person("Kate", 32, "Baker Street", "+12334567");  
kate.displayName(); // норм, метод public  
kate.displayAge(); // норм, метод имеет модификатор по умолчанию  
kate.displayPhone(); // норм, метод protected  
//kate.displayAddress(); // ! Ошибка, метод private  
System.out.println(kate.name); // норм, модификатор поля по умолчанию  
System.out.println(kate.address); // норм, модификатор поля public  
System.out.println(kate.age); // норм, модификатор поля protected  
//System.out.println(kate.phone); // ! Ошибка, модификатор поля private  
}  
}  
class Person{  
String name;  
protected int age;  
public String address;  
private String phone;  
public Person(String name, int age, String address, String phone){  
this.name = name;  
this.age = age;  
this.address = address;  
this.phone = phone;  
}  
public void displayName(){ System.out.printf("Name: %s \n", name); }  
void displayAge(){ System.out.printf("Age: %d \n", age); }  
private void displayAddress(){ System.out.printf("Address: %s \n", address); }  
protected void displayPhone(){ System.out.printf("Phone: %s \n", phone); }  
}  
**Инкапсуляция**  
Казалось бы, почему бы не объявить все переменные и методы с модификатором public, чтобы они были доступны в любой точке программы вне зависимости от пакета или класса? Возьмем, например, поле age, которое представляет возраст.Если другой класс имеет прямой доступ к этому полю, то есть вероятность, что в процессе работы программы ему будет передано некорректное значение, например, отрицательное число. Подобное изменение данных не является желательным. Либо же мы не хотим, чтобы некоторые данные были доступны напрямую, чтобы их можно было вывести на консоль или просто узнать их значение. В этой связи рекомендуется как можно больше ограничивать доступ к данным, чтобы защитить их от нежелательного доступа извне (как для получения значения, так и для его изменения). Использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное сокрытие данных внутри некоторой области видимости называется инкапсуляцией.  
Так, как правило, вместо непосредственного применения полей используют методы доступа.

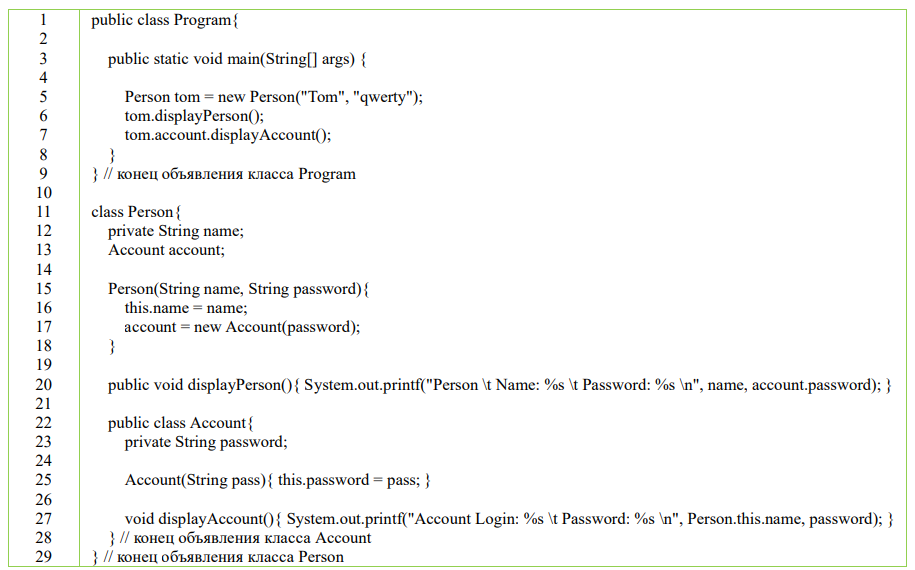
**12. Статические члены и модификатор static.**

Кроме обычных методов и полей класс может иметь статические поля, методы, константы и инициализаторы. Например, главный класс программы имеет метод main, который является статическим. Для объявления статических переменных, констант, методов и инициализаторов перед их объявлением указывается ключевое слово static.  
**Статические поля**  
При создании объектов класса для каждого объекта создается своя копия нестатических обычных полей. А статические поля являются общими для всего класса. Поэтому они могут использоваться без создания объектов класса. Например, создадим статическую переменную:  
public class Program{  
public static void main(String[] args) {  
Person tom = new Person();  
Person bob = new Person();  
tom.displayId(); // Id = 1  
bob.displayId(); // Id = 2  
System.out.println(Person.counter); // 3  
// изменяем Person.counter  
Person.counter = 8;  
Person sam = new Person();  
sam.displayId(); // Id = 8  
}  
}  
class Person{  
private int id;  
static int counter=1;  
Person(){ id = counter++; }  
public void displayId(){ System.out.printf("Id: %d \n", id); }  
}  
Класс Person содержит статическую переменную counter, которая увеличивается в конструкторе и ее значение присваивается переменной id. То есть при создании каждого нового объекта Person эта переменная будет увеличиваться, поэтому у каж-  
дого нового объекта Person значение поля id будет на 1 больше чем у предыдущего.  
**Статические константы**  
Также статическими бывают константы, которые являются общими для всего класса.  
В выражении: System.out.println("hello"); out как раз представляет статическую константу класса System. Поэтому обращение к ней идет без создания объекта класса  
System.  
**Статические инициализаторы**  
Статические инициализаторы предназначены для инициализации статических переменных, либо для выполнения таких действий, которые выполняются при создании самого первого объекта. Например, определим статический инициализатор:  
public class Program{  
public static void main(String[] args) {  
Person tom = new Person();  
Person bob = new Person();  
tom.displayId(); // Id = 105  
bob.displayId(); // Id = 106  
}  
}  
class Person{  
private int id;  
static int counter;  
static{  
counter = 105;  
System.out.println("Static initializer");  
}  
Person(){  
id=counter++;  
System.out.println("Constructor");  
}  
public void displayId(){ System.out.printf("Id: %d \n", id); }  
}  
Статический инициализатор определяется как обычный, только перед ним ставится ключевое слово static. В данном случае в статическом инициализаторе мы устанавливаем начальное значение статического поля counter и выводим на консольсообщение.  
Стоит учитывать, что вызов статического инициализатора производится после загрузки класса и фактически до создания  
самого первого объекта класса.  
**Статические методы**  
Статические методы также относятся ко всему классу в целом. Для обращения к статическому методу используется имя класса. При использовании статических методов надо учитывать ограничения: в статических методах мы можем вызывать только  
другие статические методы и использовать только статические переменные.  
Вообще методы определяются как статические если не затрагивают состояние объекта, то есть его нестатические поля и константы, и для вызова метода нет смысла создавать экземпляр класса. Например:  
public class Program{  
public static void main(String[] args) {  
System.out.println(Operation.sum(45, 23)); // 68  
System.out.println(Operation.subtract(45, 23)); // 22  
System.out.println(Operation.multiply(4, 23)); // 92  
}  
}  
class Operation{  
static int sum(int x, int y){ return x + y; }  
static int subtract(int x, int y){ return x - y; }  
static int multiply(int x, int y){ return x \* y; }  
}  
В данном случае для методов sum, subtract, multiply не имеет значения, какой именно экземпляр класса Operation используется. Эти методы работают только с параметрами, не затрагивая состояние класса. Поэтому их можно определить как статические.

**13. Внутренние и вложенные классы.**

*Внутренние и вложенные классы.*

Классы могут быть вложенными (nested), то есть могут быть определены внутри других классов. Частным случаем вложенных классов являются внутренние классы (inner class). Например, имеется класс Person, внутри которого определен класс Account:

Внутренний класс ведет себя как обычный класс за тем исключением, что его объекты могут быть созданы только внутри внешнего класса. Внутренний класс имеет доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private. Аналогично внешний класс имеет доступ ко всем членам внутреннего класса, в том числе к полям и методам с модификатором private. Ссылку на объект внешнего класса из внутреннего класса можно получить с помощью выражения Внешний\_класс.this, например, Person.this. Объекты внутренних классов могут быть созданы только в том классе, в котором внутренние классы определены. В других внешних классах объекты внутреннего класса создать нельзя. Еще одной особенностью внутренних классов является то, что их можно объявить внутри любого контекста, в том числе внутри метода и даже в цикле.

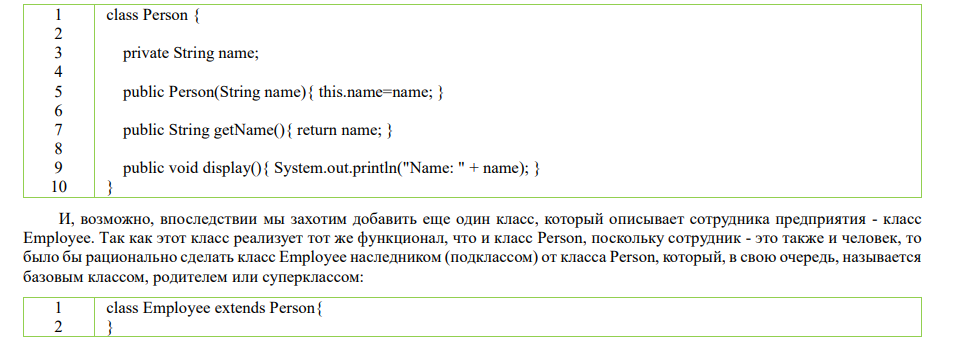
*Статические вложенные классы.*

Кроме внутренних классов также могут быть статические вложенные классы. Статические вложенные классы позволяют скрыть некоторую комплексную информацию внутри внешнего класса: Для случая, когда связь между объектом вложенного класса и объектом внешнего класса не нужна, можно сделать вложенный класс статическим(static). Из вложенного статического класса мы не имеем доступа к внешней не статической переменной внешнего класса.

