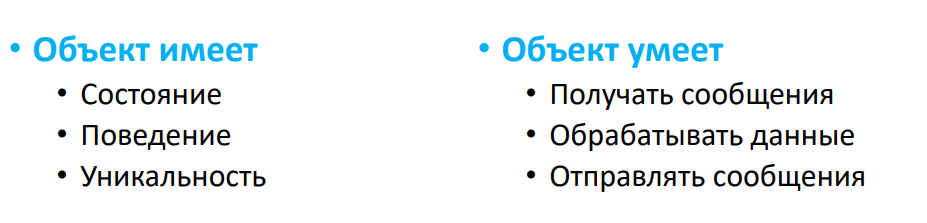
1. **Объектно-ориентированное программирование: элементы классов, принципы, достоинства и недостатки.**

ООП – это парадигма программирования, в которой базовым является понятие объекта



Программа в ходе работы представляет собой набор взаимодействующих объектов

Принципы ООП:

* Абстракция
* Инкапсуляция - объединение данных и методов их обработки в одну сущность, приводящее к сокрытию реализации класса и отделению его внутреннего представления от внешнего
* Наследование - отношение между классами, при котором один класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов
* Полиморфизм - способность объекта соответствовать во время выполнения двум или более возможным типам

Достоинства ООП:

* Упрощение разработки
* Разделение функциональности, локализация кода, инкапсуляция
* Возможность создания расширяемых систем
* Обработка разнородных структур данных, изменение поведения на этапе выполнения, работа с наследниками
* Легкость модернизации с сохранением совместимости

Недостатки ООП:

* Неэффективность на этапе выполнения
* Неэффективность в смысле распределения памяти
* Излишняя избыточность
* Психологическая сложность проектирования
* Техническая сложность проектирования и документирования

Класс можно рассматривать как шаблон, а объект – экземпляр этого класса.

2. **Особенности языка Java. Причины возникновения и область современного применения.**

* Кросс-платформенность
* Объектная ориентированность
* Привычный синтаксис (С++)
* Встроенная модель безопасности
* Ориентация на интернет-задачи, распределенные приложения
* Динамичность, легкость развития
* Легкость в освоении

Java создавался, как язык, позволяющий приложениям корректно работать на различных платформах без необходимости дополнительных преобразований, изменений кода.

Сегодня он является основой для всех приложений на Android, широко используется в облачных технологиях, обработке больших данных и создании микросервисов с помощью фреймворков вроде Spring.

3. **Модули компиляции и пакеты. Имена и их пространства. Правила именования.**

Модуль компиляции хранится в .java файле и является единичной порцией входных данных для компилятора

Состоит из:

* объявления пакета (указывает принадлежность типов в модуле к пакету)

package mypackage;

* выражений импортирования (добавляют другие классы в область видимости)

import java.net.Socket;

import java.io.\*;

* объявлений верхнего уровня (описания классов и интерфейсов)

Пакеты - способ логической группировки классов; комплект ПО, могущий распространяться независимо и применяться в сочетании с другими пакетами.

Членами пакетов являются:

* классы
* интерфейсы
* вложенные пакеты
* дополнительные файлы ресурсов

Позволяют группировать взаимосвязанные классы и интерфейсы в единое целое. Способствуют созданию пространств имен, позволяющих избежать конфликтов идентификаторов, относящихся к различным типам. Обеспечивают дополнительные средства защиты элементов кода.

Пакеты могут быть реализованы:

* в виде структуры каталогов с файлами классов
* в виде jar-архива

Путь к используемым пакетам указывается:

* непосредственно при запуске JVM с помощью ключей
* через переменную окружения CLASSPATH (по умолчанию CLASSPATH="")

Имена задаются посредством идентификаторов, указывают на компоненты программы

* Пространства имен
* пакеты
* типы
* поля
* методы
* локальные переменные и параметры
* метки

Имена типов бывают

* составные (java.lang.Double)
* простые (Double)

Правила именования:

* Пакеты - java.lang, javax.swing, ru.ssau.tk
* Типы - Student, ArrayIndexOutOfBoundException, Cloneable, Runnable, Serializable
* Поля - value, enabled, distanceFromShop
* Методы - getValue, setValue, isEnabled, length, toString
* Поля-константы - PI, SIZE\_MIN, SIZE\_MAX, SIZE\_DEF
* Локальные переменные

4. **Описание классов. Модификаторы класса. Модификаторы доступа элементов класса.**

Класс может содержать:

* поля
* методы
* вложенные классы и интерфейсы

public class Body {  
 public long idNum;  
 public String name;  
 public Body orbits;  
 public static long nextID = 0;  
}

Модификаторы класса:

* public - Признак общедоступности класса (класс виден вне пакета)
* abstract - Признак абстрактности класса (класс не полностью реализует поведение)
* final - Завершенность класса (класс не допускает наследования)
* strictfp - Повышенные требования к операциям с плавающей точкой (результаты операций одинаковые на различных платформах)

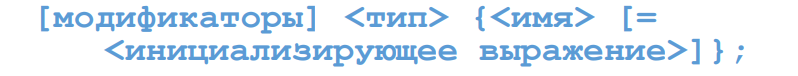
Модификаторы доступа элементов класса:

* public: Элемент доступен отовсюду, где виден сам содержащий его класс.
* protected: Элемент доступен внутри своего пакета, а также в классах-наследниках (даже если они находятся в других пакетах).
* package-private (по умолчанию): Элемент доступен только внутри своего пакета. Самый частый модификатор для внутренней реализации.
* private: Элемент доступен только внутри самого класса, в котором он объявлен. Основа инкапсуляции.

5. **Описание полей классов. Модификаторы полей.**

По сути являются переменными: обладают типом, именем и значением

Объявление поля:



Примеры

double sum = 2.5 + 3.7, a;  
public double val = sum + 2 \* Math.sqrt(2);

Если поле явно не инициализируются, ему присваивается значение по умолчанию его типа (0, false или null)

Модификаторы полей класса

* static поле статично (принадлежит контексту класса, а не объекта)
* final поле не может изменять свое значение после инициализации
* transient поле не сериализуется (влияет только на механизмы сериализации)
* volatile усиливает требования к работе с полем в многопоточных программах

6. **Описание методов классов. Модификаторы методов.**

По сути являются функциями: обладают именем, параметрами и возвращаемым значением.

