1. Общие характеристики и особенности языка java. Преимущества и надостатки

Язык программирования **Java** – это высокоуровневый объектно-ориентированный язык, разработанный в компании Sun Microsystems.

Приемущества:

Язык Java прост для изучения. При разработке Java было уделено большое внимание простоте языка, поэтому программы на Java, по сравнению с программами на других языках, проще писать, компилировать, отлаживать и изучать. Низкий порог вхождения.

1.**Переносимость**. Программы, написанные на языке Java, после однократной трансляции в байт-код могут быть исполнены на любой платформе, для которой реализована виртуальная Java-машина. Наиболее эффективно возможности реального компьютера может использовать только программа, написанная с использованием «родного» машинного кода.
2.**Безопасность.**Функционирование программы полностью определяется (и ограничивается) виртуальной Java-машиной. Отсутствуют указатели и другие механизмы для непосредственной работы с физической памятью и прочим аппаратным обеспечением компьютера. Дополнительные ограничения снижают возможность написания эффективно работающих Java-программ.
3.**Надежность.**В языке Java отсутствуют механизмы, потенциально приводящие к ошибкам: арифметика указателей, неявное преобразование типов с потерей точности и т.п. Присутствует строгий контроль типов, обязательный контроль исключительных ситуаций. Многие логические ошибки обнаруживаются на этапе компиляции. Наличие дополнительных проверок снижает эффективность выполнения Java-программ.
4.**Сборщик мусора.**Освобождение памяти при работе программы осуществляется автоматически с помощью «сборщика мусора», поэтому программировать с использованием динамически распределяемой памяти проще и надежнее. При интенсивной работе с динамически распределяемой памятью возможны ошибки из-за того, что «сборщик мусора» не успел освободить неиспользуемые области памяти.
5.**Стандартные библиотеки.**Многие задачи, встречающиеся при разработке программного обеспечения, уже решены в рамках стандартных библиотек. Использование объектно-ориентированного подхода позволяет легко использовать готовые объекты в своих программах. Для запуска приложения необходима установка JRE, содержащего полный набор библиотек, даже если все они не используются в приложении. Отсутствие библиотеки необходимой версии может воспрепятствовать запуску приложения.
6.**Самодокументируемый код.**Имеется механизм автоматического генерирования документации на основе комментариев, размещенных в тексте программ.

Недостатки:

- Ресурсоёмкость и медлительность
Java, несмотря на различные способы оптимизации, всё же довольно ресурсоёмка и медлительна. Причины в следующем:
    - автосборка мусора
    - компиляция "на лету" (Just In Time compilation)
    - отказ от таких опасных механизмов как:арифметика указателей, неявное преобразование типов с потерей точности, функции первого класса

- Многословность и громоздкость
Одна из проблем Java. И причины здесь можно выделить две: жесткая политика Объектно-Ориентированного Подхода (далее ООП), отсутствие функций первого класса (first-class function).
Политика ООП введена в Java с целью избавления от некоторых опасностей, возникающих при создании крупного проекта. В Java же невозможно создавать функции, не являющиеся методами какого-либо класса. К примеру, в C++ такого запрета нет. Из-за этого в C++, при проектировании больших проектов, возникает целый ряд опасностей. Использование функций "не методов" приводит к тому что:

- может возникнуть конфликт при именовании функций, когда имя функции с таким же набором параметров и такого же типа данных уже имеется.
- проект в целом становиться более неповоротлив
- проект становиться труден для осмысления, не только новопришедшему программисту, но и тому кто давно в проекте

Запрет на функциональный подход избавляет от подобных опасностей и учит хорошему тону программирования.
Функции первого порядка по-сути очень сходны с классами первого порядка. Это функции, которые можно передавать как параметры. В C++ это решается при помощи поддержки указателей на функции. В Java все же есть способы передачи ссылки на функцию, но обычно там где в C++ используется указатель на функцию, в Java принято использовать Интерфейсы. Это более многословная технология, но при этом она избавляет от опасностей при использовании указателей.
Как видите на одной чаше весов многословность с громоздкостью, а на другой надежность работы проекта и безопасность.