**14. Наследование.**

*Наследование*

Одним из ключевых аспектов объектно-ориентированного программирования является наследование. С помощью наследования можно расширить функционал уже имеющихся классов за счет добавления нового функционала или изменения старого. Например, имеется следующий класс Person, описывающий отдельного человека:

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово extends, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

Если в базовом классе определены конструкторы, то в конструкторе производного классы необходимо вызвать один из конструкторов базового класса с помощью ключевого слова super. Например, класс Person имеет конструктор, который принимает один параметр. Поэтому в классе Employee в конструкторе нужно вызвать конструктор класса Person. После слова super в скобках идет перечисление передаваемых аргументов. Таким образом, установка имени сотрудника делегируется конструктору базового класса. При этом вызов конструктора базового класса должен идти в самом начале в конструкторе производного класса. Наследуемый класс имеет доступ ко всем методам и полям базового класса (даже если базовый класс находится в другом пакете) кроме тех, которые определены с модификатором private. При этом наследуемый класс также может добавлять свои поля и методы.

*Переопределение методов*

Производный класс может определять свои методы, а может переопределять методы, которые унаследованы от базового класса.

Перед переопределяемым методом указывается аннотация @Override. Данная аннотация в принципе необязательна. При переопределении метода он должен иметь уровень доступа не меньше, чем уровень доступа в базовом классе. Например, если в базовом классе метод имеет модификатор public, то и в производном классе метод должен иметь модификатор public.

*Запрет наследования.*

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова final.

*Динамическая диспетчеризация методов.*

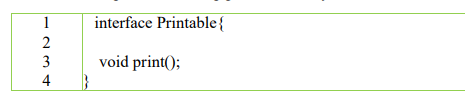
Наследование и возможность переопределения методов открывают нам большие возможности. Прежде всего мы можем передать переменной суперкласса ссылку на объект подкласса.При вызове переопределенного метода виртуальная машина динамически находит и вызывает именно ту версию метода, которая определена в подклассе. Данный процесс еще называется dynamic method lookup или динамический поиск метода или динамическая диспетчеризация методов.

**15. Иерархия наследования и преобразование типов.**

Аппарат наследования классов предусматривает возможности преобразования типов между суперклассом и подклассом. Преобразование типов в каком-то смысле является формальным. Сам объект при таком преобразовании не изменяется, преобразование относится только к типу ссылки на объект. Различаются два вида преобразований типов — upcasting и downcasting. Повышающее преобразование (upcasting) — это преобразование от типа порожденного класса (от подкласса) к базовому (суперклассу). Такое преобразование допустимо всегда. На него нет никаких ограничений и для его проведения не требуется применять никаких дополнительных синтаксических конструкций. Это связано с тем, что объект подкласса всегда в себе содержит как свою часть объект суперкласса. Понижающее преобразование (downcasting) — это преобразование от суперкласса к подклассу. Такое преобразование имеет ряд ограничений. Во-первых, оно может задаваться только явно при помощи операции преобразования типов. Во-вторых, объект, подвергаемый преобразованию, реально должен быть того класса, к которому он преобразуется. Если это не так, то возникает исключение ClassCastException в процессе выполнения программы. Допустим, у нас есть следующая иерархия классов:

**16. Интерфейсы. Интерфейсы в механизме обратного вызова.**

**Интерфейсы.**

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, мы можем наследовать только от одного класса, в отличие, например, от языка С++, где имеется множественное наследование.  
 В языке Java подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.   
 Чтобы определить интерфейс, используется ключевое слово interface. Например:  
  
 Данный интерфейс называется Printable. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.  
 Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.  
 Чтобы класс применил интерфейс, надо использовать ключевое слово implements.  
 В данном случае класс Book реализует интерфейс Printable. При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print. Потом в методе main мы можем создать объект класса Book и вызвать его метод print. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут реализовать эти методы.  
 Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости.  
Класс Book и класс Journal связаны тем, что они реализуют интерфейс Printable. Поэтому мы динамически в программе можем создавать объекты Printable как экземпляры обоих классов:

**Интерфейсы в механизме обратного вызова.**

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова. Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях. То есть одни действия вызываются другими действиями. Стандартный пример - нажатие на кнопку. Когда мы нажимаем на кнопку, мы производим действие, но в ответ на это нажатие запускаются другие действия. Например, нажатие на значок принтера запускает печать документа на принтере и т.д.

**17. Клас Object. Методы класса Object.**

**Класс Object.**

Фактически все классы наследуются от класса Object. Все остальные классы, даже те, которые мы добавляем в свой проект, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object. Рассмотрим некоторые из них

**Методы класса Object.**

1) Метод getClass позволяет получить тип данного объекта:

**Person Tom = new Person(“Tom”);**

**Sysyem.out.println(tom.getClass()); //class Person**

Полученное значение (в данном случае Person@7960847b) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод toString() нередко переопределяют.

2) Метод toString служит для получения представления данного объекта в виде строки. По умолчанию возвращает имя\_класса@hashcode в 16-ричной системе.

Person tom = new Person (“Tom”);

System.out.println(tom.toString()); //выведет что-то вроде Person@7960847b

.  
3) Метод hashCode - это целое число, генерируемое на основе конкретного объекта. Значение по умолчанию - целочисленный адрес в памяти. Его можно рассматривать как шифр с уникальным значением или его хэш-код. По данному числу, например, можно сравнивать объекты.

**Person Tom = new Person(“Tom”);**

**Sysyem.out.println(tom.hashCode ()); // напишите что-то наподобие** 2036368507

4) Метод equals сравнивает два объекта на равенство. Операция == не рекомендуется для сравнения объектов в Java. Дело в том, что при сравнении объектов, операция == вернет true лишь в одном случае — когда ссылки указывают на один и тот же объект. В данном случае не учитывается содержимое переменных класса.

Person tom = new Person("Tom");

Person bob = new Person("Bob");

System.out.println(tom.equals(bob)); // false

Person tom2 = new Person("Tom");

System.out.println(tom.equals(tom2)); // true

**18. Перечисления enum.**

При программировании часто необходимо использовать ограниченное множество допустимых значений для некоторого типа данных. Так, например, день недели может иметь 7 разных значений, месяцев в году не более 12, и всего 4 времени года. Для решения подобных задач во многих языках программирования предусмотрен специальный тип данных - перечисление (enum).

Перечисление объявляется так: после ключевого слова enum указывается название перечисления и в фигурных скобках - набор идентификаторов, которые определяют константы, являющиеся возможными значениями переменной, относящейся к типу перечисления. Пример объявления перечисления: enum MyColor {RED, GREEN, BLUE}.  
 В данном случае объявляется перечисление MyColor. Это - тип данных. Поэтому мы можем объявить переменную данного типа. Например, такую: MyColor clr;

В качестве значения переменной clr можно присвоить одну из констант - RED, GREEN или BLUE, указанных в объявлении перечисления MyColor. Причем указывается не просто название константы, а еще и название перечисления: MyColor.RED, MyColor.GREEN или MyColor.BLUE.

Пример:

public class Program{

public static void main(String[] args) {

Day current = Day.MONDAY;

System.out.println(current); // MONDAY

}

} enum Day{ MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

**19.Обобщения(Generics). Обобщенные интерфейсы. Обобщенные методы.**

**Обобщение** – это механизм построения программного кода для некоторого типа с произвольным именем с целью его дальнейшего конвертирования (преобразования) в другой конкретный ссылочный тип. Реализацию конвертирования из обобщенного типа в другой (конкретный) осуществляет компилятор.

Обобщения могут быть применены к классам, интерфейсам или методам. Если класс, интерфейс или метод оперирует некоторым обобщенным типом T, то этот класс (интерфейс, метод) называется обобщенным. Тип, который получает обобщенный класс в качестве параметра, называется параметризованным типом. Имя параметризованного типа можно задавать любым (T, Type, TT и т.д.).

**Преимущества применения обобщений**

Использование обобщений в языке Java дает следующие преимущества:

* обеспечивается компактность программного кода;
* благодаря обобщениям все операции приведения типов выполняются автоматически и неявно. Повторно используемый код обрабатывается более легко и безопасно;
* обобщения обеспечивают типовую безопасность типов в отличие от использования ссылок на тип Object.