Объявление метода:



Тело метода состоит из набора инструкций:

class Primes {  
 static int nextPrime(int current) {  
 <Вычисление простого числа в теле метода>  
 }  
}

Модификаторы доступа:

* abstract - абстрактность метода (метод объявляется, но тело при этом не описывается)
* static - статичность метода (метод принадлежит контексту класса, а не объекта)
* final - завершенность метода (метод не может быть переопределен при наследовании)
* synchronized - синхронизированность метода (особенности вызова метода в многопоточных приложениях)
* native - «нативность» метода (тело метода не описывается, при вызове вызывается метод из native-библиотеки)
* strictfp - повышенные требования к операциям с плавающей точкой (результаты операций одинаковые на различных платформах)

7. **Особенности методов, допустимые действия в методах.**

Для нестатических методов вызов через ссылку на объект reference.method();

Для статических методов вызов через имя типа, через ссылку на объект

ClassName.staticMethod();  
reference.staticMethod();

Наличие круглых скобок при вызове обязательно, т.к. они являются оператором вызова метода.

На время выполнения метода управление передается в тело метода.

Возвращается одно значение

return someValue;

Аргументы передаются по значению, т.е. значения параметров копируются в стек:

* для примитивных типов копируются сами значения
* для ссылочных типов копируется значение ссылки

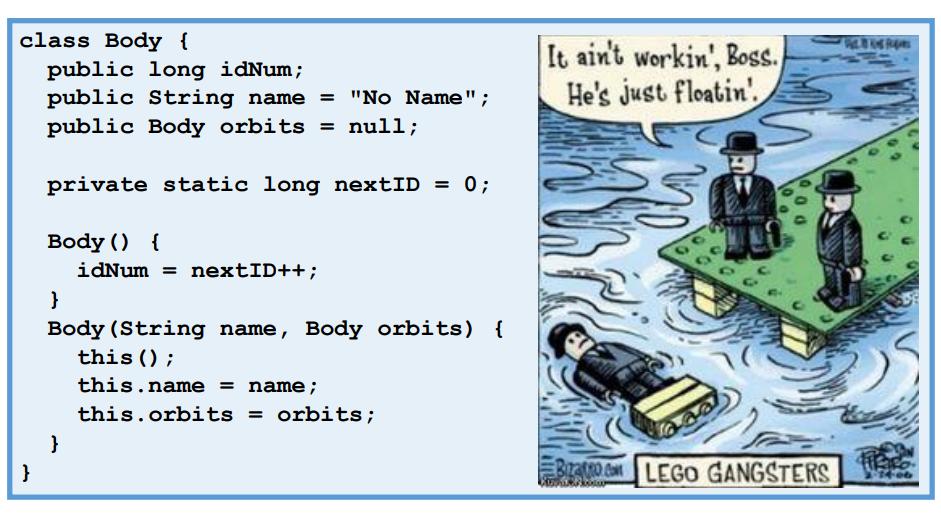
Перегруженными являются методы с одинаковыми именами и различными по типу списками параметров

8. **Конструкторы в классах.**

Память для объекта выделяет оператор new. Конструкторы предназначены для формирования начального состояния объекта. Правила написания конструктора сходны с правилами написания методов. Имя конструктора совпадает с именем класса

Для конструкторов разрешено использование только модификаторов доступа. При написании конструктор не имеет возвращаемого типа. Оператор возврата return прекращает выполнение текущего конструктора. Конструкторы могут быть перегружены. Конструкторы могут вызывать друг друга с помощью ключевого слова this в первой строке конструктора

Если в классе явно не описан ни один конструктор, автоматически создается т.н. конструктор по умолчанию, не имеющий параметров. Если в классе описан хотя бы один конструктор, то автоматически конструктор по умолчанию не создается. Также конструктором по умолчанию называют конструктор, не имеющий параметров



9. **Автоматическая сборка мусора.**

Явное уничтожение объектов является серьёзным источником ошибок: если после вызова деструктора где-то осталась ссылка (указатель) на объект, ее использование и приведет к возникновению ошибки. В Java деструкторов нет, вместо них применяется механизм автоматической сборки мусора. Задачу высвобождения ресурсов обычно выполняет простой метод объекта с говорящим названием (close(), dispose() и т.д.)

В случае нехватки памяти для создания очередного объекта виртуальная машина находит недостижимые объекты и удаляет их. Процесс сборки мусора можно инициировать принудительно. Для явного удаления объекта следует утратить все ссылки на этот объект и инициировать сбор мусора. Взаимодействие со сборщиком осуществляется через системные классы java.lang.System и java.lang.Runtime.

10. **Инициализирующие блоки и выражения классов. Порядок создания экземпляра класса.**

Если некоторые действия по инициализации должны выполняться в любом варианте создания объекта, удобнее использовать блоки инициализации. Тело блока инициализации заключается в фигурные скобки и располагается на одном уровне с полями и методами. При создании объекта сначала выполняются инициализирующие выражения полей и блоки инициализации (в порядке их описания в теле класса), а потом тело конструктора

class Body {  
 public long idNum;  
 public String name = "No Name";  
 public Body orbits = null;  
 private static long nextID = 0;  
 {  
 idNum = nextID++;  
 }  
 Body(String name, Body orbits) {  
 this.name = name;  
 this.orbits = orbits;  
 }  
}

Статический блок инициализации выполняет инициализацию контекста класса. Вызов статического блока инициализации происходит в процессе загрузки класса в виртуальную машину

class Primes {  
 static int[] knownPrimes = new int[4];  
 static {  
 knownPrimes[0] = 2;  
 for (int i=1; i<knownPrimes.length; i++)  
 knownPrimes[i] = nextPrime(i);  
 }  
*//nextPrime() declaration etc.*}

Имеем класс Car от него наследуем Truck

1) Первое что произойдет — проинициализируются статические переменные класса Car. Да-да, именно класса Car, а не Truck. Статические переменные инициализируются еще до вызова конструкторов, и начинается это в классе-родителе.