2 JVM. Структура и особенности организации

Виртуальная машина - это программное обеспечение, основанное на понятиях и идее относительно воображаемого компьютера, который имеет логический набор команд, и команд, определяющих операции этого компьютера. Это .можно сказать, небольшая операционная система. Она формирует необходимый уровень абстракции, где достигается независимость от платформы, используемого оборудования.
Компилятор конвертирует исходный текст в код, который основан на воображаемой системе команд компьютеров и не зависит от специфичности процессора. Интерпретатор - приложение, которое понимает эти потоки команд и преобразовывает эти команды для используемого оборудования, к которому относится интерпретатор. JVM создает систему поддержки выполнения внутренне, что помогает выполнению кода при

- загрузке файлов .class,
- управлению памятью
- выполнении обработки исключений.

Из-за несогласованности аппаратных платформ виртуальная машина использует понятие стека, который содержит следующую информацию:

- Описатели состояния метода
- Операнды к байт-кодам
- Параметры методов
- Локальные переменные

Когда код выполняется с помощью JVM, то существует один специальный регистр, который используется как счетчик, указывая выполняющиеся в настоящее время команды. Если необходимо, команды изменяют программу, изменяют поток выполнения, иначе поток последователен и переходит от одной команды к другой.
Другое понятие, которое становится популярным - это использование Just In Time (JIT) компилятора. Основная цель JIT состоит в том, чтобы преобразовать систему команд байт-кода к машинным командам кода, целенаправленным для специфического микропроцессора. Эти команды сохраняются и используются всякий раз, когда запрос делается к этому специфическому методу.
Непосредственно виртуальная машина Java «не знает» ничего о языке программирования Java, ей лишь известен заданный формат двоичных файлов – файлов, имеющих расширение `class`. Эти файлы содержат инструкции виртуальной машины (байткод), таблицы символов и другую вспомогательную информацию. Из соображений безопасности виртуальная машина Java предъявляет строгие синтаксические и структурные требования на код, расположенный в `class` файле. Тем не менее, любой язык, функциональность которого может быть выражена в средствами корректного `class` файла, может быть интерпретирован для виртуальной машины Java. Привлечённые общедоступностью и платформенной независимостью, разработчики компиляторов других языков могут использовать виртуальную машину как удобную платформу для своих реализаций.

3 Структура программы на Java. Пакетная организация механизмы импорта

Программа на языке Java представляет собой набор классов. В простейшем случае программа состоит из единственного класса, который обязательно содержит метод main(). Следует иметь в виду, что язык Java – регистрозависимый.
Заголовок класса содержит служебное слово **class** и имя класса.
Для имени класса существуют следующие ограничения и требования:
· Оно должно начинаться с буквы.
· Не допускается использование некоторых знаков (например – плюс, минус, апостроф, кавычки).
· Имя класса не должно совпадать со служебным словом (например – class, System, public, int)
Принято называть классы с заглавной буквы, а пакеты со строчной.
Java пакеты могут содержаться в сжатом виде в JAR файлах. Обычно в пакеты объединяют классы одной и той же категории, либо предоставляющие сходную функциональность.

- Каждый пакет предоставляет уникальное пространство имен для своего содержимого.
- Допустимы вложенные пакеты.

Классы, определенные без явно заданных модификаторов доступа (**`public`**, **`protected`**, **`private`**), видимы только внутри пакета.

