***Integrated Development Environment******(IDE)*** – графическая среда интегрированной разработки IDE (Visual Studio, IDEA).

**Горячие клавиши IDEA.**

*Ctrl+Alt+L* – автоматическое форматирование кода.

*Ctrl+Alt+0* – удалить неиспользуемые импорты.

*Ctrl+Alt+V* – извлечение значения в отдельную переменную.

*Alt+Insert* – вызов меню генерации кода.

*Ctrl+Alt+T* – вызов меню генерации тестового класса.

*Ctrl+F* – поиск. *F3* и *Shift* – перемещение по найденным элементам. *Ctrl+R* – замена на этой странице.

*Ctrl+Shift+F* – поиск по указанному пути. *Ctrl+Shift+R* – замена по указанному пути.

*Shift+F6* – переименование переменной.

Двойное нажатие *Shift* – поиск справки.

*Ctrl+Shift+F10* – запуск проекта. *Ctrl+F2* – остановка проекта.

*Ctrl+X* – вырезать текущую строку.

*Ctrl+D* – дублировать текущую строку.

*Ctrl+/* – закомментировать строки как однострочный комментарий.

*Ctrl+Shift+/* – закомментировать строки как многострочный комментарий.

*Ctrl+левый\_клик* или *Ctrl+B* – переход к месту определения метода.

Наведение на метод при нажатом *Ctrl* вызывает подсказку с параметрами метода.

Key Promoter X – плагин напоминает о горячих клавиш при использовании мыши.

Live Templates – плагин запоминает и выводит шаблоны кода.

*sout* – разворачивается в инструкцию *System.out.println();*

*ifn* – разворачивается в конструкцию *if* проверки на равенство *null*.

*fori* – разворачивается в конструкцию *for* с переменное *i* в качестве счетчика.

*iter* – разворачивается в заготовку прохода по списку или массиву.

*System.out.println(); // Вывод сообщения в консоль*

*System.in.read(); // Ввод данных с клавиатуры*

Ввод данных с клавиатуры с фильтрацией ввода:

*import java.util.Scanner; // импорт библиотеки*

*Scanner scanner = new Scanner(System.in); // объявление переменной*

*String command = scanner.nextLine(); // ввод строки из консоли*

С помощью *scanner* можно считать такую информацию:

*nextLine() — строки*

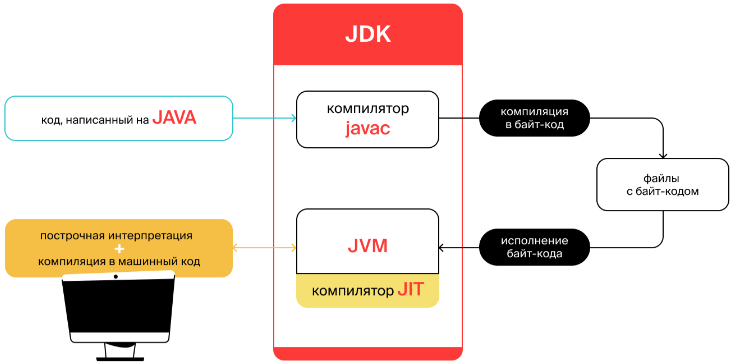
*nextInt() — целые числа*

*nextDouble() — числа с дробной частью*

В *Java* 8 появились лямбды, коллекции и потоки.

В *Java* 10 появился тип *var*.

В *Java* 11 появился релиз *HttpClient*.



***Java Virtual Machine (JVM)*** – независимая от платформы исполняющая среда для байт-кода, сгенерированного компилятором (***javac***). Она контролирует выполнение кода и ограничивает его выполнение за пределами исполняющей среды. Эталонной считается *HotSpot JVM* от разработчиков *Java*.

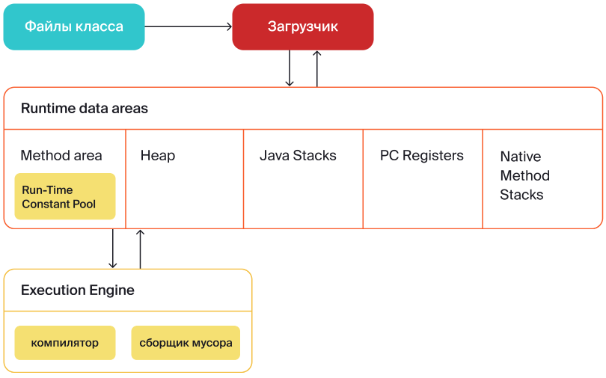
Внутреннее устройство *JVM* можно разбить на три группы:

1. **Подсистема загрузки классов**. Отвечает за поиск и загрузку файлов классов, которые нужны для выполнения программы. Класс ***Platform*** загружает стандартные библиотеки и *JDK*. Класс ***System*** загружает классы самого приложения. Класс ***Bootstrap*** загружает другие загрузчики и пакеты *JRE*.
2. **Подсистема, ответственная за выполнение байт-кода**.

По умолчанию за это отвечает ***JIT***-компилятор (***Just-In-Time***). Он оптимизирует и динамически преобразовывает наиболее часто используемый байт-код в машинный код, и за счет этого повышается производительность приложения.

Альтернативой является статический ***AOT***-компилятор (***Ahead-Of-Time***), который заранее преобразовывает всю программу в машинный код.

1. **Подсистема взаимодействия с памятью**.



***PC Registers*** – локальная область для каждого потока, содержащая адрес инструкции *JVM*, которую поток выполняет в данный момент.

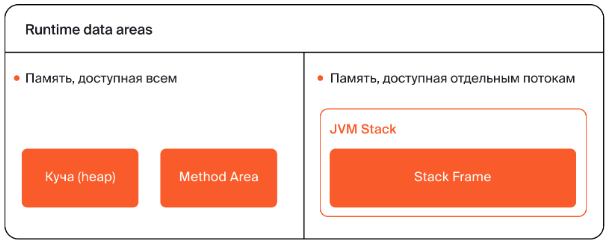
***Java Stacks* (стек)** – локальная область для каждого потока. Она хранит примитивные типы значений и ссылки на объекты кучи. Если потоку требуется больше места в стеке, чем разрешено, то возникает ошибка ***StackOverflowError*** либо ***OutOfMemoryError*** (при динамически расширяемом стеке).

***Heap* (куча)** – это общая область для всех потоков. Она содержит фактические объекты, метаданные классов, массивы и объекты, которые создаются во время работы программы и на которые ссылаются переменные из стека. Создается при запуске *JVM* и уничтожается при завершении её работы. Не освобождается автоматически и управляется ***Garbage Collector*** **(сборщиком мусора)**. Для выделения памяти в куче используется ключевое слово ***new***.