2) После инициализации статических переменных класса-предка инициализируются статические переменные класса-потомка.

3) Третьими по счету будут инициализированы нестатические переменные класса-предка. Как видишь, наследование заметно усложняет процесс создания объекта, но тут уж ничего не поделаешь: некоторые вещи в программировании придется просто запомнить :)

4) Наконец, дело дошло до конструкторов! Точнее, до конструктора базового класса. Начало его работы — четвертый

5) Теперь пришла очередь инициализации нестатических полей класса-потомка, то есть нашего класса Truck. Поля класса, объект которого мы создаем, инициализируются только в пятую очередь! Удивительно, но факт :)

6) Вызывается конструктор дочернего класса Truck.

*Мой опыт:*

1. *Статические блоки класса предка*
2. *Статические блоки класса наследника*
3. *Вызывается конструктор наследника он запускает инициализацию полей и блоков инициализации класса предка в порядке следования, потом сам конструктор.*
4. *Инициализация полей и блоков инициализации класса предка в порядке следования*
5. *Инициализация полей и блоков инициализации класса наследника в порядке следования*

11. **Точка входа программы. Порядок запуска программы. Способы запуска программы.**

Точка входа — это строго определенное место в коде, с которого начинает выполняться программа. В языках семейства C/Java эта концепция реализована как специальный метод.

В Java: Точкой входа является статический метод main с заранее заданной сигнатурой.

Сигнатура метода:

* public static void main(String[] args)

public: Метод должен быть доступен извне для JVM.

static: Метод принадлежит классу, а не объекту. JVM запускает его, не создавая экземпляр класса.

void: Метод не возвращает значение.

String[] args: Параметр для передачи аргументов командной строки в программу. args (arguments) — это массив строк.

Пример:

* public class HelloWorld {  
   public static void main(String[] args) {  
   System.out.println("Программа запущена!");  
   if (args.length > 0) {  
   System.out.println("Первый аргумент: " + args[0]);  
   }  
   }  
  }
* Важно: Класс, содержащий метод main, может иметь любое имя, но сигнатура метода должна быть соблюдена точно. JVM ищет именно метод с такой сигнатурой.

2. Порядок запуска программы (Program Startup Sequence)

Когда вы даете команду на выполнение программы (например, java HelloWorld), происходит следующая последовательность событий:

1. Загрузка класса: Виртуальная машина Java (JVM) находит и загружает в память класс, имя которого вы указали (например, HelloWorld), с помощью ClassLoader.
2. Проверка и инициализация: JVM проверяет байт-код на корректность и инициализирует статические поля и выполняет статические блоки данного класса. Это происходит единожды, при первой загрузке класса.
3. Поиск точки входа: JVM ищет в загруженном классе метод public static void main(String[] args). Если метод не найден — программа завершается с ошибкой Error: Main method not found.
4. Выполнение метода main: JVM передает управление в найденный метод main. Выполнение начинается с первой инструкции этого метода.
5. Работа программы: Внутри метода main и из методов, вызванных из него, создаются объекты, выполняются вычисления, работа с вводом/выводом и т.д.
6. Завершение программы: Программа завершается, когда:
   * Завершился поток выполнения метода main.
   * Был вызван метод System.exit().
   * Произошло необработанное исключение, завершившее поток.

Способы запуска программы (Ways to Launch a Program)

Существует несколько практических способов запустить Java-программу:

1. Из командной строки (терминала) — самый базовый способ:

* Скомпилировать: javac HelloWorld.java (создаст HelloWorld.class)
* Запустить: java HelloWorld
* С аргументами: java HelloWorld argument1 "argument 2"

2. Из среды разработки (IDE):

* IntelliJ IDEA, Eclipse, NetBeans: Обычно есть кнопка Run ▶ (зеленая стрелка) или сочетание клавиш (Shift+F10). IDE сама выполняет компиляцию и запуск. Часто можно настроить конфигурацию запуска, указав аргументы командной строки, переменные окружения и т.д.

3. Используя инструменты сборки (Build Tools):

* Maven: mvn exec:java -Dexec.mainClass="com.example.HelloWorld"
* Gradle: Можно настроить задачу run в build.gradle. Запуск: gradle run.

4. Запуск упакованного приложения (JAR, WAR):

* JAR-архив: Если в манифесте JAR-файла (META-INF/MANIFEST.MF) указан главный класс:

Main-Class: com.example.HelloWorld

То запуск осуществляется командой: java -jar myapp.jar

* WAR-архив: Запускается путем развертывания на сервере приложений (Tomcat, Jetty, WildFly). Точкой входа в этом случае являются сервлеты или методы, аннотированные @PostConstruct в managed-бинах.

5. Использование модулей JPMS (Java Platform Module System, Java 9+):

* Запуск из корня модуля: java --module-path mods -m com.example/com.example.MainClass

6. Запуск в контейнере (Docker):

* Программа упаковывается в Docker-образ вместе с JVM. Запуск производится командой docker run my-java-app. Это современный стандарт для развертывания.

12. **Инкапсуляция и её реализация.**

Изначальное значение слова «*инкапсуляция*» в программировании — **объединение данных и методов работы с этими данными в одной упаковке** («капсуле»). В Java в роли упаковки-капсулы выступает *класс*. Класс содержит в себе и *данные* (поля класса), и *методы* для работы с этими данными.

Про сокрытие. Все как в реальной жизни: пользователю предоставлен интерфейс (методы). Если ему нужно, чтобы автомобиль в программе выполнил действие, достаточно вызвать нужный метод. А уж что там происходит внутри этих методов — информация лишняя, главное, чтобы все работало как надо.