**Общая форма объявления обобщенного класса и объявление ссылки на обобщенный класс**

**class ClassName<Type1, Type2, ..., TypeN> {**

**// ...**

**}**

При использовании обобщенных классов параметр типа должен быть только ссылочного типа. При использовании обобщенного класса запрещается использовать базовые типы (int, char, double и т.д.) в качестве аргумента типа.

**Пример обобщенного класса, который реализует метод поиска элемента в двумерной матрице**

Объявляется обобщенный класс GenericMatrix<Type>, в котором реализован метод SearchKey(), который вычисляет количество вхождений заданного элемента key в матрице.

Метод SearchKey() получает следующие параметры:

M – исходная матрица обобщенного типа Type[][], в которой осуществляется поиск ключа key;

m, n – размерность матрицы, количество строк и столбцов соответственно;

key – ключ (элемент), количество вхождений которого в матрицу M нужно вычислить.

*// Обобщенный класс, оперирующий типом Type*

**class GenericMatrix<Type> {**

// Метод, который осуществляет подсчет количества элементов key в двумерной матрице M.

**public int SearchKey(Type[][] M, int m, int n, Type key) {**

**int count = 0;**

**for (int i=0; i<m; i++)**

**for (int j=0; j<n; j++)**

**if (key==M[i][j])**

**count++;**

**return count;**

**}**

**}**

**20.Ограничения обобщений. Наследование и обобщения.**

Когда мы указываем универсальный параметр у обобщений, то по умолчанию он может представлять любой тип. Однако иногда необходимо, чтобы параметр соответствовал только некоторому ограниченному набору типов. В этом случае применяются **ограничения**, которые позволяют указать базовый класс, которому должен соответствовать параметр.

Для установки ограничения после универсального параметра ставится слово extends, после которого указывается базовый класс ограничения:

**class Account{ }**

**class Transaction<T extends Account>{ }**

К примеру, в данном случае для параметра T в Transaction ограничением является класс Account. То есть на место параметра T мы можем передать либо класс Account, либо один из его классов-наследников.

Обобщенные классы могут участвовать в иерархии **наследования:** могут наследоваться от других, либо выполнять роль базовых классов. Рассмотрим различные ситуации.

При наследовании от обобщенного класса класс-наследник должен передавать данные о типе в конструкции базового класса:

**class Account<T>**

**{**

**private T \_id;**

**T getId(){return \_id;}**

**Account(T id)**

**{**

**\_id = id;**

**}**

**}**

**class DepositAccount<T> extends Account<T>{**

**DepositAccount(T id){**

**super(id);**

**}**

**}**

**21.Records.**

**Records** представляют классы, которые предназначены для создания контейнеров неизменяемых данных. Кроме того, records позволяют упростить разработку, сократив объем кода.

Для определения классов record применяется ключевое слово record, после которого идет название и далее в круглых скобках список полей record:

**record название (поле1, поле2,...полеN){**

**// тело record**

**}**

Records определяются с помощью ключевого слова **record**, за которым следует название записи. Дальше идет список полей записи. То есть, например, определяется два поля - *name* и *age*. Причем по умолчанию все они будут приватными и иметь модификатор **final**.

Также будет создаваться конструктор с двумя параметрами *name* и *age*. А каждого поля автоматически будет создаваться одноименный общедоступный метод для получения значения — это поля. Например, для поля *name* создается метод *name()*, который возвращает значение поля *name*.

**Ограничения records**

Следует учитывать, что мы не можем наследовать запись **record** от других классов. Также нельзя наследовать классы от records. Однако классы record могут реализовать интерфейсы. Кроме того, классы record не могут быть абстрактными.

В **record** нельзя явным образом определять нестатические поля и инициализаторы.

Но можно определять статические переменные и инициализаторы, также как статические и нестатические методы:

**record Person(String name, int age){**

**static int minAge;**

**static{**

**minAge = 18;**

**System.out.println("Static initializer");**

**}**

**}**

**22. Введение в обработку исключений. Оператор throws. Создание своих классов исключений.**

Нередко в процессе выполнения программы могут возникать ошибки, при том необязательно по вине разработчика. Некоторые из них трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Так, например, может неожиданно оборваться сетевое подключение при передаче файла. Подобные ситуации называются исключениями.

В языке Java предусмотрены специальные средства для обработки подобных ситуаций. Одним из таких средств является конструкция try…catch…finally. При возникновении исключения в блоке try управление переходит в блок catch, который может обработать данное исключение. Если такого блока не найдено, то пользователю отображается сообщение о необработанном исключении, а дальнейшее выполнение программы останавливается. И чтобы подобной остановки не произошло, и надо использовать блок try..catch **.**При использовании блока try…catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции сatch. Выражение catch имеет следующий синтаксис: catch (тип\_исключения имя\_переменной). В данном случае объявляется переменная ex, которая имеет тип Exception. Но если возникшее исключение не является исключением типа, указанного в инструкции сatch, то оно не обрабатывается, а программа просто зависает или выбрасывает сообщение об ошибке.Но так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения. Обработка же исключения в данном случае сводится к выводу на консоль стека трассировки ошибки с помощью метода printStackTrace(), определенного в классе Exception.После завершения выполнения блока catch программа продолжает свою работу, выполняя все остальные инструкции после блока catch. Конструкция try..catch также может иметь блок finally. Однако этот блок необязательный, и его можно при обработке исключений опускать. Блок finallyвыполняется в любом случае, возникло ли исключение в блоке try или нет:

try{

    int[] numbers = new int[3];

    numbers[4]=45;

    System.out.println(numbers[4]);

}

catch(Exception ex){

    ex.printStackTrace();

}

finally{

    System.out.println("Блок finally");

}

System.out.println("Программа завершена");

**Оператор throw**

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения. Например, в нашей программе происходит ввод числа, и мы хотим, чтобы, если число больше 30, то возникало исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package firstapp;    import java.util.Scanner;  public class FirstApp {        public static void main(String[] args) {            try{              Scanner in = new Scanner(System.in);              int x = in.nextInt();              if(x>=30){                 throw new Exception("Число х должно быть меньше 30");             }          }          catch(Exception ex){                System.out.println(ex.getMessage());          }          System.out.println("Программа завершена");      }  } |

Здесь для создания объекта исключения используется конструктор класса Exception, в который передается сообщение об исключении. И если число х окажется больше 29, то будет выброшено исключение и управление перейдет к блоку catch.

В блоке catch мы можем получить сообщение об исключении с помощью метода getMessage().

**23. Типы коллекций. Интерфейс Collection. Класс ArrayList и интерфейс List. Очереди и класс ArrayDeque. Класс LinkedList.**

Коллекции — это хранилища поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных структур данных, поддерживающих три основные операции:

• добавление нового элемента в коллекцию;

• удаление элемента из коллекции;

• изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: заменить, просмотреть элементы, подсчитать их количество и др. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других. Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал. Среди этих интерфейсов можно выделить следующие:

* Collection: базовый интерфейс для всех коллекций и других интерфейсов коллекций
* Queue: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
* Deque: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
* List: наследует интерфейс Collection и представляет функциональность простых списков
* Set: расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
* SortedSet: расширяет интерфейс Set для создания сортированных коллекций
* NavigableSet: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
* Map: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

**Интерфейс Collection**

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

public interface Collection extends Iterable{

// определения методов

}

Здесь Е обозначает тип объектов, которые будет содержать коллекция

Среди методов интерфейса Collection можно выделить следующие:

* boolean add (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном – false
* boolean addAll (Collection col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном – false
* void clear (): удаляет все элементы из коллекции
* boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
* boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false
* Iterator iterator (): возвращает объект Iterator для обхода элементов коллекции
* boolean remove (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
* boolean removeAll (Collection col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
* boolean retainAll (Collection col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
* int size (): возвращает число элементов в коллекции
* Object[] toArray (): возвращает массив, содержащий все элементы коллекции Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же. Единообразный интерфейс упрощает понимание и работу с различными типами коллекций. Так, добавление элемента будет производиться с помощью метода add, который принимает добавляемый элемент в качестве параметра. Для удаления вызывается метод remove(). Метод clear будет очищать коллекцию, а метод size возвращать количество элементов в коллекции.

**Класс ArrayList и интерфейс List**

Для создания простых списков применяется интерфейс List, который расширяет функциональность интерфейса Collection. Некоторые наиболее часто используемые методы интерфейса List:

* void add(int index, E obj): добавляет в список по индексу index объект obj
* boolean addAll(int index, Collection col): добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе возвращается false
* E get(int index): возвращает объект из списка по индексу index
* int indexOf(Object obj): возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1
* int lastIndexOf(Object obj): возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1
* ListIterator listIterator (): возвращает объект ListIterator для обхода элементов списка
* static List of(элементы): создает из набора элементов объект List
* E remove(int index): удаляет объект из списка по индексу index, возвращая при этом удаленный объект
* E set(int index, E obj): присваивает значение объекта obj элементу, который находится по индексу index  
  void sort(Comparator comp): сортирует список с помощью компаратора comp
* List subList(int start, int end): получает набор элементов, которые находятся в списке между индексами start и end

По умолчанию в Java есть встроенная реализация этого интерфейса - класс ArrayList. Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано. ArrayList имеет следующие конструкторы:

* ArrayList(): создает пустой список
* ArrayList(Collection col): создает список, в который добавляются все элементы коллекции col.
* ArrayList (int capacity): создает список, который имеет начальную емкость capacity

**Очереди и класс ArrayDeque**

Очереди представляют структуру данных, работающую по принципу FIFO (first in - first out). То есть чем раньше был добавлен элемент в коллекцию, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди. Однако бывают и двунаправленные - то есть такие, в которых мы можем добавить элемент не только в начала, но и в конец. И соответственно удалить элемент не только из конца, но и из начала.