	Класс	*Пакет*	*Подклассы*	*Все*
`private`	Да	—	—	—
(без)	Да	Да	—	—
`protected`	Да	Да	Да	—
`public`	Да	Да	Да	Да

- **`private`** — доступ только внутри класса (наиболее рекомендуемый)
- (без модификатора) — только внутри пакета (по умолчанию)
- **`protected`** — межпакетный доступ только для подклассов
- **`public`** — межпакетный доступ (наименее рекомендуемый)

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен. Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву `package`, после которой указывается имя пакета.
Механизм импорта
В исходном файле Java-программы операторы import должны следовать непосредственно за оператором package (если таковой имеется) перед любыми определениями классов.
Оператор import имеет следующую общую форму:
import пакет1 [.пакет2].(имя__класса|*);
В этой форме пакет1 — имя пакета верхнего уровня, пакет 2 — имя подчиненного пакета внутри внешнего пакета, отделенное символом точки (.). Глубина вложенности пакетов практически не ограничена ничем, кроме файловой системы. И, наконец, имя_ класса может быть задано либо явно, либо с помощью символа звездочки (*), который указывает компилятору Java о необходимости импорта всего пакета. Использование формы с применением символа звездочки может привести к увеличению времени компиляции — особенно при импорте нескольких больших пакетов
В java есть также особая форма импорта - статический импорт. Для этого вместе с директивой import используется модификатор static.
При наличии в двух различных пакетах, импортируемых с применением формы со звездочкой, классов с одинаковыми именами компилятор никак на это не отреагирует, если только не будет предпринята попытка использования одного из этих классов. В этом случае возникнет ошибка времени компиляции, и имя класса придется указать явно, задавая его пакет.

4. Механизм трансляции программы в байт-код. Структура *.class файла.
Сам механизм:
У одного байта существует 256 возможных значений, поэтому всего 256 возможных кодов операций в байт-коде. Код CA_{16} зарезервирован для использования отладчиком и не используется языком, как и коды FE_{16} и FF_{16}, которые зарезервированы для использования виртуальной машиной и отладчиками. Коды в диапазоне CB_{16}—FD_{16} в текущей версии JVM не используются и зарезервированы для будущих дополнений.
Инструкции можно разделить на несколько групп:

- загрузка и сохранение (например, `ALOAD_0`, `ISTORE`),
- арифметические и логические операции (например, `IADD`, `FCMPL`),
- преобразование типов (например, `I2B`, `D2I`),
- создание и преобразование объекта (например, `NEW`, `PUTFIELD`),
- управление стеком (например, `DUP`, `POP`),
- операторы перехода (например, `GOTO`, `IFEQ`),
- вызовы методов и возврат (например, `INVOKESTATIC`, `IRETURN`).

Также есть несколько инструкций, выполняющих специфические задачи, такие как выбрасывание исключений, синхронизация и т. д.
Многие инструкции имеют префиксы или суффиксы, соответствующие их операндам:

Префикс или суффикс	Тип операнда
`I`	integer
`L`	long
`S`	short
`B`	byte
`C`	character
`F`	float
`D`	double
`A`	reference

Например, операция `IADD` — сложение двух целых чисел, в то время как `FADD` — сложение чисел с плавающей точкой.
`Class`-файл состоит из одной структуры `ClassFile`:
ClassFile {
u4 зарезервировано;
u2 младшая_часть_номера_версии;
u2 старшая_часть_номера_версии;
u2 количество_константных_пулов;
cp_info константный_пул[количество_константных_пулов-1];
u2 флаги_доступа;
u2 текущий_класс;
u2 предок;
u2 количество_интерфейсов;
u2 интерфейсы[количество_интерфейсов];
u2 количество_полей;
field_info поля[количество_полей];
u2 количество_методов;
method_info методы[количество_методов];
u2 количество_атрибутов;
attribute_info атрибут[количество_атрибутов];
}
Тут дофига описания возможных значений, но решил оставить
Описание элементов структуры `ClassFile`:

зарезервировано

Представляет собой номер, определяющий формат class-файла. Он имеет значение 0xCAFEBABE.