***Method Area* (область метода)** – общая область для всех потоков. Она хранит структуры для каждого класса. Создается при запуске *JVM*.

***Run-Time Constant Pool* (пул констант времени выполнения)** – это структура данных в памяти для каждого класса или типа. Содержит константы, необходимые для запуска кода определенного класса.

***Native Method Stacks* (стеки нативных методов)** – выделяются для каждого потока при его создании. Используются для методов, написанных не на *Java*.



Сборщик мусора работает **на слабой гипотезе о поколениях** – чем младше объект, тем больше вероятность, что он станет не нужным после первого использования. Во время сборки мусора JVM приостанавливается. Если объект является **достижимым** (на него ссылаются), значит его нельзя удалять. Подходы проверки достижимости:

* **Метод подсчета ссылок**. Для каждого объекта в куче создается счетчик ссылок. Сборщик мусора обходит все объекты и оставляет только те, у которых счетчик больше нуля. Из минусов – мусорные объекты могут ссылаться сами на себя; постоянные задержки во время перерасчета счетчика.
* **Метод флагов (или метод маркировки и выметания)**. Для каждого объекта в памяти хранится бит, указывающий на его достижимость. Изначально все объекты помечаются как недостижимые. Не проверяются корневые объекты.
* **Метод карточного стола**. Представляет объекты старшего поколения в виде набора карт. Если старший объект ссылается на молодой, то карта переворачивается и при следующей проверке на достижимость проверяются только объекты, доступные “из-под этой карты”. Сборщик мусора может убрать только часть кучи.

Область в куче делится на область младшего (***Eden, S0, S1***) и область старшего (***Tenured***) поколений. Это позволяет избавиться от фрагментации памяти. Если места не хватает в области старшего поколения, то запускается полная сборка мусора в обоих областях.

*HotSpot JVM* включает в себя три типа сборщиков с разными характеристиками производительности:

* ***Serial* (последовательные)** – используют один поток для сборки мусора. Подходит для однопроцессорных пк и объемом данных до 100Мб. Требует полную остановку приложения.
* ***Parallel* (параллельные)** – используют несколько потоков для сборки мусора. Подходят для многопроцессорных пк и средним объемом данных. Требует полную остановку приложения.
* ***Concurent* (конкурентные)** – многопоточные сборщики мусора. Большая часть работ выполняется, не останавливая потоки приложения, а конкурирую с ними. В *Java* 11 по умолчанию используется сборщик мусора ***G1***. Начиная с *Java* 15, альтернативой является сборщик мусора ***ZGC***.

**Основные характеристики *JVM*:**

* пропускная способность определяет пиковую нагрузку
* время отклика определяет как приложение справляется с остановками на сбор мусора
* размер используемой памяти, необходимый для эффективной работы сборщика мусора

**Основные принципы настройки сборщика мусора:**

* Максимальное число объектов должно быть удалено во время малой сборки. Это позволит избежать полной остановки приложения.
* Чем больше памяти выделено, тем лучше работает сборка мусора.
* Для эффективной работы приложения нужно настроить две из трех основных характеристик JVM.

Задать эти настройки можно с помощью флагов при запуске *JVM*.

Java чувствителен к регистру имен переменных и методов. В именах допускается использовать цифры (но нельзя начинать имя с цифры), \_ и $. В Java не поддерживаются целочисленные значения без знака. Однострочные комментарии обозначаются //, многострочные - /\* \*/.

**Внутренние типы данных**

|  |  |  |
| --- | --- | --- |
| **Обозначение** | **Диапазон** | **Описание** |
| bool | true или false | Признак истинности или ложности |
| byte | от -128 до 127 | 8-битное число |
| short | от -32 768 до 32 767 | 16-битное число |
| int | от -2 147 483 648 до 2 147 483 647 | 32-битное число |
| long | от -9 223 372 036 854 775 808  до 9 223 372 036 854 775 807 | 64-битное число |
| char | от U+0000 до U+ffff | Одиночный 16-битный символ Unicode |
| float | от -3.4х10^38 до 3.4х10^38 | 32-битное число с плавающей точкой |
| double | от +-5.0х10^(-324) до +-1.7х10^308 | 64-битное число с плавающей точкой |

При записи чисел типа *long* необходимо добавлять букву *L* в конце. Переменную *char* можно инкрементировать, а также присваивать целочисленные значения. Данный тип задается парными одинарными кавычками (например, ‘k’). Тип *float* хранит 38 знаков после запятой, а тип *double* - больше 300 знаков после запятой. В коде вместо 10 ставится буква *Е*. Все примитивы хранят конкретное значение.

**Управляющие последовательности**

|  |  |  |  |
| --- | --- | --- | --- |
| \’ | Одинарная кавычка | \f | Перевод страницы |
| \” | Двойная кавычка | \t | Горизонтальная табуляция |
| \\ | Обратная косая черта | \b | Возврат на одну позицию |
| \r | Возврат каретки | \ddd | Восьмеричная константа |
| \n | Перевод строки | \uxxxx | Шестнадцатеричная константа |

Не рекомендуется использовать синтаксис множественных объявлений или инициализации переменных, т.к. это мешает восприятию и сбивает с толку.

Префиксная форма (++x) записи операций инкремента и декремента изменяет значение до ей использования, а постфиксная (x++) – после.

**Логические операторы**

|  |  |  |  |
| --- | --- | --- | --- |
| & | И (побитовое) | && | Укороченное И |
| | | ИЛИ (побитовое) | || | Укороченное ИЛИ |
| ^ | Исключающее ИЛИ | ! | НЕ |

Побитовые операторы неактуальны.

Укороченные операторы могут не вычислять второй оператор, если результат уже не изменится.

По приоритетам – отрицание (!), операции в скобках, логическое И (&&), логическое ИЛИ (||).

Допускается **неявное преобразование типов**. Например, byte в int.

Для **явного преобразования типов** нужно явно указать целевой тип. Например, здесь потеряется дробная часть в результате преобразования:

*double x = 10.0, y = 3.0;*

*int i = (int) (x / y);*

В *java* реализованы условные конструкции ***if-else*** и ***if-else-if***.

**Условная операция** **(тернарный оператор)** (?:) является сокращенным способом написания простого оператора if-else.

*условие ? первое\_выражение : второе\_выражение;*

Например:

*absval = val < 0 ? -val : val;*

*или*

*if(val < 0) absval = -val;*

*else absval = val;*

Также реализована условная конструкция ***switch***. Условное выражение может быть типами *int*, *char, byte, short* или *String*. При отсутствии ключевого слова *break* сначала выполняются инструкции текущей ветви, а затем следующей за ней и т.д. Поддерживаются вложенные условные конструкции *switch*.