В Java очереди представлены рядом классов. Одни из низ - класс ArrayDeque. Этот класс представляет обобщенную двунаправленную очередь, наследуя функционал от класса AbstractCollection и применяя интерфейс Deque. Он не добавляет свои методы. Класс ArrayDeque создает динамический массив, не имеющий ограничений по емкости. (Интерфейс Deque поддерживает реализации с ограниченной емкостью, но не налагает на ее величину никаких ограничений.) Этот класс не позволяет хранить элементы со значением null.

**Класс LinkedList.**

Обобщенный класс LinkedList представляет структуру данных в виде связанного списка. Он наследуется от класса AbstractSequentialList и реализует интерфейсы List, Dequeue и Queue. То есть он соединяет функциональность работы со списком и функциональность очереди

24.Интерфейс Set и класс HashSet. SortedSet, NavigableSet, TreeSet.

Интерфейс Set расширяет интерфейс Collection и представляет набор уникальных элементов (не допускающих дублирования). Set не добавляет новых методов, только вносит изменения унаследованные. В частности, метод add() добавляет элемент в коллекцию и возвращает true, если в коллекции еще нет такого элемента.

Обобщенный класс HashSet представляет хеш-таблицу. Он наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с заданным ключом. Преобразование ключа в хеш-код выполняется автоматически.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

* HashSet(): создает пустое множество
* HashSet(Collection col): создает хеш-таблицу, в которую добавляет все элементы коллекции col
* HashSet(int capacity): параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16
* HashSet(int capacity, float koef): параметр koef 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.

Интерфейс SortedSet предназначен для создания коллекций, которые хранят элементы в отсортированном виде (сортировка по возрастанию). SortedSet расширяет интерфейс Set, поэтому такая коллекция опять же хранит только уникальные значения.

Интерфейс NavigableSet расширяет интерфейс SortedSet и позволяет извлекать элементы на основании их значений.

Обобщенный класс TreeSet представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet, а следовательно, и интерфейс SortedSet. Время доступа и извлечения элементов достаточно мало, благодаря чему класс TreeSet оказывается отличным выбором для хранения больших объемов отсортированных данных, которые должны быть быстро найдены.

В классе TreeSet определены следующие конструкторы:

* TreeSet(): создает пустое дерево
* TreeSet(Collection col): создает дерево, в которое добавляет все элементы коллекции col
* TreeSet(SortedSet set): создает дерево, в которое добавляет все элементы сортированного набора set
* TreeSet(Comparator comparator): создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

**25.Интерфейсы Comparable и Comporator. Сортировка. Итераторы**

Для того, чтобы объекты Person можно было сравнить и сортировать, мы должны применять интерфейс Comparable<E>.

При применении интерфейса он типизируется текущим классом. Применим его к классу Person:

class Person implements Comparable<Person>{

private String name;

Person(String name){ this.name = name; }

String getName(){return name;}

public int compareTo(Person p){ return name.compareTo(p.getName()); }

}

Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с

объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator.

Интерфейс Comparator содержит ряд методов, ключевым из которых является метод compare():

public interface Comparator<E> {

int compare(T a, T b);

// остальные методы

}

Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе

- наоборот. А если метод возвращает ноль, то объекты равны

Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Он возвращает итератор - то есть

объект, реализующий интерфейс Iterator.

Интерфейс Iterator имеет следующее определение:

public interface Iterator <E>{

E next();

boolean hasNext();

void remove();

}

Реализация интерфейса предполагает, что с помощью вызова метода next() можно получить следующий элемент. С помощью метода hasNext() можно узнать, есть ли следующий элемент, и не достигнут ли конец коллекции. И если элементы еще

имеются, то hasNext() вернет значение true. Метод hasNext() следует вызывать перед методом next(), так как при достижении

конца коллекции метод next() выбрасывает исключение NoSuchElementException. Метод remove() удаляет текущий элемент,

который был получен последним вызовом next().

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор - интерфейс ListIterator. Данный итератор используется классами, реализующими интерфейс List, то есть классами

LinkedList, ArrayList и др.

**26.Интерфейс Map и класс HashMap. Интерфейсы SortedMap и NavigableMap. Класс TreeMap.**

Интерфейс Map представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта. Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map НЕ расширяет интерфейс Collection.

Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов Map.Entry.

Обобщенный интерфейс Map.Entry<K, V> представляет объект с ключом типа K и значением типа V и определяет следующие методы:

boolean equals(Object obj): возвращает true, если объект obj, представляющий интерфейс Map.Entry, идентичен текущему

K getKey(): возвращает ключ объекта отображения

V getValue(): возвращает значение объекта отображения

V setValue(V v): устанавливает для текущего объекта значение v

int hashCode(): возвращает хеш-код данного объекта

Базовым классом для всех отображений является абстрактный класс AbstractMap, который реализует большую часть методов интерфейса Map. Наиболее распространенным классом отображений является HashMap, который реализует интерфейс Map и наследуется от класса AbstractMap.

Пример использования класса:

import java.util.\*;

public class Program{

public static void main(String[] args) {

Map<Integer, String> states = new HashMap<Integer, String>();

states.put(1, "Germany");

states.put(2, "Spain");

states.put(4, "France");

states.put(3, "Italy");

// получим объект по ключу 2

String first = states.get(2);

System.out.println(first);

// получим весь набор ключей

Set<Integer> keys = states.keySet();

// получить набор всех значений

Collection<String> values = states.values();

//заменить элемент

states.replace(1, "Poland");

// удаление элемента по ключу 2

states.remove(2);

// перебор элементов

for(Map.Entry<Integer, String> item : states.entrySet()){

System.out.printf("Key: %d Value: %s \n", item.getKey(), item.getValue());

}

Map<String, Person> people = new HashMap<String, Person>();

people.put("1240i54", new Person("Tom"));

people.put("1564i55", new Person("Bill"));

people.put("4540i56", new Person("Nick"));

for(Map.Entry<String, Person> item : people.entrySet()){

System.out.printf("Key: %s Value: %s \n", item.getKey(), item.getValue().getName());

}

}

}

class Person{

private String name;

public Person(String value){ name=value; }

String getName(){return name;}

}

**27.Работа со строками. StringBuffer и StringBuilder. Регулярные выражения.**

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы StringBuffer и StringBuilder. По сути они напоминают расширяемую строку, которую можно изменять без ущерба для производительности. Эти классы похожи, практически двойники. Они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются. Единственное их различие состоит в том, что класс StringBuffer синхронизированный и потокобезопасный. То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который не потокобезопасный, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.  
StringBuffer определяет четыре конструктора:  
StringBuffer()

StringBuffer(int capacity)

StringBuffer(String str)

StringBuffer(CharSequence chars)

Аналогичные конструкторы определяет StringBuilder:

StringBuilder()

StringBuilder(int capacity)

StringBuilder(String str)

StringBuilder(CharSequence chars)

Рассмотрим работу этих классов на примере функциональности StringBuffer. При всех операциях со строками StringBuffer/StringBuilder перераспределяет выделенную память. И чтобы избежать слишком частого перераспределения памяти, StringBuffer/StringBuilder заранее резервирует некоторую область памяти, которая может использоваться. Конструктор без параметров резервирует в памяти место для 16 символов. Если мы хотим, чтобы количество символов было иным, то мы можем применить второй конструктор, который в качестве параметра принимает количество символов. Третий и четвертый конструкторы обоих классов принимают строку и набор символов, при этом резервируя память для дополнительных 16 символов.

Регулярные выражения дают возможность осуществить сложный поиск или замену в строке. В языке Java использовать регулярные выражения позволяют классы Pattern и Matcher. Класс Pattern описывает шаблон регулярного выражения, а класс Matcher реализует методы поиска по шаблону. Прежде чем использовать эти классы, необходимо подключить их с помощью инструкций:

import java.util.regex.Pattern;

import java.util.regex.Matcher;

**28. Лямбда-выражения. Лямбды как параметры и результаты методов. Встроенные функциональные интерфейсы.**

Среди новшеств, которые были привнесены в язык Java с выходом JDK 8, особняком стоят лямбда-выражения. Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и 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. Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно: функциональный интерфейс должен содержать только один единственный метод без реализации. Функциональный интерфейс — это интерфейс, у которого ровно один абстрактный метод (метод, который объявлен, но не описан). Функциональный интерфейс может содержать несколько методов, но все они, за исключением одного, должны иметь код по умолчанию.