младшая_часть_номера_версии, старшая_часть_номера_версии
Значения младшая_часть_номера_версии и старшая_часть_номера_версии определяют совместно версию class-файла. Если class-файл имеет старшую часть номера версии равную M и младшую часть номера версии равную m, то общую версию class-файла мы будем обозначать как M.m. Поэтому версии формата class-файла могут быть упорядочены лексикографически, например: 1.5 < 2.0 < 2.1.
Реализация виртуальной машины Java может поддерживать формат class-файла версии v тогда и только тогда, когда v находится в пределах Mi.0 ≤ v ≤ Mj.m.
Значения пределов зависят от номера выпуска виртуальной машины Java.

количество_константных_пулов
Значение элемента количество константных пулов равно на единицу больше количества элементов константный_пул в таблице. Индекс в таблице константных пулов считается действительным, если он больше нуля и меньше значения величины количество_константных_пулов, с учетом исключения для типов long и double.

константный_пул[]
Таблица константных_пулов это таблица структур представляющих различные строковые константы, имена классов и интерфейсов, имена полей и другие константы, на которые есть ссылки в структуре ClassFile и ее подструктурах. Формат каждой следующей структуры константный_пул отделяется от предыдущей «маркерным» байтом. Таблица константных пулов индексируется от 1 до значения количество_константных_пулов-1.

флаги_доступа
Значение элемента флаги_доступа представляет собой маску флагов, определяющих уровень доступа к данному классу или интерфейсу, а также его свойства. Расшифровка установленного значения флагов приведена в таблице 4.1.
Таблица 4.1 Доступ к классу и модификаторы свойств.

Имя флага	Значение	Пояснение
ACC_PUBLIC	0x0001	Объявлен как public; доступен из других пакетов.
ACC_FINAL	0x0010	Объявлен как final; наследование от класса запрещено.
ACC_SUPER	0x0020	При исполнении инструкции invokespecial обрабатывать методы класса предка особым способом.
ACC_INTERFACE	0x0200	Class-файл определяет интерфейс, а не класс.
ACC_ABSTRACT	0x0400	Объявлен как abstract; создание экземпляров запрещено.
ACC_SYNTHETIC	0x1000	Служебный класс. Отсутствует в исходном коде.
ACC_ANNOTATION	0x2000	Объявлен как аннотация
ACC_ENUM	0x4000	Объявлен как перечисление (тип enum)

Класс может быть помечен флагом ACC_SYNTHETIC, что означает, что класс сгенерирован компилятором и отсутствует в исходном коде.
Флаг ACC_ENUM означает, что класс или его предок, объявлены как перечисления.

Если установлен флаг ACC_INTERFACE, то это означает что задано определение интерфейса. Если флаг ACC_INTERFACE сброшен, то в class-файле задан класс, а не интерфейс.
Если для данного class-файла установлен флаг ACC_INTERFACE, то флаг ACC_ABSTRACT должен быть также установлен. При этом такой класс не должен иметь установленных флагов: ACC_FINAL, ACC_SUPER и ACC_ENUM.
Если задана аннотация, то флаг ACC_ANNOTATION должен быть установлен. Если флаг ACC_ANNOTATION установлен, то флаг ACC_INTERFACE должен также быть установлен.
Если флаг ACC_INTERFACE для данного class-файла сброшен, то допустимо устанавливать любые флаги из таблицы 4.1 за исключением флага ACC_ANNOTATION. Однако обратите внимание, что одновременно флаги ACC_FINAL и ACC_ABSTRACT не могут быть установлены.
Флаг ACC_SUPER указывает, какая из двух альтернативных семантик будет подразумеваться в инструкции *invokespecial*, если она будет использована в данном class-файле. Этот флаг устанавливается компилятором, преобразующим исходный код в байт-код виртуальной машины Java.
Все биты элемента access_flags, не обозначенные в таблице 4.1 зарезервированы для будущего использования. Им должны быть присвоены нулевые значения в class-файле, к тому же виртуальная машина Java должна их игнорировать.