*switch (month) {*

*case 1: // январь*

*case 3: // март*

*case 5: // май*

*case 7: // июль*

*case 8: // август*

*case 10: // октябрь*

*case 12: // декабрь*

*System.out.println("В этом месяце 31 день.");*

*break;*

*case 2: // февраль*

*System.out.println("В этом месяце 28 или 29 дней.");*

*break;*

*case 4: // апрель*

*case 6: // июнь*

*case 9: // сентябрь*

*case 11: // ноябрь*

*System.out.println("В этом месяце 30 дней.");*

*break;*

*default:*

*System.out.println("Месяца с таким номером не существует."); }*

Цикл ***for*** подходит для тех случаев, когда заранее известно количество итераций. Цикл *for* не ограничивается определением и итерацией только одной переменной, их может быть несколько. В условии цикла не обязательно должна находиться переменная его итерирования. Цикл *for* позволяет при объявлении пропускать одно, два или все условия.



Цикл ***while*** используется для расчётов, когда работа программы зависит от внешних данных, которые изначально неизвестны — например, их вводит пользователь.

*while (условие) {*

*последовательность инструкций*

*}*

Цикл ***do-while*** используется для выполнения инструкций до проверки условия цикла.

*do {*

*последовательность инструкций*

*} while (условие);*

Для выхода из любого цикла можно использовать ключевое слово ***break***. Для досрочного завершения текущей итерации и переходу к следующей используется ключевое слово ***continue***.

В коде со множеством вложенных циклов можно использовать переход по меткам. **Метка** – это имя, обозначающее блок кода. Например:

*for(i=1; i<4; i++) {*

*one: {*

*two: {*

*three: {*

*System.out.println("\ni равно " + i);*

*if(i==l) break one; // Переход по метке*

*if(i==2) break two;*

*if(i==3) break three; }}}}*

**Массив** – это набор однотипных элементов данных, для доступа к которым применяется числовой индекс. Индексация массивов начинается с нуля. Если массив объявлен, но его элементы не заданы явным образом, то они получат стандартные значения в соответствии с их типом. Массивы являются ссылочными типами.

**Индекс** — номер элемента в массиве, его тип значения всегда *int*. Узнать длину массива можно с помощью вызова ***length***.

Объявление массива:

*String[] currencies = {"USD", "EUR"};*

или

*String[] currencies = new String[2];*

*currencies[0] = "USD";*

*currencies[1] = "EUR";*

Пример объявления **двухмерного массива**:

*int table[][] = new int[2][3];*

*table[0][0] = 1;*

*table[0][1] =2;*

*…*

или

*int table[][] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} }*

Перебор массива циклом *for*:

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

*System.out.println("Поддерживаемая валюта: " + currencies[i]); }*

Перебор массива циклом *for-each*:

*for (String s : currencies) {*

*System.out.println("Поддерживаемая валюта: " + s); }*

Другие многомерные массивы объявляются и инициализируются аналогичным образом.

**Нерегулярные массивы** позволяют выделить память под массив на основании первого (самого левого) размера. Это позволяет определить размер второго измерения вручную (они могут различаться по размеру):

*int table[][] = new int[3][];*

*table[0] = new int[10];*

*table[1] = new int[10];*

*table[2] = new int[2];*



**Перечисления** – это список именованных констант, определяющих новый тип данных. Объекты перечислений могут иметь значения только из этого списка. Для объявления перечислений используется ключевое слово ***enum***. Константы являются неявно ***static final*** членами перечислений.

Для сравнения констант перечислимого типа используется оператор сравнения (==). Перечисления можно использовать в качестве селектора в переключателе *switch*.

*enum Transport {CAR, AIPLANE, TRAIN}*

*class EnumDemo {*

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

*Transport tp = Transport.AIPLANE; // Инициализация без new*

*if (tp == Transport.TRAIN) // Можно сравнивать через ==*

*System.out.println(“tp содержит TRAIN\n”);*

*switch (tp) {*

*case CAR: break;*

*case AIPLANE: break;*

*case TRAIN: break; } } }*

По умолчанию все перечисления реализуют два метода:

*public static перечислимый\_тип[] values()*

*public static перечислимый\_тип valueOf(String str)*

Первый возвращает массив со списком констант перечисления, а второй – константу перечисления, которая соответствует строке *str*.

*Transport allTransport[] = Transport.values();*

*tp = Transport.valueOf(“AIRPLANE”);*

В Java перечисления реализованы как типы классов, но для их объявления **не требуется слово** ***new***. Допускается объявление конструкторов, методов и переменных, а также реализация интерфейсов. В этом случае после объявления констант ставится точка с запятой.

Перечисления не могут быть подклассами и суперклассами, однако они наследуют переменные и методы класса ***java.lang.Enum***. Основными методами являются ***ordinal()*** и ***compareTo()***. Первый возвращает порядковое значение константы (начиная с нуля), а второй сравнивает эти порядковые значения.

*if (tp.compareTo(tp2) < 0)*

*System.out.println(tp + “ идет перед ” + tp2);*

К *enum* можно применять методы *ToString(), getClass(), hashCode()* и другие.

**Строка (*String*)**,начиная с *java* 9, является массивом байтов (выделяет на символ столько места, сколько требуется). Значения ***String*** являются константными (строковые литералы), поэтому они не меняются, а создаются новые строки.

Все строки являются ссылочными типами. Строки можно объединять в массивы. При создании строк с помощью литерала (не через *new*) они попадают в особое хранилище – пул строк. При повторном создании идентичной строки вернется ссылка на уже существующую строку.

Метод ***equals(String)*** используется для сравнения строк и возвращает булево значение.

Метод ***toString()*** возвращает строковое представление объекта, а при вызове для строк просто возвращает их самих.

Метод ***length()*** возвращает количество символов в строке.

Метод ***isEmpty()*** возвращает *true*, если строка является пустой.

Метод ***isBlank()*** возвращает *true*, если строка пустая или содержит только пробельные символы.

Метод ***trim()*** возвращает строку, из которой удалены пробельные символы в начале и в конце.

Методы ***toUpperCase()*** и ***toLowerCase()*** переводят строки в верхний и нижний регистры.

Метод ***split(String regex)*** превращает строку в массив строк по разделителю и возвращает его. Для использования спец.символов в качестве разделителя, их нужно экранировать с помощью **\\** .

Метод ***join(String delimeter, String … strs)*** превращает массив подстрок в строку с разделителем и возвращает её. Запись ***String … strs*** называется ***variable arguments (varargs)*** и обозначает, что метод может принять любое количество аргументов или массив.

**Подстрока** **(*substring*)** – это непрерывный набор символов внутри строки. Символы в подстроках нумеруются как в массиве (с нуля).