Пример:

class Rectangle {  
 private int width, height;  
 private int area;  
 public int getWidth() {  
 return width;  
 }  
 public int getHeight() {  
 return height;  
 }  
 public void setWidth(int width) {  
 this.width = width;  
 area = width \* height;  
 }  
 public void setHeight(int height) {  
 this.height = height;  
 area = width \* height;  
 }  
 public int getArea() {  
 return area;  
 }  
}

Использование инкапсуляции дает нам несколько важных преимуществ:

1. **Контроль за корректным состоянием объекта.** Примеры этому были выше: благодаря сеттеру и модификатору private, мы обезопасили нашу программу от котов с весом 0.
2. **Удобство для пользователя за счет интерфейса.** Мы оставляем «снаружи» для доступа пользователя только методы. Ему достаточно вызвать их, чтобы получить результат, и совсем не нужно вникать в детали их работы.
3. **Изменения в коде не отражаются на пользователях.** Все изменения мы проводим внутри методов. На пользователя это не повлияет: он как писал auto.gas() для газа машины, так и будет писать. А то, что мы поменяли что-то в работе метода gas() для него останется незаметным: он, как и раньше, просто будет получать нужный результат.

**13. Кодировка, структура исходного кода (комментарии, пробелы и лексемы). Виды лексем.**

Java ориентирован на Unicode. Первые 128 символов почти идентичны набору ASCII. Символы Unicode задаются с помощью escapeпоследовательностей \u262f, \uu2042, \uuu203d. Java чувствителен к регистру!

Исходный код разделяется на:

* Пробелы
* Комментарии
* Лексемы

Комментарии:

* // Комментарий Символы после // и до конца текущей строки игнорируются
* /\* Комментарий \*/ Все символы, заключенные между /\* и \*/, игнорируются
* /\*\* Комментарий \*/ Комментарии документирования

Комментарии документирования:

* Начинаются с /\*\*, заканчиваются \*/
* В строках начальные символы \* и пробелы перед ними игнорируются
* Допускают использование HTML-тэгов, кроме заголовков
* Специальные тэги @see, @param, @deprecated

Виды лексем:

* Идентификаторы
* Служебные слова class, public, const, goto, и т.д.
* Литералы
* Разделители { } [ ] ( ) ; . ,
* Операторы = > < ! ? : == && ||

14. Типы данных и их классификация. Примитивные типы.

Ссылочные (объектные)

* Предназначены для работы с объектами
* Переменные содержат ссылки на объекты
* Ссылка – это не указатель!
* Тип переменной определяет контракт доступа к объекту

Примитивные (простые)

* Предназначены для работы со значениями естественных, простых типов
* Переменные содержат непосредственно значения

Тип данных определяется следующими характеристиками:

* Множество значений
  + для примитивных типов – числа, не выходящие за диапазон типа
  + для ссылочных типов – ссылки на объекты, контракт которых включает в себя контракт, определяемый типом ссылки
* Возможные операции со значениями
  + для примитивных типов – операторы
  + для ссылочных типов – действия, входящие в контракт типа (вызов методов и обращение к полям), и операторы
* Форма хранения и представления
  + форма хранения определяется реализацией JVM
  + JVM гарантирует одинаковое представление, не зависящее от реализации

Примитивные типы данных:



15. **Переменные, константы и литералы.**

Переменные - именованные участки памяти, способные содержать значения определенного типа Могут быть объявлены в различных частях кода

* поля объектов
* поля классов (статические поля)
* параметры методов
* локальные переменные методов и блоков инициализации

Объявление переменной состоит из наименования типа, идентификатора и инициализации. Область видимости переменной определяется местом ее объявления. Локальные переменные должны быть инициализированы перед их использованием.

Констант как особого вида переменных и полей в Java нет. Если необходима константа в методе, то переменная при объявлении снабжается модификатором final. Так можно сделать даже с параметром метода. Это не имеет особого смысла, но иногда это явно требуется в многопоточных программах.

Если необходима общедоступная константа, то создаётся поле с модификаторами public static final. Такие поля и называют константами (условно). Их имена записываются заглавными буквами. Обращение чаще всего происходит через имя класса, например BigInteger.ZERO

Литералы:

* Булевы true false
* Символьные 'a' '\n' '\\' '\377' '\u0064'
* Целочисленные 29 035 0x1D 0X1d 0xffffL - По умолчанию имеют тип int
* Числовые с плавающей запятой 1. .1 1e1 1e-4D 1e+5f - По умолчанию имеют тип double
* Строковые "Это строковый литерал" ""

**16. Арифметические и побитовые операторы примитивных числовых типов.**

Арифметические операторы

* + – сложение двух значений
* - – вычитание второго значения из первого
* \* – умножение двух значений
* / – деление первого значения на второе
* % – остаток от деления первого значения на второе

Результат имеет тип, совпадающий с «наиболее широким» типом из типов операндов, но не меньше, чем int.

Инкременты и декременты – соответственно, увеличивают и уменьшают значение на 1

* Постфиксная форма: i++, i— результатом оператора является прежнее (неизмененное) значение
* Префиксная форма: ++i, --I результатом оператора является новое значение

Унарные + и -

* Аналогичны случаю, когда первый операнд равен 0. Если знак + или - находится перед литералом, он может трактоваться как часть литерала

Побитовые операторы

*Логические операторы*

• & – «и» (and)

1 & 3 -> 1

00000001 & 00000011 -> 00000001

• | – «или» (or)

1 | 3 -> 3

00000001 | 00000011 -> 00000011

• ^ – «исключающее или» (xor)

1 ^ 3 -> 2

00000001 ^ 00000011 -> 00000010

• ~ – побитовое отрицание

~ 1 -> -2

~00000001 -> 11111110

• Вычисления производятся в типе int либо long

*Операторы сдвига*

• << – сдвиг влево

1 << 2 -> 4

00000001 << 2 -> 00000100

• >> – арифметический сдвиг вправо

4 >> 2 -> 1

00000100 >> 2 -> 00000001

-1 >> 2 -> -1

11111111 >> 2 -> 11111111

• >>> – логический сдвиг вправо

4 >>> 2 -> 1

00000100 >>> 2 -> 00000001

-1 >>> 2 -> 1073741823

11111111 >>> 2 -> 00111111 11111111 11111111 11111111

• Вычисления производятся в типе int либо long

17. **Операторы сравнения, логических действий, присваивания и приведения типов для примитивных типов.**