Например

**public class** Main {

**public static void** main(String[] args) {

Foo foo = (x,y) -> x + y;

**int** result = foo.calculate(10, 20);

System.out.println(result); //30

}

}

**interface** Foo {

**int** calculate(**int** x**, int** y);

}

Лямбда-выражение может использовать переменные, которые объявлены во вне (в более общей области видимости) - на уровне класса или метода, в котором лямбда-выражение определено. Однако в зависимости от того, как и где определены переменные, могут различаться способы их использования в лямбдах. Рассмотрим первый пример - использования переменных уровня класса:

**public class** Main {

**static int** x = 10;

**static int** y = 20;

**public static void** main(String[] args) {

Foo foo = () -> {

x = 30;

**return** x + y;

};

System.out.println(foo.calculate()); // 50

System.out.println(x); // 30 - значение x изменилось

}

}

**interface** Foo {

**int** calculate();

}

Переменные x и y объявлены на уровне класса, и в лямбда-выражении мы их можем получить и даже изменить. Так, в данном случае после выполнения выражения изменяется значение переменной x. Теперь рассмотрим другой пример - локальные переменные на уровне метода:

**public static void** main(String[] args) {

**int** n = 70;

**int** m = 30;

Foo foo = () -> {

//n = 100; - так нельзя сделать

**return** m + n;

};

// n = 100; - так тоже нельзя

System.out.println(foo.calculate()); // 100

}

Локальные переменные уровня метода мы также можем использовать в лямбдах, но изменять их значение нельзя. Если мы попробуем это сделать, то среда разработки может высветить ошибку и то, что такую переменную надо пометить с помощью ключевого слова **final**, то есть сделать константой: **final int** n = 70; Однако это необязательно. Более того, мы не сможем изменить значение переменной, которая используется в лямбда-выражении, вне этого выражения. То есть даже если такая переменная не объявлена как константа, по сути она является константой.

Одним из преимуществ лямбд в java является то, что их можно передавать в качестве параметров в методы. Рассмотрим пример:

**public class** Main {

**public static void** main(String[] args) {

int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int sum = calculateSum(nums, n -> n % 2 == 0);

System.out.println(sum); // 20

}

**private static int** calculateSum(int[] numbers, Condition cond){

int result = 0;

**for** (int i : numbers) {

if (cond.test(i)) result += i;

}

**return** result;

}

}

**interface** Condition {

**boolean** test(int n);

}

В JDK 8 вместе с самой функциональностью лямбда-выражений также было добавлено некоторое количество встроенных функциональных интерфейсов, которые мы можем использовать в различных ситуациях и в различные API в рамках JDK 8. В

частности, эти интерфейсы широко применяются в Stream API -- прикладном интерфейсе для работы с данными. Рассмотрим основные из этих интерфейсов:

**public interface** Predicate<T> {

**boolean** test(T t);

}

**public interface** BinaryOperator<T> {

T apply(T t1, T t2);

}

**public interface** UnaryOperator<T> {

T apply(T t);

}

**public interface** Function<T, R> {

R apply(T t);

}

**public interface** Consumer<T> {

**void** accept(T t);

}

**public interface** Supplier<T> {

T get();

}

29. Потоки ввода-вывода. FileInputStream и FileOutputStream. BufferedInputStream и BufferedOutputStream. DataOutputStream и DataInputStream.

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета java.io. Ключевым понятием здесь является понятие потока. Понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом мы

будем говорить о потоке (stream), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. У нас может быть определен поток, который связан с файлом и через который мы можем вести чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-вывод в консоли мы будем решать в Java с помощью потоков. Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные -

потоком вывода. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода. В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: InputStream (представляющий потоки ввода) и OutputStream (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы Reader (для чтения потоков символов) и Writer (для записи потоков символов).

Класс FileOutputStream предназначен для записи байтов в файл. Он является производным от класса OutputStream, поэтому наследует всю его функциональность. Например, запишем в файл строку:

**import** java.io.\*;

**public class** Main {

**public static void** main(String[] args) {

String text = "Hello world!"; // строка для записи

**try** (FileOutputStream fos = **new** FileOutputStream("C:\\xep.txt")) {

**byte**[] buffer = text.getBytes();

fos.write(buffer, 0, buffer.length);

} **catch** (IOException ex){

ex.printStackTrace();

}

}

}

Для создания объекта FileOutputStream используется конструктор, принимающий в качестве параметра путь к файлу для записи. Если такого файла нет, то он автоматически создается при записи. Если не удалось создать или открыть файл на запись, то генерируется исключение.

Для считывания данных из файла предназначен класс FileInputStream, который является наследником класса InputStream и поэтому реализует все его методы.

Считаем данные из ранее записанного файла и выведем на консоль:

**import** java.io.\*;

**public class** Main {

**public static void** main(String[] args) {

**try** (FileInputStream fin = **new** FileInputStream("C:\\xep.xyu")) {

**byte**[] buffer = new **byte**[fin.available()];

fin.read(buffer, 0, fin.available());

**for** (**int** i = 0; i < buffer.length; i++) {

System.out.print(**(char)**buffer[i]);

}

} **catch** (IOException ex){

ex.printStackTrace();

}

}

}

Класс BufferedInputStream накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода. Класс BufferedInputStream определяет два конструктора:

BufferedInputStream(InputStream inputStream)

BufferedInputStream(InputStream inputStream, int bufSize)

Первый параметр - это поток ввода, с которого данные будут считываться в буфер. Второй параметр - размер буфера.

Класс BufferedOutputStream аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству. И когда буфер заполнен, производится запись данных. BufferedOutputStream определяет два конструктора:

BufferedOutputStream(OutputStream outputStream)

BufferedOutputStream(OutputStream outputStream, int bufSize)

Первый параметр - это поток вывода, который унаследован от OutputStream, а второй параметр - размер буфера.

Классы DataOutputStream и DataInputStream позволяют записывать и считывать данные примитивных типов.

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

writeBoolean(boolean v) : записывает в поток булевое однобайтовое значение writeByte(int v): записывает в поток 1 байт, которые представлен в виде целочисленного значения

writeChar(int v): записывает 2-байтовое значение char

writeDouble(double v): записывает в поток 8-байтовое значение double

writeFloat(float v): записывает в поток 4-байтовое значение float

writeInt(int v): записывает в поток целочисленное значение int

writeLong(long v): записывает в поток значение long

writeShort(int v): записывает в поток значение short

writeUTF(String str): записывает в поток строку в кодировке UTF-8

Класс DataInputStream действует противоположным образом -- он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

boolean readBoolean(): считывает из потока булевое однобайтовое значение

byte readByte(): считывает из потока 1 байт

char readChar(): считывает из потока значение char

double readDouble(): считывает из потока 8-байтовое значение double

float readFloat(): считывает из потока 4-байтовое значение float

int readInt(): считывает из потока целочисленное значение int

long readLong(): считывает из потока значение long

short readShort(): считывает значение short

String readUTF(): считывает из потока строку в кодировке UTF-8

int skipBytes(int n): пропускает при чтении из потока n байтов

30. Чтение и запись текстовых файлов. Буферизация символьных потоков. BufferedReader и BufferedWriter.

Для работы с текстовыми файлами служат классы, которые являются наследниками абстрактных классов Reader и Writer. UTF-8 - самая популярная кодировка (encoding) в мире. Она может кодировать все письменности мира, включая китайские и японские иероглифы. Начиная с Java5, UTF-8 является кодировкой по умолчанию для чтения и записи файлов. Файлы UTF-8, созданные Java, не имеют BOM (Byte Order Mark) (Первые bytes файла, которые помечают его как UTF-8).

Класс FileWriter является производным от класса Writer. Он используется для записи текстовых файлов.

Запишем в файл какой-нибудь текст:

**import** java.io.\*;

**public class** Main {

**public static void** main(String[] args) {

**try** (FileWriter writer = **new** FileWriter("notes.txt", false)) {

String text = "Hello Gould!";

writer.write(text);

writer.flush();

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

}

В конструкторе использовался параметр со значением false - то есть файл будет перезаписываться. Затем с помощью методов, определенных в базовом классе Writer производится запись данных.

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность для чтения текстовых файлов.

**import** java.io.\*;

**public class** Main {

**public static void** main(String[] args) {

**try** (FileReader reader = **new** FileReader("notes.txt")) {

**int** c;

**while** ((c = reader.read()) != -1) {

System.out.print(**(char)**c);

}

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

}

Класс BufferedWriter записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных. Класс BufferedWriter имеет следующие конструкторы:

BufferedWriter(Writer out)

BufferedWriter(Writer out, int sz)

В качестве параметра он принимает поток вывода, в который надо осуществить запись. Второй параметр указывает на размер буфера.

Класс BufferedReader считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока. Класс BufferedReader имеет следующие конструкторы:

BufferedReader(Reader in)

BufferedReader(Reader in, int buffer)

Второй конструктор, кроме потока ввода, из которого производится чтение, также определяет размер буфера, в который будут считываться символы.