Метод ***charAt(int)*** возвращает символ, расположенный по указанному индексу.

Методы ***indexOf(String)*** и ***lastIndexOf(String)*** возвращают индекс начала подстроки (отличаются направлением поиска). Если подстрока не найдена, вернется -1. Для поиска с конкретного символа используется перегрузка этих методов - ***indexOf(String, int)*** и ***lastIndexOf(String, int)***.

Метод ***contains(String)*** возвращает *true*, если подстрока будет найдена в любом месте строки.

Метод ***startsWith(String)*** возвращает *true*, если подстрока будет найдена в начале строки.

Метод ***endsWith(String)*** возвращает *true*, если подстрока будет найдена в конце строки.

Метод ***substring(int)*** возвращает подстроку, начиная с указанного индекса и до конца строки.

Метод ***substring(int firstIndex, int lastIndex)*** возвращает подстроку, указанную между индексами, включая первый и не включая последний индексы.

Метод ***replace(String target, String replacement)*** возвращает новую строку, в которой заменены все вхождения подстроки *target* на подстроку *replacement*.

Метод ***replaceFirst(String target, String replacement)*** возвращает новую строку, в которой заменено первое вхождение подстроки *target* на подстроку *replacement*.

Для представления строки в виде набора редактируемых символом используется класс ***StringBuilder***. У него есть изменяемый размер буфера, как у списка. Вместимость можно задать при создании экземпляра через конструктор, принимающий *int*.

Метод ***toString()*** преобразует содержимое *StringBuilder* в неизменяемую строку и возвращает её.

Класс *StringBuilder* поддерживает методы *indexOf(String)*, *lastIndexOf(String)*, *substring(int)* и *substring(int beginIndex, int endIndex)*.

Метод ***append(String)*** добавляет подстроку в конец.

Метод ***insert(int index, String str)*** добавляет подстроку в середину (со сдвигом вправо).

Метод ***replace(int indexFromInclusive, int indexToExclusive, String replacement)*** заменяет обозначенный промежуток на указанную подстроку. Первый индекс берется включительно, а второй - нет.

Методы ***deleteCharAt(int)*** и ***delete(int startIndexInclusive, int endIndexExclusive)*** удаляют элементы по указанным индексам. Первый индекс берется включительно, а второй - нет.

Метод ***reverse()*** позволяет отзеркалить (перевернуть) строку.

Метод ***setLength(int)*** обрезает строку до указанного размера.

**Палиндром** – это число, сочетание букв или текста, которые читаются одинаково как слева направо, так и справа налево.

Для форматирования вывода используется метод ***String.format(String format, Object… args)***:

*String[] colors = new String[]{"красный", "жёлтый", "зелёный"};*

*String trafficLight = String.format("Цвета светофора: %s, %s и %s.", colors[0], colors[1], colors[2]);*

**Популярные символы преобразования**

|  |  |  |  |
| --- | --- | --- | --- |
| Обозначение | Тип | Обозначение | Тип |
| %s | String | %d | Целое число |
| %S | Вывод строки в верхнем регистре | %c | char |
| %b | boolean | %t | Date |
| %B | boolean в верхнем регистре | %% | символ % |
| %f | Вещественное число с плавающей точкой | %n | Перенос строки в форм.строках |

Также для форматирования можно использовать метод ***System.out.printf((String format, Object… args)***.

*System.out.printf("%b", "Панама") // выведет true (преобразование строк)*

*System.out.printf("%25s", pizza); // минимальное число символов в строке = 25 (по правому краю)*

*System.out.printf("%-25s", pizza); // теперь выровняем текст по левому краю*

*System.out.printf("%1.12s", pizza); // в строке теперь от 1 до 12 символов*

*System.out.printf("%2.2f", price); // минимум 2 символа и ровно 2 символа после запятой*

***SOLID*** – это набор принципов, следуя которым, программный код будет более чистым и гибким.

***S****ingle-responsibility* (единая ответственность) – один класс или файл должен иметь только одну цель и одно единственное назначение.

***O****pen-closed* (принцип открытости/закрытости, или наследования) – программные сущности должны быть открыты для расширения, но закрыты для модификации.

***L****istov substitution principle* (принцип подстановки Барбары Лисков) – поведение методов в дочернем классе должно следовать принципам базового класса, а не изменять его. Т.е. если базовый метод возвращал высоту, то и дочерний должен возвращать высоту, а не длину/скорость или что-то иное.

***I****nterface segregation* (принцип разделения интерфейса) – лучше создавать много отдельных интерфейсов, чем один, который включает в себя все функции.

***D****ependency inversion* (инверсия зависимостей) – модули верхних уровней не должны зависеть от модулей нижних уровней. Т.е. пишем код так, чтобы он имел как можно меньше зависимостей.

Одним из главных свойств Java является поддержка **объектно-ориентированного программирования** (ООП). В языках ООП определяются данные и процедуры, которым разрешается обрабатывать эти данные. Таким образом, тип данных определяет те операции, которые применимы к этим данным. Во всех языках ООП поддерживаются три основных принципа ООП: инкапсуляция, полиморфизм и наследование.

1. **Инкапсуляция** характерная черта, описывающая способность языка скрывать лишние детали реализации от пользователя объекта и предохраняющая целостность данных (черный ящик).

Данные и код внутри объекта могут быть закрытыми (**private**) или открытыми (**public**). Закрытый код или данные доступны только элементам, содержащимся в том же самом объекте. Если код или данные являются открытыми, то к ним можно обращаться из любой части программы. Как правило, открытые элементы объекта используются для создания управляемого интерфейса к его закрытым элементам. Основной языковой конструкцией, поддерживающей инкапсуляцию, является класс.

1. **Полиморфизм** – способность языка трактовать связанные объекты в сходной манере. Она позволяет реализовывать одно и то же действие в зависимости от контекста (например, рисовать разные геометрические фигуры).
2. **Наследование** – процесс, в ходе которого один объект приобретает свойства другого объекта. Это отражает возможность языка разрешать построение определений новых классов на основе определений существующих классов (стимулирование многократного использования уже написанного кода).

**Класс** – это определяемый пользователем тип, состоящий из полей данных (переменных-членов) и членов, которые оперируют полями данных (конструкторы, свойства, методы, события и т.д.). **Объектами** называются экземпляры классов в памяти. Для обращения к переменных-членам класса используется **точечная нотация**. Классы являются ссылочными типами.