Операторы сравнения примитивных типов:

* > и < – строгое сравнение
* >= и <= – нестрогое сравнение
* == – определение равенства
* != – определение неравенства
* Результат – логическое значение: true или false
* Сравнение проводится в наиболее широком типе из типов операндов

Операторы примитивного логического типа:

* == – определение равенства
* != – определение неравенства
* ! – отрицание
* & – логическое «и» (and)
* | – логическое «или» (or)
* ^ – логическое «исключающее или» (xor)
* && – условное «и» (может не вычислять второй операнд)
* || – условное «или» (может не вычислять второй операнд)

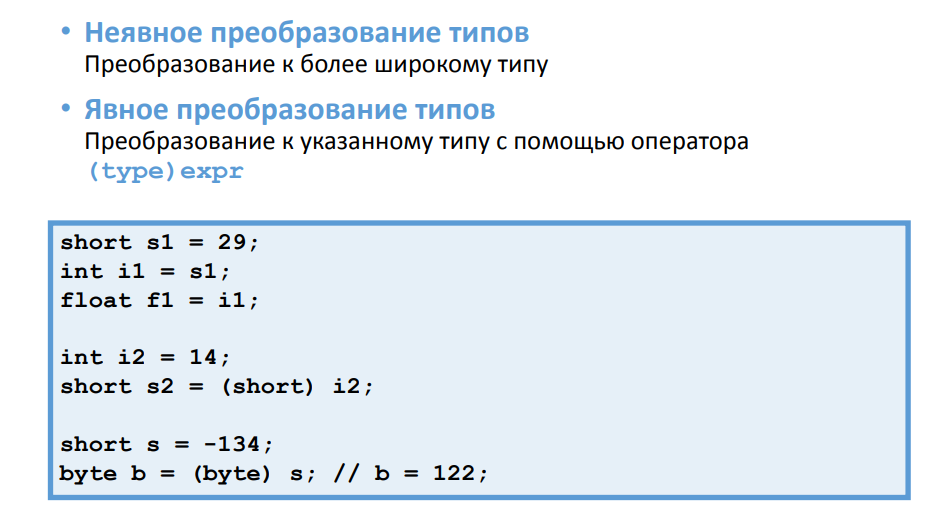
Операторы присваивания:

• = – простое присваивание. Тип выражения справа должен допускать присваивание в переменную слева

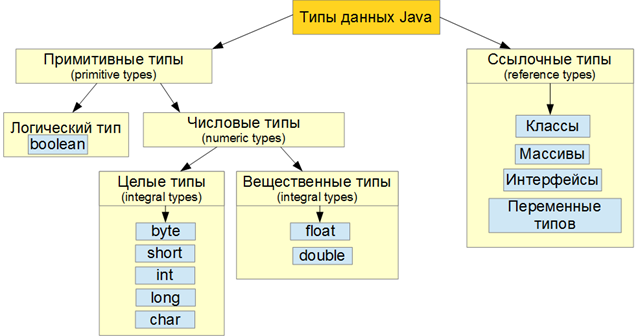
• +=, -=,\*=, /=, %=, >>=, <<=, >>>=, &=,^=, |=. Присваивание с действием. Выражение a ?= b эквивалентно a = a ? b, но выполняется быстрее. Типы операндов должны позволять совершить операцию

Операторы приведения:

* Неявное преобразование типов. Преобразование к более широкому типу
* Явное преобразование типов. Преобразование к указанному типу с помощью оператора (type)expr



18. **Типы данных и их классификация. Ссылочные типы и их литералы.**



Литерал — это фиксированное значение, записанное непосредственно в исходном коде. Оно известно на этапе компиляции и не требует вычисления

Единственный литерал, связанный с объектом — это null.

В соответствии с примитивами, литералы также делятся на разделы:

1. Числовые:
   1. Целочисленные;
   2. С плавающей точкой;
2. Строковые;
3. Символьные;
4. Логические.

В Java нет ссылочных литералов в классическом понимании, кроме null.

19. **Операторы для ссылочных типов.**

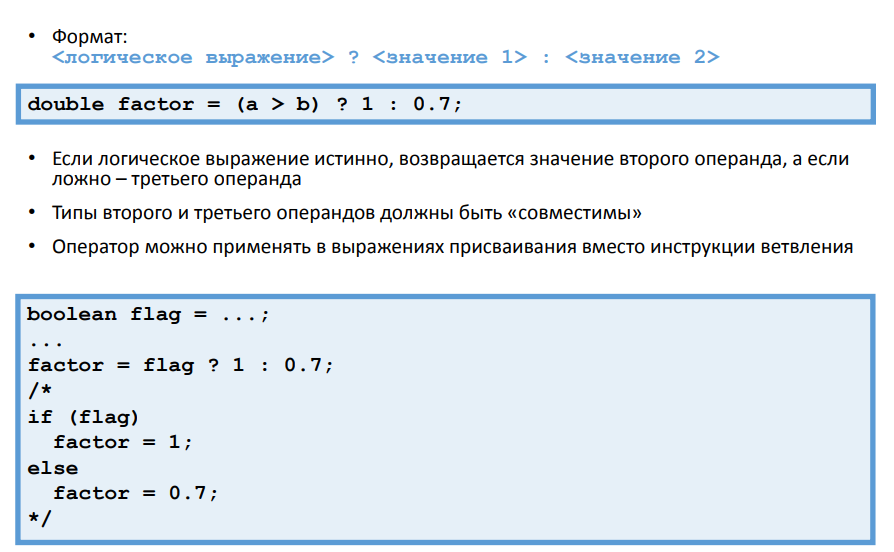
* new – создание объекта класса
* = – присвоение ссылки. Тип выражения справа должен допускать присвоение в тип переменной слева
* == и != – сравнение ссылок. Сравниваются только ссылки, а не состояние объектов!
* •. – разыменование ссылки