Соединим оба класса BufferedReader и BufferedWriter для считывания с консоли в файл.

**import** java.io.\*;

**public class** Main {

**public static void** main(String[] args) {

**try** (BufferedReader br = **new** BufferedReader (**new**

InputStreamReader(System.in));

BufferedWriter bw = **new** BufferedWriter(**new** FileWriter("notes.txt"))) {

String text;

**while** (!(text = br.readLine()).equals("ESC")) {

bw.write(text + "\n");

bw.flush();

}

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

}

31. Сериализация объектов

Сериализация представляет процесс записи состояния объекта в поток. Соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами. Сериализовать можно только те объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован. Для сериализации объектов в поток используется класс ObjectOutputStream. Он записывает данные в поток. Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись: **ObjectOutputStream(OutputStream out).** Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

**void close()** - закрывает поток;

**void flush()** - очищает буфер и сбрасывает его содержимое в выходной поток;

**void write(byte[] buf)** - записывает в поток массив байтов;

**void write(int val)** - записывает в поток один младший байт из val;

**void writeBoolean(boolean val)** - записывает в поток значение boolean;

**void writeByte(int val)** - записывает в поток один младший байт из val;

**void writeChar(int val)** - записывает в поток значение типа char, представленное целочисленным значением;

**void writeObject(Object obj)** - записывает в поток отдельный объект;

Внутри класса, поддерживающего сериализацию, необходимо также объявить закрытую статическую константу seriaiversionUiD. Пример объявления константы: private static final long serialVersionUID = 1L; Это значение сохраняется при сериализации и проверяется при восстановлении объекта. Если значения не совпадают, то генерируется исключение. Если константа не существует в классе, то это значение генерируется автоматически. Однако на разных компьютерах это автоматическое значение может оказаться разным, что может привести к исключению даже при правильной работе процесса сериализации. Также стоит помнить, что **при сериализации объекта сериализуются все объекты, на которые он ссылается в своих переменных экземпляра.** И если те объекты тоже ссылаются на третьи объекты, они тоже сериализуются. И так до бесконечности. Все классы в этой цепочке должны быть Serializable, иначе их невозможно будет сериализовать и будет выброшено исключение.

Класс ObjectInputStream отвечает за обратный процесс - чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода: **ObjectInputStream(InputStream in).** Функционал ObjectInputStream сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

**void close()** - закрывает поток;

**int skipBytes(int len)** - пропускает при чтении несколько байт, количество которых равно len;

**int available()** - возвращает количество байт, доступных для чтения;

**int read()** - считывает из потока один байт и возвращает его целочисленное представление; **boolean readBoolean()** - считывает из потока одно значение boolean;

**byte readByte()** - считывает из потока один байт;

**char readChar()** - считывает из потока один символ char;

**Object readObject()** - считывает из потока объект;

При сериализации объекта сериализуются все объекты, на которые он ссылается в своих переменных экземпляра. И если те объекты тоже ссылаются на третьи объекты, они тоже сериализуются. И так до бесконечности. Все классы в этой цепочке должны быть Serializable, иначе их невозможно будет сериализовать и будет выброшено исключение. Однако, возможно, мы хотим, чтобы некоторые поля были исключены из сериализации. Для этого они должны быть объявлены с модификатором transient.

import java.io.\*;

import java.util.ArrayList;

public class Program {

public static void main(String[] args) {

String filename = "people.dat";

// создадим список объектов, которые будем записывать

ArrayList<Person> people = new ArrayList<Person>();

people.add(new Person("Tom", 30, 175, false));

people.add(new Person("Sam", 33, 178, true));

try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename))) {

oos.writeObject(people);

System.out.println("File has been written");

}

catch(Exception ex){ System.out.println(ex.getMessage()); }

// десериализация в новый список

ArrayList<Person> newPeople= new ArrayList<Person>();

try(ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename))) {

newPeople=((ArrayList<Person>)ois.readObject());

}

catch(Exception ex){ System.out.println(ex.getMessage()); }

for(Person p : newPeople)

System.out.printf("Name: %s \t Age: %d \n", p.getName(), p.getAge());

}

}

class Person implements Serializable{

private String name;

private int age;

private double height;

private boolean married;

Person(String n, int a, double h, boolean m){

name=n;

age=a;

height=h;

married=m;

}

String getName() {return name;}

int getAge(){ return age;}

double getHeight(){return height;}

boolean getMarried(){return married;}

}

32. Класс File. Работа с файлами и каталогами.

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File. В зависимости от того, что должен представлять объект File - файл или каталог, мы можем использовать один из конструкторов для создания объекта:

File(String путь\_к\_каталогу);

File(String путь\_к\_каталогу, String имя\_файла);

File(File каталог, String имя\_файла);

Класс File имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

**boolean createNewFile()** - создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false;

**boolean delete()** - удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true;

**boolean exists()** - проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false;

**String getAbsolutePath()** - возвращает абсолютный путь для пути, переданного в конструктор объекта;

**String getName()** - возвращает краткое имя файла или каталога;

**String getParent()** - возвращает имя родительского каталога;

**boolean isDirectory()** - возвращает значение true, если по указанному пути располагается каталог;

**boolean isFile()** - возвращает значение true, если по указанному пути находится файл;

**boolean isHidden()** - возвращает значение true, если каталог или файл являются скрытыми;

**long length()** - возвращает размер файла в байтах;

**long lastModified()** - возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи (1-1-1970);

**String[] list()** - возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге;

**File[] listFiles()** - возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге;

**boolean mkdir()** - создает новый каталог и при удачном создании возвращает значение true;

**boolean renameTo(File dest)** - переименовывает файл или каталог;

Работа с каталогами: Если объект File представляет каталог, то его метод isDirectory() возвращает true. И поэтому мы можем получить его содержимое - вложенные подкаталоги и файлы с помощью методов list() и listFiles().

import java.io.File;

public class Program {

public static void main(String[] args) {

// определяем объект для каталога

File dir = new File("C:\\SomeDir");

// если объект представляет каталог

if(dir.isDirectory()) {

// получаем все вложенные объекты в каталоге

for(File item : dir.listFiles()){

if(item.isDirectory()){ System.out.println(item.getName() + " \t folder"); }

else{ System.out.println(item.getName() + "\t file"); }

}

}

}

Работа с файлами:

import java.io.File;

import java.io.IOException;

public class Program {

public static void main(String[] args) {

// определяем объект для каталога

File myFile = new File("C:\\SomeDir\\notes.txt");

System.out.println("File name: " + myFile.getName());

System.out.println("Parent folder: " + myFile.getParent());

if(myFile.exists())

System.out.println("File exists");

else

System.out.println("File not found");

System.out.println("File size: " + myFile.length());

if(myFile.canRead())

System.out.println("File can be read");

else

System.out.println("File can not be read");

if(myFile.canWrite())

System.out.println("File can be written");

else

System.out.println("File can not be written");

// создадим новый файл

File newFile = new File("C://SomeDir//MyFile");

try {

boolean created = newFile.createNewFile();

if(created)

System.out.println("File has been created");

}

catch(IOException ex){ System.out.println(ex.getMessage()); }

}

}

33. Многопоточное программирование. Создание и выполнение потоков. Завершение и прерывание потока.

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому большинство реальных приложений, которые многим из нас приходится использовать, практически не мыслимы без многопоточности. В Java функциональность отдельного потока заключается в классе Thread. И чтобы создать новый поток, нам надо создать объект этого класса. При запуске Java-программы начинает выполняться главный поток (main). Особенность главного потока состоит в том, что в нем порождаются все дочерние потоки. Главный поток отождествляется с программой. Программа начинается с выполнения главного потока. В отличие от дочерних потоков главный поток создается автоматически. Внутри класса Thread определено перечисление State, в котором заданы все возможные состояния потока. В каждый момент времени у потока может быть только одно состояние. Список всех состояний: • NEW – поток создан, но еще не запущен; • RUNNABLE – поток запущен и выполняется в JVM; • BLOCKED – поток блокирован; • WAITING – поток ждет окончания работы другого потока; • TIMED\_WAITING – поток приостановлен, ожидает некоторого действия или будет возобновлен по окончании времени ожидания; • TERMINATED – поток завершил выполнение;

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

**getName()** - возвращает имя потока;

**setName(String name)** - устанавливает имя потока;

**getPriority()** - возвращает приоритет потока;

**setPriority(int proirity)** - устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5. (Потоки с высоким приоритетом имеют преимущество над потоками с низким в глазах профилировщика потоков. Но это не значит, что потоки с низким приоритетом никогда не выполнятся. Технически завязываться на приоритет не всегда верно, т.к. на разных операционных системах реализованы разные механизмы параллельного выполнения потоков.);

**isAlive()** - возвращает true, если поток активен;

**isInterrupted()** - возвращает true, если поток был прерван;

**join()** - ожидает завершение потока;

**run()** - определяет точку входа в поток;

**sleep()** - приостанавливает поток на заданное количество миллисекунд;

**start()** - запускает поток, вызывая его метод run();

Создать и запустить поток можно двумя способами:

1) Создать наследника от класса Thread и переопределить метод run().