Создание класса, как типа данных:

*class имя\_класса {*

*// описание переменных-членов }*

Создание объекта класса:

*имя\_класса имя\_объекта\_класса // объявление ссылки на объект*

*имя\_объекта\_класса = new имя\_класса(список параметров); // выделение памяти для объекта*

или сразу

*имя\_класса имя\_объекта\_класса = new имя\_класса(список параметров);*

**Конструктор** – это специальный метод класса, который неявно вызывается при создании объекта с применением слова ***new***. Он позволяет установить состояние объекта в момент его создания. Конструктор не имеет возвращаемого значения и всегда именуется идентично имени класса, объекты которого он конструирует.

**Конструктор по умолчанию** есть у всех классов, он инициирует все переменные экземпляра их значениями по умолчанию (ссылочные типы – null, логические – false, остальные – нулями).

**Простой конструктор** позволяет задавать нужные стандартные значения по умолчанию. Например:

*class MyClass {*

*int x;*

*MyClass() {*

*x = 10; } }*

**Cпециальный конструктор** работает аналогично перегрузке метода (с тем же самым именем, но разным количеством или типами аргументов).

*class MyClass {*

*int x;*

*MyClass(int i) {*

*x = i; } }*

При наличии неоднозначности в именах для доступа класса к собственным данным или членам используется слово **this**. Например:

*class Motorcycle {*

*public String driverName;*

*public void SetDriverName (String driverName) => this.driverName = driverName; }*

Конструкторы можно перегружать.

Через ***this*** можно обратиться к другим конструкторам, чтобы не дублировать код:

*class Bot {*

*String name;*

*public Bot() { // конструктор без параметров*

*System.out.println("Чат-бот создан"); }*

*public Bot(String name) { // конструктор с параметром*

*this(); // вызвали конструктор без параметров*

*this.name = name;*

*System.out.println("Вас приветствует " + name); } }*

**Методы** – это подпрограммы, которые манипулируют данными. Во многих случаях они предоставляют доступ к закрытым данным класса. Имя метода лучше начинать с глагола.

*возвращаемый\_тип имя\_метода(список параметров) {*

*// тело метода }*

Ключевое слово ***return*** позволяет завершить текущий метод и вывести возвращаемое значение заданного типа (при наличии).

Значения аргументов могут передаваться методу двумя способами:

1. **Вызов по значению**. В этом случае метод копирует значение аргумента и не может изменить исходный аргумент. Например:

*void noChange(int i, int j) {*

*i = i + j;*

*j = -j; }*

1. **Вызов по ссылке**. В этом случае метод получает ссылку на аргумент и оказывает на него влияние.

*int a, b;*

*Test (int i, int j) {*

*a = i;*

*b = j; }*

*void change(Test ob) {*

*ob.a = ob.a + ob.b;*

*ob.b = -ob.b; }*

**Перегрузка методов** – набор идентично именованных методов, которые отличаются друг от друга количеством или типами параметров.

Ключевое слово ***static*** используется для обозначения глобальных членов класса, которые можно использовать на уровне класса без ссылки на объект (принадлежит классу, а не объекту). Они будут общими для всех объектов класса. Например, метод *sqrt()* класса *Math*. Ограничения статичных методов:

* Допускается использовать только статических переменных, определенных в его классе.
* Допускается вызов только других статических методов.
* Должна отсутствовать ссылка *this* и *super*.

**Статические блоки** (аналоги статических конструкторов) – это набор инструкций, который выполняется до первого использования класса. Например, в нем можно инициализировать начальные значения статичных переменных:

*class StaticBlock{*

*static double rootOf2;*

*static double rootOf3;*

*static{ rootOf2 = Math.sqrt(2.0);*

*rootOf3 = Math.sqrt(3.0); }*

Статические переменные и методы нельзя перезаписать, но можно переделать через сокрытие.

**Вложенный класс** — это такой класс, который объявляется в другом классе. Вложенные классы, объявленные без применения ключевого слова *static,* являются **внутренними**. Внутренний класс имеет доступ ко всем элементам внешнего класса.

Для передачи списка параметров не фиксированной длины используется **метод с переменным числом аргументов**. Такой список должен передаваться последним аргументом. Методы с переменным числом аргументов также можно перегружать, но лучше отказаться от этого в связи с возможной неоднозначностью при вызовах метода. Например, следующий код объявляет аргументами строку *msg* и массив *v* типа *int* :

*static void vaTest(String msg, int … v) { }*

Наследуемый класс называется **суперклассом**, а наследующий – **подклассом**. Наследование объявляется с помощью ключевого слова ***extends***:

*class имя\_подкласса extends имя\_суперкласса*

*class Circle extends TwoDShape*

Множественное наследование не поддерживается в Java – у каждого подкласса может быть только один суперкласс.

Подклассы не имеют доступ к закрытым переменным суперкласса. Для доступа к закрытым переменным и сохранения инкапсуляции класса, рекомендовано создать и использовать геттеры и сеттеры – специальные методы ***get*** и ***set***. Например:

*double getHeight() { return height; }*

*void setHeight(double h) { height = h; }*

**Сокрытие** – объявления поля в подклассе с таким же именем, что и поле в суперклассе.

Объект наследника можно сохранить в переменной родительского типа.

Если конструктор определен только в подклассе, то конструируется объект подкласса, а родительская часть создается конструктором по умолчанию (без параметров).

Если конструктор определен в подклассе и в суперклассе, то выполняются оба конструктора. Для вызова конструктора суперкласса используется ключевое слово ***super*** – будет вызван конструктор ближайшего суперкласса, который соответствует указанным аргументам. Это справедливо и для многоуровневой иерархии.

*class TwoDShape {*

*private double width;*

*private double height;*

*TwoDShape(double w, double h) {*

*width = w;*

*height = h; }*

*class Triangle extends TwoDShape {*

*private String style;*

*Triangle(String s, double w, double h) {*

*super(w, h);*

*style = s; }*

При создании объекта конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Метод ***super()*** должен быть первым в инструкции конструктора подкласса. Если в классе-родителе есть конструктор без параметров, то в конструкторе подкласса компилятор вызовет его автоматически, неявно.

Допускается конструирование объекта на основе другого объекта:

*TwoDShape(TwoDShape ob) {*

*width = ob.width;*

*height = ob.height; }*

Ключевое слово ***super*** также используется для обращения к переменным и методам суперкласса, скрываемых переменными и методами подкласса (одно и то же имя). Например:

*B(int a, int b) {*

*super.i = a; // переменная i из класса A*

*i = b; } // переменная i из класса B*

Вызвать родительский метод можно также через точечную нотацию *super.someMethod()*.