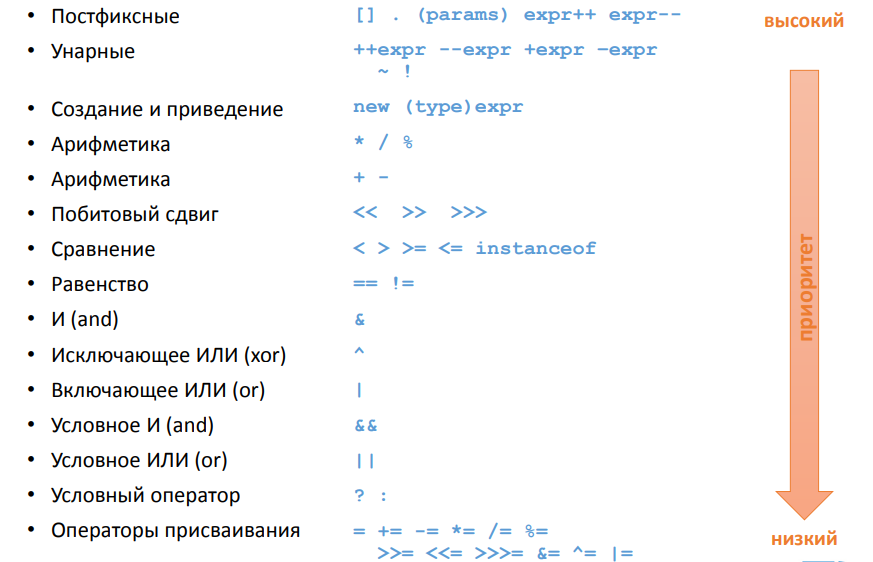
reference.method()

reference.field

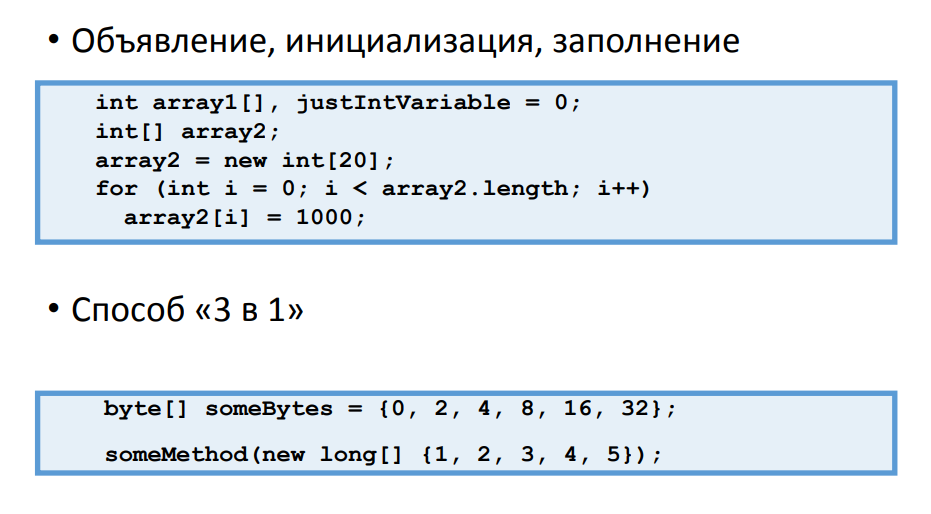
* () – вызов метода. У любого объекта можно вызвать методы, объявленные в классе Object

20. **Оператор ветвления. Приоритеты операторов.**



Приоритет операторов 

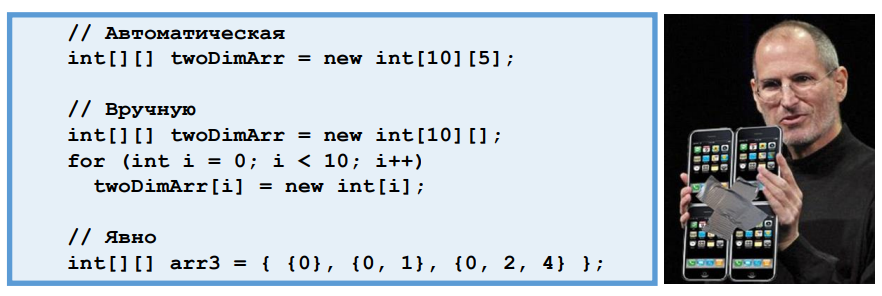
21. **Массивы: особенности, разновидности, объявление и инициализация.**

Массив – упорядоченный набор элементов одного типа. Элементами могут быть значения простых и ссылочных типов. Массивы сами по себе являются объектами и наследуют от класса Object. Доступ к элементам по целочисленному индексу с помощью оператора []

Работа с одномерными массивами: форма объявления ссылки на массив с квадратными скобками

после типа элемента является более предпочтительной. Объект массива создается с помощью оператора new. Массив при этом заполняется значениями по умолчанию для типа его элементов (0, false или null). Нумерация в массивах начинается с 0. Длина массива хранится в публичном неизменяемом поле length. Изменить длину массива после создания его объекта нельзя

Многомерные массивы: состоят из одномерных массивов, элементами которых являются ссылки на массивы меньшей размерности. При создании объекта необязательно указывать все

Размерности. Массив необязательно должен быть «прямоугольным». 