2) Реализовать интерфейс Runnable и передать объект полученного класса в конструктор класса Thread.

Пример создания наследника от класса Thread:

class JThread extends Thread {

JThread(String name){ super(name); }

public void run(){

System.out.printf("%s started... \n", Thread.currentThread().getName());

try{ Thread.sleep(500); }

catch(InterruptedException e){

System.out.println("Thread has been interrupted");

}

System.out.printf("%s fiished... \n", Thread.currentThread().getName());

}

}

public class Program {

public static void main(String[] args) {

System.out.println("Main thread started...");

new JThread("JThread").start();

System.out.println("Main thread finished...");

}

}

Пример реализации интерфейса Runnable:

class MyThread implements Runnable {

public void run(){

System.out.printf("%s started... \n", Thread.currentThread().getName());

try{ Thread.sleep(500); }

catch(InterruptedException e){

System.out.println("Thread has been interrupted");

}

System.out.printf("%s finished... \n", Thread.currentThread().getName());

}

}

public class Program {

public static void main(String[] args) {

System.out.println("Main thread started...");

Thread myThread = new Thread(new MyThread(), "MyThread");

myThread.start();

System.out.println("Main thread finished...");

}

}

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение. Например:

class MyThread implements Runnable {

private boolean isActive;

void disable(){ isActive=false; }

MyThread(){ isActive = true; }

public void run(){

System.out.printf("%s started... \n", Thread.currentThread().getName());

int counter=1; // счетчик циклов

while(isActive){

System.out.println("Loop " + counter++);

try{ Thread.sleep(400); }

catch(InterruptedException e){

System.out.println("Thread has been interrupted");

}

}

System.out.printf("%s finished... \n", Thread.currentThread().getName());

}

}

public class Program{

public static void main(String[] args) {

System.out.println("Main thread started...");

MyThread myThread = new MyThread();

new Thread(myThread,"MyThread").start();

try{

Thread.sleep(1100);

myThread.disable();

}

catch(InterruptedException e){

System.out.println("Thread has been interrupted");

}

System.out.println("Main thread finished...");

}

}

Для остановки потока в java существует метод interrupt. Вызов этого метода устанавливает у потока статус, что он прерван. При этом сам вызов этого метода НЕ завершает поток, он только устанавливает статус:

false - значит поток живой, что-то там делает. Т.е. false - НЕТ, не прерван.

true - значит поток остановлен. Т.е. true - Да, прерван.

Когда поток прерывается другим потоком, происходит одно из двух:

1. Если поток ожидает выполнения прерываемого метода блокирования, таких как Thread.sleep(), Thread.join() или Object.wait(), то ожидание прерывается и метод генерирует InterruptedException. Флаг interrupted устанавливается в false.
2. Флаг interrupted устанавливается в true.

Есть три метода для работы с прерыванием потока:

**public void interrupt() { ... }**

**public boolean isInterrupted() { ... }**

**public static boolean interrupted() { ... }**

Пример:

class JThread extends Thread {

JThread(String name){ super(name); }

public void run(){

System.out.printf("%s started... \n", Thread.currentThread().getName());

int counter=1; // счетчик циклов

while(!isInterrupted()){ System.out.println("Loop " + counter++); }

System.out.printf("%s finished... \n", Thread.currentThread().getName());

}

}

public class Program {

public static void main(String[] args) {

System.out.println("Main thread started...");

JThread t = new JThread("JThread");

t.start();

try{

Thread.sleep(150);

t.interrupt();

Thread.sleep(150);

}

catch(InterruptedException e){ System.out.println("Thread has been interrupted"); }

System.out.println("Main thread finished...");

}

}

**34. Многопоточное программирование. Синхронизация потоков. Оператор synchronized. Взаимодействие потоков. Методы wait и notify**

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно

Oсновное отличие потоков от процессов состоит в том, что потоки не

защищены друг от друга средствами операционной системы. Поэтому любой из потоков может получить доступ и даже внести изменения в данные, которые другой поток считает своими. Решение этой проблемы состоит в синхронизации потоков

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

notify(): продолжает работу потока, у которого ранее был вызван метод wait()

notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

**35. Многопоточное программирование. Семафоры. Класс Exchanger. Класс Phaser.**

Большинство языков программирования поддерживают такую важную функциональность как многопоточность, и Java в этом плане не исключение. При помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно

Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом Semaphore, который располагается в пакете java.util.concurrent.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore.

Класс Exchanger предназначен для обмена данными между потоками. Он является типизированным и типизируется типом данных, которыми потоки должны обмениваться.

Обмен данными производится с помощью единственного метода этого класса exchange()

Класс Phaser позволяет синхронизировать потоки, представляющие отдельную фазу или стадию выполнения общего действия. Phaser определяет объект синхронизации, который ждет, пока не завершится определенная фаза. Затем Phaser переходит к следующей стадии или фазе и снова ожидает ее завершения.

Для управления доступом к общему ресурсу в качестве альтернативы оператору synchronized мы можем использовать блокировки. Функциональность блокировок заключена в пакете java.util.concurrent.locks.

Вначале поток пытается получить доступ к общему ресурсу. Если он свободен, то на него накладывает блокировку. После завершения работы блокировка с общего ресурса снимается. Если же ресурс не свободен и на него уже наложена блокировка, то поток ожидает, пока эта блокировка не будет снята.Класы блокировок реализуют интерфейс Lock

**36. Stream API. Создание потока данных. Фильтрация, перебор элементов и отображение. Сортировка. Получение подпотока и объединение потоков. Методы skip и limit.**

Начиная с JDK 8 в Java появился новый API - Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.Ключевым понятием в Stream API является поток данных.

Для создания потока данных можно применять различные методы. В качестве источника потока мы можем использовать коллекции. В частности, в JDK 8 в интерфейс Collection, который реализуется всеми классами коллекций, были добавлены два метода для работы с потоками:

default Stream<E> stream: возвращается поток данных из коллекции

default Stream<E> parallelStream: возвращается параллельный поток данных из коллекции

Пример:

ArrayList<String> cities = new ArrayList<String>();

Collections.addAll(cities, "Париж", "Лондон", "Мадрид");

citiesStream = cities.stream() // получаем поток

Для перебора элементов потока применяется метод forEach(), который представляет терминальную операцию. В качестве параметра он принимает объект Consumer<? super String>, который представляет действие, выполняемое для каждого элемента набора.

Например:

citiesStream.forEach(s->System.out.println(s));

Для фильтрации элементов в потоке применяется метод filter(), который представляет промежуточную операцию. Он принимает в качестве параметра некоторое условие в виде объекта Predicate<T> и возвращает новый поток из элементов, которые удовлетворяют этому условию:

citiesStream.filter(s->s.length()==6).forEach(s->System.out.println(s));

Коллекции, на основе которых нередко создаются потоки, уже имеют специальные методы для сортировки содержимого. Однако класс Stream также включает возможность сортировки. Такую сортировку мы можем задействовать, когда у нас идет набор промежуточных операций с потоком, которые создают новые наборы данных, и нам надо эти наборы отсортировать.

Для простой сортировки по возрастанию применяется метод sorted():

Например:

citiesStream.sorted();// сортировка по возрастанию

Ряд методов Stream API возвращают подпотоки или объединенные потоки на основе уже имеющихся потоков

Метод takeWhile() выбирает из потока элементы, пока они соответствуют условию. Если попадается элемент, который не соответствует условию, то метод завершает свою работу. Выбранные элементы возвращаются в виде потока.

Статический метод concat() объединяет элементы двух потоков, возвращая объединенный поток.

Метод skip(long n) используется для пропуска n элементов. Этот метод возвращает новый поток, в котором пропущены первые n элементов.

Метод limit(long n) применяется для выборки первых n элементов потоков. Этот метод также возвращает модифицированный поток, в котором не более n элементов.

1. **Stream API. Метод reduce. Тип Optional.**

**Stream API.** Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream. Ключевым понятием в Stream API является поток данных. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы, и коллекции.

Вместо цикла и условных конструкций, мы можем записать цепочку методов, которые будут выполнять те же действия. При работе со Stream API важно понимать, что все операции с потоками бывают либо терминальными (terminal), либо промежуточными (intermediate). Промежуточные операции возвращают трансформированный поток. К возвращенному потоку также можно применить ряд промежуточных операций. Терминальные (конечные) операции возвращают конкретный результат. После этого никаких промежуточных операций, естественно, применять нельзя. Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция.

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

interface BaseStream < T, S extends BaseStream<T,S>>

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream. BaseStream определяет базовый функционал для работы с потоками, который реализуется через его методы. Например: void close(): закрывает поток, boolean isParallel(): возвращает true, если поток является параллельным , Iterator iterator(): возвращает ссылку на итератор потока.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

1. Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
2. Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
3. Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

**Метод reduce.** Метод reduce выполняет терминальные операции сведения, возвращая некоторое значение - результат операции. Он имеет следующие формы:

1. Optional<T> reduce(BinaryOperator<T> accumulator)
2. T reduce(T identity, BinaryOperator<T> accumulator)
3. U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

Первая форма возвращает результат в виде объекта Optional.