текущий_класс
Значение элемента текущий_класс должно быть действительным индексом в таблице константных_пулов[]. Элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info, представляющей собой описание класса или интерфейса, определённого данным class-файлом.
предок
Для класс, значение элемента предок должно быть либо нулем, либо действительным индексом в таблице константных_пулов[]. Если значение элемента предок не нулевое, то элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info, представляющей собой описание непосредственного предка класса, определенного в данном class-файле. Ни непосредственный предок, ни предок произвольной глубины не должны иметь установленным флаг ACC_FINAL элемента access_flags структуры ClassFile.
Если значение элемента предок есть ноль, то class-файл должен описывать класс Object – единственный класс или интерфейс без непосредственных предков.
Для интерфейсов значение элемента предок должно всегда быть действительным индексов в таблице константных_пулов[]. Элемент константный_пул по указанному индексу должен быть структурой CONSTANT_Class_info, представляющей собой описание класса Object.

количество_интерфейсов
Значение количество_интерфейсов говорит о количестве непосредственных интерфейсов предков данного класса или интерфейса.

интерфейсы[]
Каждый элемент в массиве интерфейсы[] представляет собой действительный индекс в таблице константных_пулов[]. Каждый элемент константный_пул соответствующий индексу, взятому из таблицы интерфейсы[i], где $0 \le i <$ interfaces_count, должен быть структурой CONSTANT_Class_info представляющей собой интерфейс – прямой предок данного класса или интерфейса. Индексы интерфейсов в массиве интерфейсы[] должны соответствовать порядку интерфейсов в исходном коде, если читать слева направо.

количество_полей

Элемент количество_полей определяет число элементов field_info в таблице поля[]. Структура field_info описывает все поля, объявленные в данном классе или интерфейсе: как принадлежащие экземпляру, так и принадлежащие классу.

поля[]
Каждое значение в таблице поля[] должно быть структурой field_info, дающей полное описание поля в классе или интерфейсе. Таблица поля[] включает в себя только те поля, которые объявлены в данном классе или интерфейсе. Она не содержит полей, унаследованных от класса-предка или интерфейса предка.

количество_методов
Содержит число элементов method_info в таблице методы[].

методы[]
Каждый элемент в таблице методы[] должен представлять собой структуру method_info, дающую полное описание метода в классе или интерфейсе. Если флаги ACC_NATIVE и ACC_ABSTRACT сброшены в элементе access_flags, то структура method_info содержит байт-код виртуальной машины Java, реализующий метод.
Структура method_info содержит все методы, объявленные в классе или интерфейсе, включая методы экземпляра, методы класса (статические), инициализирующие методы экземпляра, а также инициализирующие методы класса или интерфейса. Таблица методы[] не содержит элементы соответствующие унаследованным методам из классов предков или интерфейсов предков.

количество_атрибутов
Значение соответствует числу элементов в массиве атрибуты[].
атрибуты[]
Каждый элемент таблицы атрибуты[] должен представлять собой структуру attribute_info.
Реализация виртуальной машины Java должна игнорировать без сообщений об ошибках все атрибуты таблицы атрибуты[] структуры ClassFile, которые она не может распознать. Атрибуты не определённые данной спецификацией не должны влиять на семантику class-файла, они могут содержать лишь описательную информацию.

5 исполняемые архивы JAVA. Структура jar-файла. Структура манифеста.
JAR-файл — это Java-архив (**J**ava **AR**chive). Это простой архивный файл, сжатый (иногда с нулевой компрессией) по алгоритму **zip**.
Он был создан для удобства распространения программ, написанных на **Java**. Так как обычная программа содержит сотни, тысячи, а иногда и миллионы файлов. Файл может содержать:

- файл манифеста META-INF/MANIFEST.MF
- java-файлы (исходный код)
- class-файлы
- файлы, необходимые для работы программы: картинки, файлы с настройками и прочее (ресурсы)
- электронные подписи, которые позволяют защитить программу от модификации

Манифест - это текстовый файл формата ключ: значение; он содержит описание jar-файла. В нем могут быть следующие ключи:

- Manifest-Version - версия манифеста
- Main-Class - имя главного класса (должен содержать метод main), такой jar-файл можно запустить как обычный исполняемый файл
- Class-Path - позволяет указать CLASSPATH, который необходим для полноценной работы программы
- SHA-Digest - контрольная сумма определенного файла внутри архива

Кроме **jar**, также существуют другие архивы, связанные с Java:

- **WAR** (**W**eb **A**pplication a**R**chive) - содержит в себе приложение для веба
- **EAR** (**E**nterprise **A**pplication a**R**chive) - содержит в себе энтерпрайз приложение (обычно из нескольких модулей)
- **APK** (**A**ndroid a**P**plication pac**K**age) - содержит в себе приложение для Android

6 ООП на JAVA

1. Все является объектом

Все данные программы хранятся в объектах. Каждый объект создается (есть средства для создания объектов), существует какое-то время, потом уничтожается.

2. Программа есть группа объектов, общающихся друг с другом

Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически это происходит при вызове метода данного объекта.

3. Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных.

Объект хранит некоторые данные. Эти данные — это другие объекты, входящие в состав данного объекта и/или данные элементарных типов, такие как целое, вещественное, символ, и т.п.

4. Каждый объект имеет свой тип (класс)

Т.е. в объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы. Мы должны сначала описать (создать) тип (класс) объекта, указав в этом описании из каких элементов (полей) будут состоять объекты данного типа. После этого мы можем создавать объекты этого типа. Все они будут состоять из одних и тех же элементов (полей).

5. Все объекты одного и того же типа могут получать одни и те же сообщения

Кроме описания структуры данных, входящих в объекты данного типа, описание типа содержит описание всех сообщений, которые могут получать объекты данного типа (всех методов данного класса). Более того, в описании типа мы должны задать не только перечень и сигнатуру сообщений данного типа, но и алгоритмы их обработки.

Java — полностью объектно-ориентированный язык, поэтому, как мы уже отмечали, все действия, выполняемые программой, находятся в методах тех или иных классов.
Описание класса начинается с ключевого слова **`class`**, после которого указывается идентификатор — имя класса. Затем в фигурных скобках перечисляются атрибуты и методы класса. Атрибуты в языке Java называются *полями* (в дальнейшем мы будем использовать это наименование). Поля и методы называются *членами класса*.
Поля описываются как обычные переменные.
Конструктор — это особенный метод класса, который вызывается автоматически в момент создания объектов этого класса. Имя конструктора совпадает с именем класса. Есть перегрузка конструкторов.
Наследование — это отношение между классами, при котором один класс расширяет функциональность другого. Это значит, что он автоматически перенимает все его поля и методы, а также добавляет некоторые свои.
Для того, чтобы один класс был потомком другого, необходимо при его объявлении после имени класса указать ключевое слово **`extends`** и название суперкласса.
Например:

```
class Dalmatian extends Dog {
// дополнительные поля и методы
...
}
```

Если ключевое слово **`extends`** не указано, считается, что класс унаследован от универсального класса Object.
В Jave нет множественного наследования. Если возникает такая потребность, она реализуется за счет имплементацию интерфейсов.