22. **Инструкции и их разновидности. Блоки, ветвления, блок переключателей.**

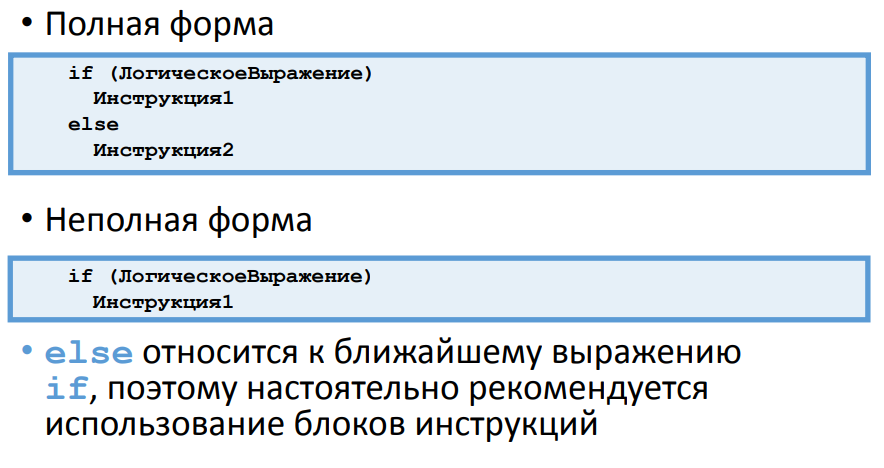
Инструкция - описание одного действия. «Заканчивается» знаком ;. Тела методов, конструкторов и блоков инициализации состоят из набора инструкций.

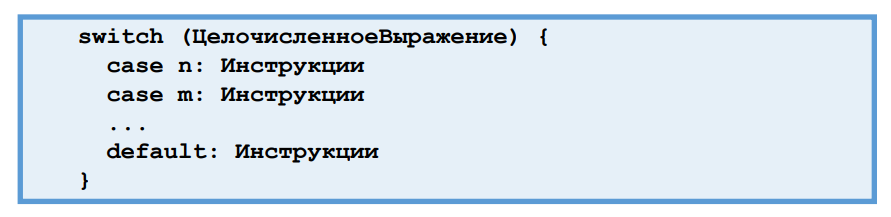
Виды инструкций:

* Выражения присваивания
* Префиксные и постфиксные формы выражений с операторами инкремента и декремента
* Конструкции вызова методов
* Выражения создания объектов
* Составные инструкции
* Управляющие порядком вычислений

Блок - Составная инструкция. Может использоваться в любом месте, где допускается инструкция. Определяет область видимости локальных переменных: объявленная внутри блока переменная не видна за его пределами.



Ветвление:

Блок переключателей: 

Для типов char, byte, short, int.(РАБОТАЕТ И ДЛЯ char, byte, short, int, их обертки, enum, String, Object).

Выполняются инструкции, расположенные за меткой case, предложение которой совпало со значением параметра блока переключателей.