**Аннотация** позволяет отслеживать изменения суперкласса и сигнализировать об этом при компиляции.

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с применением процесса, который называется **переопределением метода**. Переопределение (или **перегрузка**) метода происходит, когда сигнатуры (имя и аргументы) переопределяемого и переопределяющего методов совпадают. Например, переопределим метод *area()*:

*class TwoDShape {*

*double area() {*

*return 0.0 } }*

*class Triangle extends TwoDShape {*

*@Override // аннотация переопределения метода (необязательна)*

*double area() {*

*return getWidth() \* getHeight() / 2 ; } }*

Специальный класс ***Object*** является суперклассом для всех остальных классов. Он всегда неявно наследуется. Ссылочная переменная типа *Object* может ссылаться на объект любого класса.

Для сравнения объектов собственных классов и корректного поиска по спискам нужно **переопределить** метод ***equals(Object)***. Для корректного поиска по хэш-таблицам нужно **переопределить** методы ***equals(Object)*** и ***hashCode()***, т.к. одинаковые объекты должны иметь одинаковый хеш.

*import java.util.Objects;*

*import java.util.Arrays;*

*import java.util.Objects;*

*@Override*

*public boolean equals(Object o) {*

*if (this == o) return true; // проверяем адреса объектов*

*if (o == null || getClass() != o.getClass()) return false; // проверяем ссылку на null и сравниваем классы*

*Book otherBook = (Book) o; // открываем доступ к полям другого объекта*

*return Objects.equals(title, otherBook.title) && // проверяем все поля*

*Objects.equals(author, otherBook.author) && // нужно логическое «и»*

*(pagesNumber == otherBook.pagesNumber); } // примитивы сравниваем через ==*

*@Override*

*public int hashCode() {*

*// вызываем вспомогательный метод и передаём в него нужные поля*

*return Objects.hash(title, author, pagesNumber); }*

Для получения текущего состояния объекта в виде текста используется метод ***toString()***. В связи с тем, что его базовая реализация неинформативна, рекомендуется его всегда **переопределять**.

*class Address {*

*public String city;*

*public String street;*

*public int houseNumber;*

*@Override // переопределяем toString*

*public String toString() {*

*return city + ", " + street + ", д. " + houseNumber; } } // вызов println(address) вернет эту строку*

Метод ***Arrays.toString(Object[] o)*** проверяет массив на *null* и возвращает его текстовое представление.

Классы и методы имеют модификаторы доступов – открытые (***public***), закрытые (***private***) и защищенные (***protected***).

Закрытые члены доступны только внутри класса, в котором определены.

Защищенные члены могут использоваться классом, который их определяет, и любым дочерним классом. Однако, они недоступны внешнему миру.

Если модификатор не указан, то используется модификатор по умолчанию – ***default*** или ***package-private***.

**Пакет** – область в **пространстве имен**, объединяющая логически связанные классы. В одном пространстве имен все классы должны обладать уникальным именем.

В пакетах определены специальные права доступа к содержимому пакета. Это обеспечивает дополнительный уровень инкапсуляции.

Чтобы создать пакет, достаточно поместить инструкцию ***package*** в начало файла. В результате классы, определенные в этом файле, будут принадлежать указанному пакету. Поскольку пакет определяет пространство имен, имена классов, содержащиеся в файле, войдут в это пространство имен как его составные части. Класс не может относиться к нескольким пакетам сразу.

*package имя\_пакета;*

**Уровни доступа к членам классов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Закрытый член | Член, доступный по умолчанию | Защищенный член | Открытый член |
| Доступен в том же классе | Да | Да | Да | Да |
| Доступен из подкласса в том же пакете | Нет | Да | Да | Да |
| Доступен из любого класса в том же пакете | Нет | Да | Да | Да |
| Доступен из подкласса в любом пакете | Нет | Нет | Да | Да |
| Доступен из всякого класса в любом пакете | Нет | Нет | Нет | Да |

Исходя из таблицы, чтобы класс и его методы стали доступны за пределами пакета, в котором он определен, необходимо объявить открытыми (*public*) класс, его конструктор и методы.

Для того чтобы воспользоваться классом в другом пакете, нужно применить инструкцию ***import*** либо указать полное имя класса, предваренное именем пакета. Чтобы импортировать все содержимое пакета, вместо имени класса следует указать звездочку (**\***). Инструкция *import* должна следовать сразу за инструкцией *package*. Пакеты могут быть вложенными (пакет с пакетами).

*import имя\_пакета.имя\_класса;*

*import имя\_пакета.\*;*

Популярные пакеты – *java.lang* (импортируется автоматически), *java.io* , *java.net* , *java.applet* , *java.awt* .

В java поддерживается механизм **статического импорта**, позволяющего ссылаться на статические члены по их простым именам без дополнительного указания имени класса. Например, класс *Math*:

*import static java.lang.Math.\*; // для всех статических членов*

*import static java.lang.Math.sqrt; // для одного статического члена*

*double x = sqrt(2);*

**Абстрактный метод** – это член базового класса, который не представляет базовую реализацию, а предлагает только сигнатуру.

**Абстрактный класс** – это базовый класс, объекты которого нельзя создать. При его создании используется ключевое слово **abstract**. Если в классе объявлен абстрактный метод, то он обязан тоже быть абстрактным. Подкласс наследующий абстрактный класс, должен реализовывать все методы суперкласса. В абстрактном классе могут быть определены конкретные методы без указания ключевого слова *abstract*.

В абстрактный класс выносятся общие для всех наследников поля и методы.

Например, просто объектов *TwoDShape* не существует, он должен быть либо *Triangle*, либо *Circle*. Идея фигуры является допустимой, но это не конкретная сущность.

*abstract class TwoDShape {*

*abstract double area(); }*

*class Triangle extends TwoDShape {*

*@Override*

*double area() {*

*return getWidth() \* getHeight() / 2 ; } }*

Конструктор наследника неявно использует конструктор абстрактного родителя.

Методы и классы, объявленные с использованием ключевого слова ***final***, не могут быть переопределены и унаследованы.

Ключевое слово ***final*** также используется для объявления именованных констант переменных.

Для объявления константы класса используются оба слова ***static*** и ***final*** в любой последовательности:

*static final тип ИМЯ\_КОНСТАНТЫ = значение;*

**Интерфейс** – это коллекция абстрактных членов (методов), не представляющих свои стандартные реализации. Он добавляет нескольким классам определенный функционал. Классы обязаны иметь реализацию всех методов интерфейсов, которые они поддерживают. Интерфейсы объявляются с помощью ключевого слова ***interface*** и могут быть реализованы в любом количестве классов.

*уровень\_доступа interface имя {*

*возвращаемый\_тип имя\_метода\_1 (список\_параметров);*

*тип переменная\_1 = значение; }*

Уровень доступа может быть *public* (доступен в любом пакете) либо не указывается (доступен только внутри пакета). Методы, объявленные в интерфейсе, по умолчанию неявно считаются *public*. Переменные интерфейса неявно обозначаются ключевыми словами *public*, *final*, *static* и являются константами, которые обязательно подлежат инициализации при объявлении интерфейса.