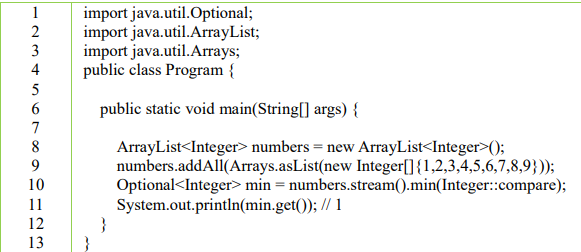
Вторая версия метода reduce() принимает два параметра:

T reduce(T identity, BinaryOperator<T> accumulator)

Первый параметр - T identity - элемент, который предоставляет начальное значение для функции из второго параметра, а также предоставляет значение по умолчанию, если поток не имеет элементов. Второй параметр - BinaryOperator accumulator, как и первая форма метода reduce, представляет ассоциативную функцию, которая запускается для каждого элемента в потоке и принимает два параметра. Первый параметр представляет промежуточный результат функции, а второй параметр - следующий элемент в потоке.

Третий параметр представляет бинарную операцию, которая суммирует все промежуточные вычисления.

**Тип Optional.** Ряд операций сведения, такие как min, max, reduce, возвращают объект Optional. Этот объект фактически обертывает результат операции. После выполнения операции с помощью метода get() объекта Optional мы можем получить его значение:



Если поток не содержит вообще никаких данных, то в этом случае программа выдаст исключение java.util.NoSuchElementException. Для решения этого исключения класс Optional предоставляет ряд методов. Самой простой способ избежать подобной ситуации - это предварительная проверка наличия значения в Optional с помощью метода isPresent(). Он возвращает true, если значение присутствует в Optional, и false, если значение отсутствует.

1. **Stream API. Метод collect.**

**Stream API.** Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream. Ключевым понятием в Stream API является поток данных. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы, и коллекции.

Вместо цикла и условных конструкций, мы можем записать цепочку методов, которые будут выполнять те же действия. При работе со Stream API важно понимать, что все операции с потоками бывают либо терминальными (terminal), либо промежуточными (intermediate). Промежуточные операции возвращают трансформированный поток. К возвращенному потоку также можно применить ряд промежуточных операций. Терминальные (конечные) операции возвращают конкретный результат. После этого никаких промежуточных операций, естественно, применять нельзя. Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция.

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

interface BaseStream < T, S extends BaseStream<T,S>>

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream. BaseStream определяет базовый функционал для работы с потоками, который реализуется через его методы. Например: void close(): закрывает поток, boolean isParallel(): возвращает true, если поток является параллельным , Iterator iterator(): возвращает ссылку на итератор потока.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

1. Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
2. Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
3. Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

**Метод collect**. Большинство операций класса Stream, которые модифицируют набор данных, возвращают этот набор в виде потока. Однако бывают ситуации, когда хотелось бы получить данные не в виде потока, а в виде обычной коллекции, например, ArrayList или HashSet. И для этого у класса Stream определен метод collect. Первая версия метода принимает в качестве параметра функцию преобразования к коллекции:

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

Параметр R представляет тип результата метода, параметр Т - тип элемента в потоке, а параметр А - тип промежуточных накапливаемых данных. В итоге параметр collector представляет функцию преобразования потока в коллекцию. Эта функция представляет объект Collector, который определен в пакете java.util.stream. Мы можем написать свою реализацию функции, однако Java уже предоставляет ряд встроенных функций, определенных в классе Collectors:

* toList(): преобразование к типу List
* toSet(): преобразование к типу Set
* toMap(): преобразование к типу Map

Вторая форма метода collect имеет три параметра:

<R>R collect(Supplier <R> supplier, BiConsumer <R,? super T> accumulator, BiConsumer<R,R> combiner)

* supplier: создает объект коллекции
* accumulator: добавляет элемент в коллекцию
* combiner: бинарная функция, которая объединяет два объекта

1. **Stream API. Параллельные потоки.**

**Stream API.** Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream. Ключевым понятием в Stream API является поток данных. Применительно к Stream API поток представляет канал передачи данных из источника данных. Причем в качестве источника могут выступать как файлы, так и массивы, и коллекции.

Вместо цикла и условных конструкций, мы можем записать цепочку методов, которые будут выполнять те же действия. При работе со Stream API важно понимать, что все операции с потоками бывают либо терминальными (terminal), либо промежуточными (intermediate). Промежуточные операции возвращают трансформированный поток. К возвращенному потоку также можно применить ряд промежуточных операций. Терминальные (конечные) операции возвращают конкретный результат. После этого никаких промежуточных операций, естественно, применять нельзя. Все потоки производят вычисления, в том числе в промежуточных операциях, только тогда, когда к ним применяется терминальная операция.

В основе Stream API лежит интерфейс BaseStream. Его полное определение:

interface BaseStream < T, S extends BaseStream<T,S>>

Здесь параметр T означает тип данных в потоке, а S - тип потока, который наследуется от интерфейса BaseStream. BaseStream определяет базовый функционал для работы с потоками, который реализуется через его методы. Например: void close(): закрывает поток, boolean isParallel(): возвращает true, если поток является параллельным , Iterator iterator(): возвращает ссылку на итератор потока.

Несмотря на то, что все эти операции позволяют взаимодействовать с потоком как неким набором данных наподобие коллекции, важно понимать отличие коллекций от потоков:

1. Потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
2. Операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
3. Для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

**Параллельные потоки.** Кроме последовательных потоков Stream API поддерживает параллельные потоки. Распараллеливание потоков позволяет задействовать несколько ядер процессора (если целевая машина многоядерная) и тем самым может повысить производительность и ускорить вычисления. В то же время говорить, что применение параллельных потоков на многоядерных машинах однозначно повысит производительность - не совсем корректно. В каждом конкретном случае надо проверять и тестировать. Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод parallel. Кроме того, можно также использовать метод parallelStream() интерфейса Collection для создания параллельного потока из коллекции. В то же время если рабочая машина не является многоядерной, то поток будет выполняться как последовательный.

1. Структура программы. Комментарии. Консольный ввод/вывод в Java.

2. Переменные и константы.

3. Базовые типы данных.

4. Арифметические операции. Условные выражения. Операции присваивания и приоритет операций. 5. Преобразования базовых типов данных.

6. Условные конструкции.

7. Операторы цикла. Операторы break и continue.

8. Массивы. Объявление массивов. Инициализация массивов. Одномерные массивы. Многомерные массивы.

9. Методы. Параметры методов. Результат метода. Перегрузка методов.

10. Java ООП. Классы и объекты. Конструкторы. Ключевое слово this. Инициализаторы.

11. Модификаторы доступа и инкапсуляция.

12. Статические члены и модификатор static.

13. Внутренние и вложенные классы.

14. Наследование.

15. Иерархия наследования и преобразование типов.

16. Интерфейсы. Интерфейсы в механизме обратного вызова.

17. Клас Object. Методы класса Object.

18. Перечисления enum.

19. Обобщения (Generics). Обобщенные интерфейсы. Обобщенные методы.

20. Ограничения обобщений. Наследование и обобщения.

21. Records.

22. Введение в обработку исключений. Оператор throws. Создание своих классов исключений.

23. Типы коллекций. Интерфейс Collection. Класс ArrayList и интерфейс List. Очереди и класс ArrayDeque. Класс LinkedList.

24. Интерфейс Set и класс HashSet. SortedSet, NavigableSet, TreeSet.

25. Интерфейсы Comparable и Comporator. Сортировка. Итераторы.

26. Интерфейс Map и класс HashMap. Интерфейсы SortedMap и NavigableMap. Класс TreeMap.

27. Работа со строками. StringBuffer и StringBuilder. Регулярные выражения.

28. Лямбда-выражения. Лямбды как параметры и результаты методов. Встроенные функциональные интерфейсы.

29. Потоки ввода-вывода. FileInputStream и FileOutputStream. BufferedInputStream и BufferedOutputStream. DataOutputStream и DataInputStream.

30. Чтение и запись текстовых файлов. Буферизация символьных потоков. BufferedReader и BufferedWriter.

31. Сериализация объектов.

32. Класс File. Работа с файлами и каталогами.

33. Многопоточное программирование. Создание и выполнение потоков. Завершение и прерывание потока.

34. Многопоточное программирование. Синхронизация потоков. Оператор synchronized. Взаимодействие потоков. Методы wait и notify

35. Многопоточное программирование. Семафоры. Класс Exchanger. Класс Phaser. Блокировки. ReentrantLock.

36. Stream API. Создание потока данных. Фильтрация, перебор элементов и отображение. Сортировка. Получение подпотока и объединение потоков. Методы skip и limit.

37. Stream API. Метод reduce. Тип Optional.

38. Stream API. Метод collect.

39. Stream API. Параллельные потоки.