Доступ к любому члену класса — полю или методу — может быть ограничен. Для этого перед его объявлением ставится ключевое слово **`private`**. Оно означает, что к этому члену класса нельзя будет обратиться из методов других классов.
Ключевое слово **`public`** может употребляться в тех же случаях, но имеет противоположный смысл. Оно означает, что данный член класса является доступным. Если это поле, его можно использовать в выражениях или изменять при помощи присваивания, а если метод, его можно вызывать.
Ключевое слово **`protected`** означает, что доступ к полю или методу имеет сам класс и все его потомки.
Если при объявлении члена класса не указан ни один из перечисленных модификаторов, используется модификатор по умолчанию (**`default`**). Он означает, что доступ к члену класса имеют все классы, объявленные в том же пакете.
Возможность скрывать поля и методы класса используется для того, чтобы уберечь программиста от возможных ошибок, сделать классы понятнее и проще в использовании. При этом реализуется принцип инкапсуляции.
Инкапсуляция означает сокрытие деталей реализации класса. Класс разделяется на две части: внутреннюю и внешнюю. Внешняя часть (интерфейс) тщательно продумывается исходя из того, каким образом могут взаимодействовать с объектами данного класса другие объекты программы. Внутренняя часть закрыта от посторонних, она нужна только самому классу для обеспечения правильной работы открытых методов.
В одном классе можно создать несколько методов с одним и тем же именем, различающихся по своим параметрам. Этот прием называется перегрузкой методов. Когда один из этих методов будет вызван, произойдет сопоставление переданных ему параметров (их количества и типов) с параметрами всех методов класса с таким именем. Если подходящий метод будет найден, выполнится именно он.
Полиморфизм — это возможность класса выступать в программе в роли любого из своих предков, несмотря на то, что в нем может быть изменена реализация любого из методов. Изменить работу любого из методов, унаследованных от класса-предка, класс-потомок может, описав новый метод с точно таким же именем и параметрами. Это называется *переопределением*. При вызове такого метода для объекта класса-потомка будет выполнена новая реализация.
Если в классе не описан ни один конструктор, для него автоматически создается конструктор по умолчанию. Этот конструктор не имеет параметров, все что он делает — это вызывает конструктор без параметров класса-предка.
Вызов конструктора суперкласса
Ключевое слово **`super`** означает суперкласс.
Вызов конструктора суперкласса должен происходить в самом начале конструктора.
Вместо вызова конструктора суперкласса можно вызвать один из конструкторов того же самого класса. Это делается с помощью ключевого слова **`this`**`()` — с параметрами в скобках, если они нужны.
Если в начале конструктора нет ни вызова **`this`**`()`, ни вызова **`super`**`()`, автоматически происходит обращение к конструктору суперкласса без аргументов.
Объект класса-потомка можно присвоить переменной типа класса-предка. При этом Java производит автоматическое преобразование типа, называемое расширением. *Расширение* — это переход от более конкретного типа к менее конкретному. Переход от **`byte`** к **`int`** — это тоже расширение.
Внутри фигурных скобок, заключающих в себе тело одного класса, помимо описания его полей и методов можно поместить описание другого класса. Он будет называться *вложенным классом*.
Класс можно объявить внутри метода другого класса. В этом случае класс "виден" только внутри метода (за пределами метода нельзя объявить переменную типа этого класса).
Анонимным классом называется класс, не имеющий имени. Очевидно, если у класса имени нет, к нему нельзя обратиться из программы. Точнее, это можно сделать только один раз — в том месте, где класс объявляется.
Описание анонимного класса начинается с вызова конструктора его суперкласса, после чего в фигурных скобках описывается тело класса.
Анонимные классы используются в том случае, когда нужен единственный объект такого класса на всю программу.

Модификатор static

Любой член класса можно объявить статическим, указав перед его объявлением ключевое слово **`static`**. Статический член класса «разделяется» между всеми его объектами.

Для поля это означает, что любое изменение его значения, сделанное одним из объектов класса, сразу же «увидят» все остальные объекты.

Метод, объявленный с модификатором **`static`**, "дает обещание" не изменять никаких полей класса, кроме статических.

Для обращения к статическому члену класса можно использовать любой объект этого класса. Более того, это обращение можно осуществлять даже тогда, когда не создано ни одного такого объекта. Вместо имени объекта можно просто указывать имя класса.

Модификатор final

Любое поле класса можно объявить неизменяемым, указав перед его объявлением ключевое слово **`final`**. Неизменяемому полю можно присвоить значение только один раз (обычно это делается сразу при объявлении). Константы в языке Java очевидным образом описываются путем совмещения модификаторов **`static`** и **`final`**.