Для реализации интерфейса в классе используется ключевое слово ***implements***:

*class имя\_класса extends суперкласс implements интерфейс\_1, интерфейс\_2 {*

*// тело класса }*

Интерфейсы поддерживают наследование с помощью ключевого слова ***extends***. Если класс реализует один интерфейс, наследующий другой, то в нем следует определить все методы, объявленные в интерфейсах по всей цепочке наследования.

Поля переменных в интерфейсе всегда являются константами.

Нельзя создать объекты на основе интерфейса, но можно объявить переменную с типом интерфейса и присвоить ей в качестве значения любой объект, который реализует данный интерфейс. Грубо говоря, мы берем объект и обрезаем все лишнее, чтобы он поместился в родительский класс.

*MailClient drone = new Drone(); // Интерфейс переменная = новый объект*

Начиная с JDK 8, интерфейсы могут иметь **реализацию по умолчанию** (они также не могут иметь экземпляров объектов). Они используются, если другие реализации не предоставляются. Это обеспечивает сохранение работоспособности существующего кода, даже если интерфейс был обновлен, и не реализовывать опциональные методы интерфейса. Для реализации интерфейсов по умолчанию используется ключевое слово ***default***. При желании можно перегрузить метод интерфейса по умолчанию.

*public interface MyIF {*

*int getUserID();*

*default int getAdminID() {*

*return 1; } }*

Реализация методов, определенная в классе, всегда имеет более высокий приоритет по сравнению с методами по умолчанию.

Если класс реализует два интерфейса с одинаковыми именами методов по умолчанию и не переопределяет их, то возникнет ошибка.

Если класс реализует два интерфейса с одинаковыми именами методов по умолчанию, не переопределяет их и один из этих интерфейсов наследует другой, то будет применяться реализация метода более расширенного интерфейса.

На заданную по умолчанию реализацию можно ссылаться явно с помощью ключевого слова ***super***:

*Alpha.super.reset();*

Начиная с JDK 8 интерфейс может определять статические методы. Их также можно вызывать только без экземпляров объектов. Для вызова такого метода в классе, реализовывающем такой интерфейс, нужно использовать следующую форму:

*имя\_интерфейса.имя\_статического\_метода();*

Начиная с JDK 9 интерфейс может включать закрытые методы, которые вызываются только методом по умолчанию или другими закрытыми методами. Они позволяют нескольким методам по умолчанию использовать один и тот же фрагмент кода, не дублируя его.

**Полиморфный класс** – класс, который связан отношением ***is-a*** с более чем одним типом (*Object*).

**Классический полиморфизм** – способность языка одинаково работать с объектами (поддержка наследуемых методов), если они имеют одинаковый интерфейс или родительский класс.

**Динамический *ad-hoc* полиморфизм** – заключается в переопределении методов. Он позволяет им демонстрировать различное поведение при вызове для разных типов.

**Статический *ad-hoc* полиморфизм** – заключается в перегрузке методов. В классе может быть несколько методов с одинаковым названием, но с разными типами параметров.

**Параметрический полиморфизм** – позволяет реализовать поведение класса или метода с указанием параметров и возвращаемых значений общего вида вместо конкретных (не зависимо от типа данных).

Автоматическое приведение ссылочных типов происходит при присваивании объекта класса-наследника переменной с типом интерфейса или базового класса.

*Document document = new RussianPassport("1111", "567899");*

*System.out.println(document.getSeries()); // ошибка - тип Document ничего не знает о методах*

*RussianPassport passport = (RussianPassport) document; // явное приведение, теперь все ок*

*System.out.println(passport.getSeries());*

Для определения типа объекта используется метод ***instanceof***, возвращающий *boolean*. Например:

*Boolean isPasport = document instanceof RussianPassport;*

Термин **обобщение** (**дженерик**) означает параметризированный тип. Классы, интерфейсы и методы, оперирующие параметризированными типами, называются **обобщенными**. Это помогает алгоритмам работать одинаково, независимо от того, к данным какого типа они применяются (например, быстрая сортировка). Таким образом обобщения обеспечивают безопасность типов.

*class Gen<T> { // класс с обобщенным типом T*

*T ob;*

*Gen(T o) { // конструктор принимает объект типа T*

*ob = o; }*

*T getob() { // метод возвращает объект типа T*

*return ob; }*

*void showType() {*

*System.out.println(“Тип T – это “ + ob.getClass().getName()); } }*

*class GenDemo {*

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

*Gen<Integer> iOb; // создание ссылки на объект типа Gen<Integer>*

*iOb = new Gen<Integer>(88);*

*iOb.showType();*

*int v = iOb.getob();*

*Gen<String> strOb = new Gen<String>(“Тест”); // создание ссылки на объект типа Gen<String>*

*strOb.showType();*

*String str = strOb.getob(); } }*

В данном примере имя параметра ***T*** является **заполнителем**, подлежащим замене фактическим типом, передаваемым конструктору при создании объекта.

Механизм обобщений не позволяет использовать примитивные типы (int, double, char). Вместо них можно использовать любой тип класса (Integer, Double).

Обобщенный класс можно создать с несколькими параметрами типа:

*class TwoGen<T, V> { // класс с двумя обобщенными типами*

*T ob1;*

*V ob2;*

*Gen(T o1, V o2) {*

*ob1 = o1;*

*ob2 = o2; }*

*T getob1() {*

*return ob1; }*

*V getob2(){*

*return ob2; } }*

*class GenDemo {*

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

*TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, “Тест”);*

*Int v = tgObj.getob1();*

*String str = tgObj.getob2(); } }*

Когда необходимо ограничить параметризированные типы набором определенных типов используются **ограниченные типы**. Например, когда параметр *T* предполагает использование только числовых типов. Для информации, классы *Integer* и *Double* являются подклассами суперкласса *Number*.

*class NumericFns<T extends Number> { // аргументом типа должен быть Number или его подкласс*

*T num;*

*NumericFns(T n) { // конструктор принимает объект типа T*

*num = n; }*

*double reciprocal() { // метод возвращает объект типа double*

*return 1 / num.doubleValue(); } }*

*class BoundsDemo {*

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

*NumericFns <Integer> iOb = new NumericFns <Integer>(5);*

*System.out.println(“Обратная величина iOb - ” + iOb.reciprocal());*

*NumericFns <Double> dOb = new NumericFns <Double>(5.25);*

*System.out.println(“Обратная величина dOb - ” + dOb.reciprocal()); } }*

Для сравнения двух параметризированных типов, один из которых заранее неизвестен, используется **шаблон аргумента** (**?**). В данном случае выражение *NumericFns<?>* соответствует любому типу объекта из класса *NumericFns*:

*boolean absEqual(NumericFns<?> ob) {*

*if (Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()) {*

*return true; } }*

Шаблоны аргументов также можно ограничивать, как и параметры типов.

*static void test(Gen<? extends A> o) { // указать верхнюю границу (доступно A и всем его подклассам)*

*… }*

*static void test(Gen<? super B> o) { // указать нижнюю границу (доступно B и всем его суперклассам)*

*… }*

**Обобщенные методы** не требуют наличия обобщенного класса. Параметры типа объявляются перед возвращаемым типом. Например, метод сравнения двух массивов с одинаковыми или совместимыми типами:

*static <T extends Comparable<T>, V extends T> boolean arraysEqual(T[] x, V[] y) {*

*if (x.length != y.length) return false;*

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

*if (!x[i].equals(y[i])) return false; }*

*return true; }*

**Обобщенные конструкторы** не требуют наличия обобщенного класса. Например, класс ниже вычисляет сумму всех чисел от 0 до *N*, при этом *N* передается обобщенному конструктору:

*class Summation {*

*private int sum;*

*<T extends Number> Summation(T arg) {*

*sum = 0;*

*for (int i =0; i <= arg.intValue(); i++) sum += i; }*

*int getSum() {*

*return sum; } }*

Любой класс реализующий **обобщенный интерфейс** должен быть также обобщенным. Исключение возможно только при указании интерфейсом конкретного типа. Обобщенные интерфейсы также могут быть ограниченными.

*interface Containment<T> {*

*boolean contains(T o); }*

*class MyClass<T> implements Containment<T> { // типизированный класс*

*… }*

*class MyClass implements Containment<T> { // нетипизированный класс*

*… }*

Обобщения позволяют использовать в качестве параметризированного типа тип ***Object***, но это небезопасно и может привести к ошибкам.

Вызов перегрузки методов обобщенных типов может вызывать неоднозначность, поэтому рекомендуется использовать методы с разными именами.

Начиная с JDK 7, можно сокращать объявление объектов обобщенного типа с помощью **ромбовидного оператора**:

*TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(42, “testing”); // полностью*

*TwoGen<Integer, String> tgOb = new TwoGen<>(42, “testing”); // сокращенно*

Ограничения использования обобщений:

1. Создать экземпляр параметра типа невозможно.

*class Gen<T> {*

*T ob;*

*Gen() {*

*ob = new T(); } } // нельзя*

1. В статическом члене нельзя использовать параметры типа, объявленные в его классе.

*class Wrong<T> {*

*static T ob; // нельзя*

*static T getob() { // нельзя*

*return ob; } }*

1. Нельзя получить экземпляр массива, тип элементов которого определяется параметром типа.

*vals = new T[10] // нельзя*

1. Нельзя создать массив обобщенных ссылок на объекты конкретного типа.

*Gen<Integer> gens[] = new Gen<Integer>[10] // нельзя*

1. Обобщенный класс не может расширять класс *Throwable*.

**Автоупаковка** - процесс автоматической инкапсуляции простого типа данных в объектную оболочку соответствующего типа всякий раз, когда в этом возникает необходимость (создавать такой объект явным образом не нужно). **Автораспаковка** – это обратный процесс автоматического извлечения значения, упакованного в объектную оболочку. Автоматический процесс выглядит так:

*Integer iOb = 100;*

*int i = iOb;*

До версии JDK 5 упаковка и распаковка выполнялись вручную:

*Integer iOb = new Integer(100);*

*int i = iOb.intValue();*

Благодаря автораспаковке объектные оболочки целочисленных типов можно использовать в инструкции *switch*.

*switch (iOb) {*

*case 1: break;*

*case 2: break; }*

Следует помнить, что с любой операцией распаковки и упаковки связаны издержки, отсутствующие при использовании простых типов.

Преобразование строк для каждого из классов-обёрток:

*Byte.parseByte("12");*

*Short.parseShort("345");*

*Integer.parseInt("999999");*

*Long.parseLong("1000000000000");*

*Float.parseFloat("12.3");*

*Double.parseDouble("456.789");*

*Boolean.parseBoolean("true");*

**Модуль** – совокупность пакетов и ресурсов, доступных по имени модуля. Начиная с JDK 9 модули доступны для *Java API*. Компиляция модульных программ происходит с помощью ***javac***. Модульное приложение может быть упаковано в файл *JAR* (*Java Archive*).

*module имя\_модуля {*

*// определение модуля }*

Также можно определить по умолчанию открытый модуль с помощью ключевого слова ***open module***. Это позволяет сделать доступными пакеты модуля для рефлексии. **Рефлексия** – механизм, позволяющий программе анализировать код на этапе выполнения.

*open module имя\_модуля {*

*// определение модуля }*

Для открытия доступа только к определенному пакету модуля используется ключевое слово ***opens***.

*opens имя\_пакета;*

Для объявления зависимости одного модуля от другого используется ключевое слово ***requires***. Для объявления доступности пакетов модуля для других используется ключевое слово ***exports***. Для сужения открываемого доступа к модулю используется ключевое слово ***to*** (**квалифицированный экспорт**). Для предоставления возможности неявного чтения (A зависит от B, B зависит от С, A хочет читать C) используется ключевое слово ***transitive***. Ключевое слово ***static*** позволяет указать, что модуль требуется только для компиляции, но не для выполнения.

*requires имя\_модуля;*

*requires transitive имя\_модуля;*

*requires static имя\_модуля;*

*exports имя\_пакета;*

*exports имя\_пакета to имена\_модулей;*

В силу своей значимости модуль *java.base* автоматически доступен для других модулей. Код, который не является частью именованного модуля, автоматически становится частью **безымянного модуля**. В безымянном модуле все пакеты автоматически экспортируются, и он имеет доступ к любым другим модулям, а вместо пути модуля используется путь класса при компиляции.

В Java поддерживается архитектура подключаемых приложений благодаря использованию **служб** и **провайдеров служб**. **Служба** – это программная единица, функциональность которой определяется интерфейсом или абстрактным классом. **Провайдер** предоставляет конкретную реализацию службы. Например, служба определяет возможность перевода на другой язык, а разные провайдеры реализуют перевод на нужный язык.

*class ServiceLoader<S> // класс определяющий провайдеры*

*public static <S> ServiceLoader<S> load(Class <S> тип\_службы) // загрузка провайдера*

Ключевой слово ***provides*** используется для указания, что модуль реализует службу. Ключевой слово ***uses*** используется для указания службы, которая требуется модулю. Ключевой слово ***with*** используется для указания конкретного типа провайдера службы.

*provides тип\_службы with типы\_реализации;*

*uses тип\_службы;*