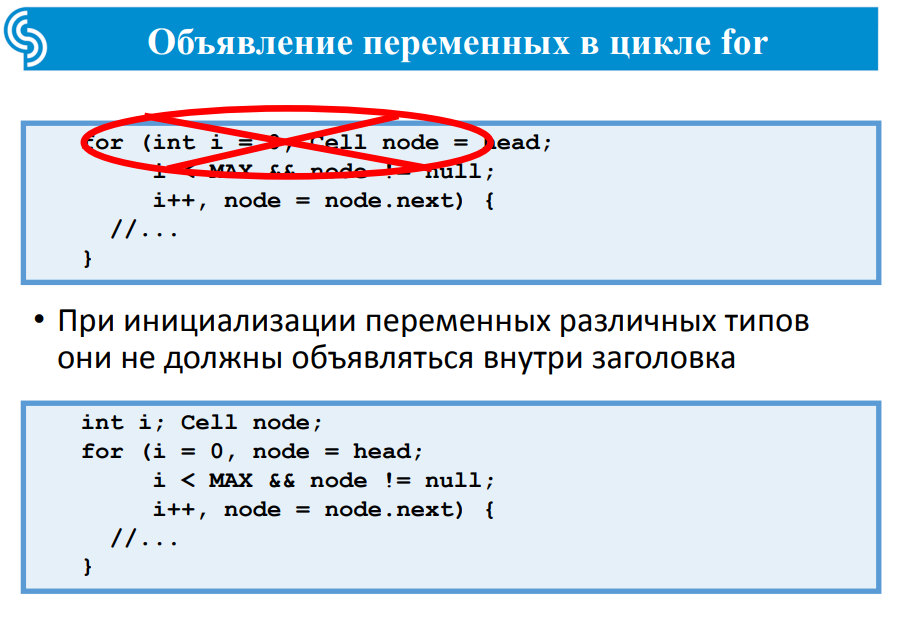
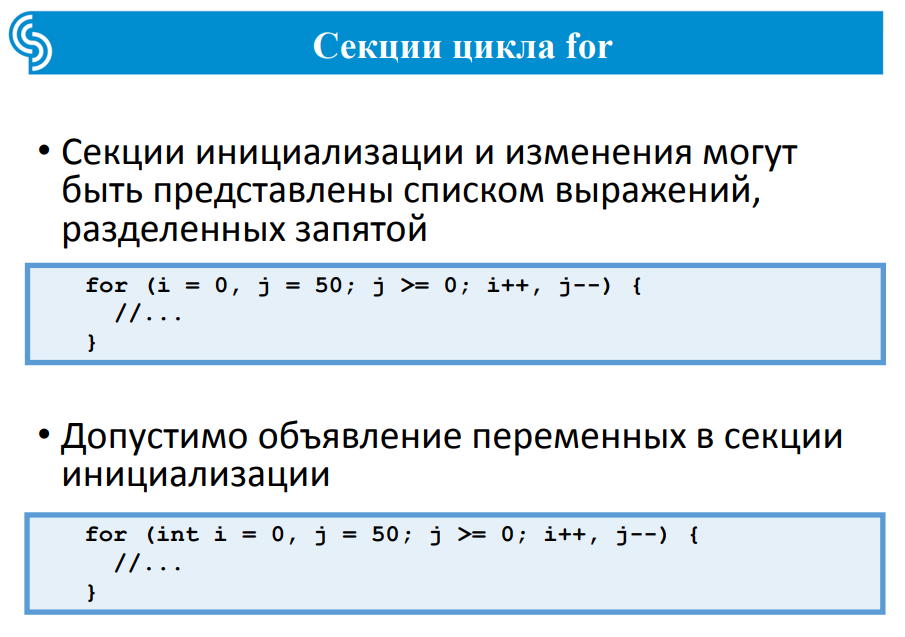
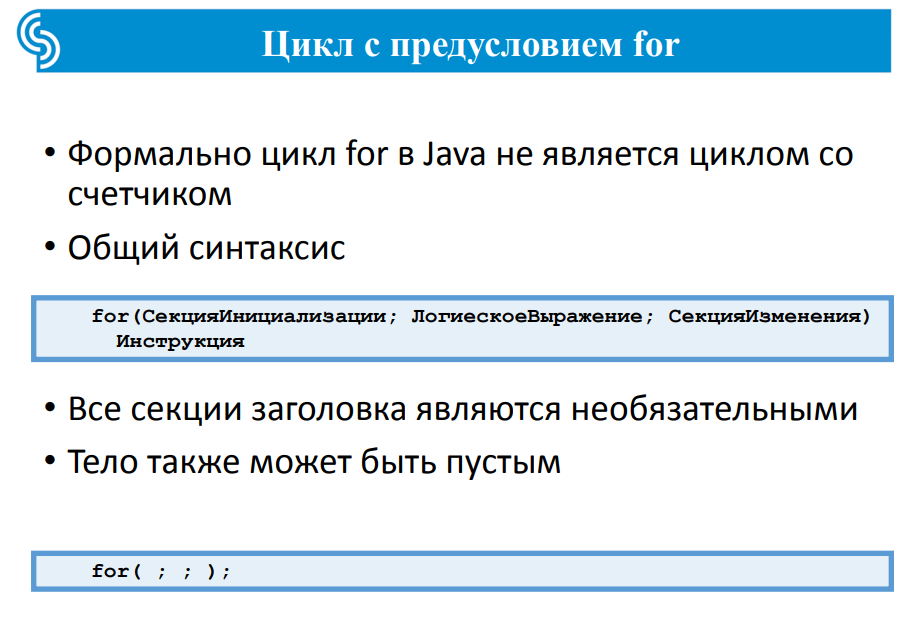
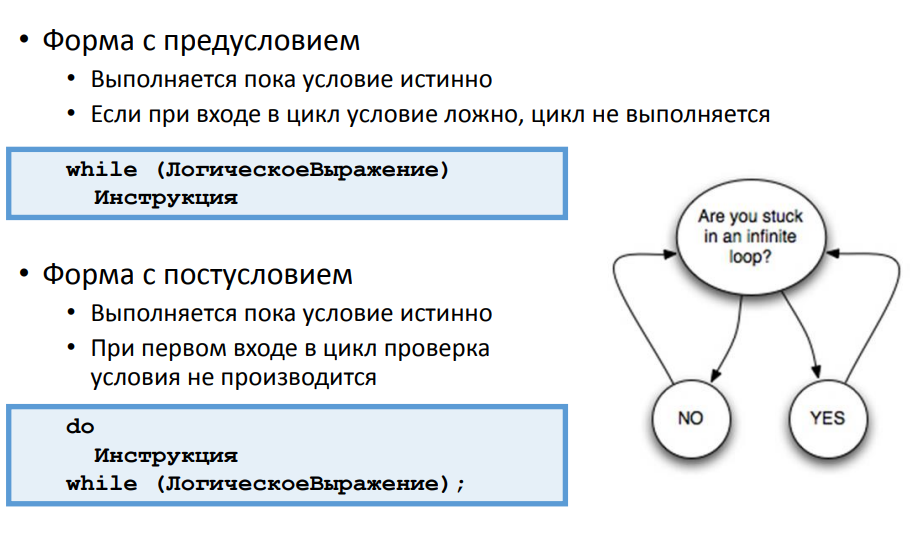
Если ни одно из предложений не подошло, выполняются инструкции, расположенные за меткой default.

Метка default является необязательной.

Метка case или default не служит признаком завершения блока переключателей.

Команда break передает управление первой инструкции, следующей за блоком переключателей.

23. **Циклы.**



24. **Метки.**

Метка

метка: Инструкция

Оператора goto в Java нет!!! Метками можно помечать блоки инструкций и циклы. Обращаться к меткам разрешено только с помощью команд break и continue.

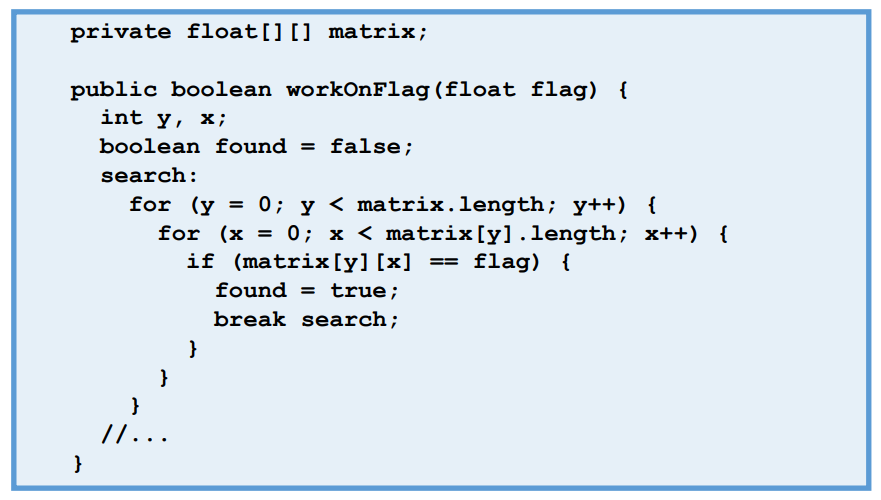
Break:

• Применяется для завершения выполнения кода блока инструкций

• Завершение текущего блока (безымянная форма) - break;

• Завершение указанного блока (именованная форма) - break метка;

• Завершить блок, который сейчас не выполняется, нельзя!



Continue:

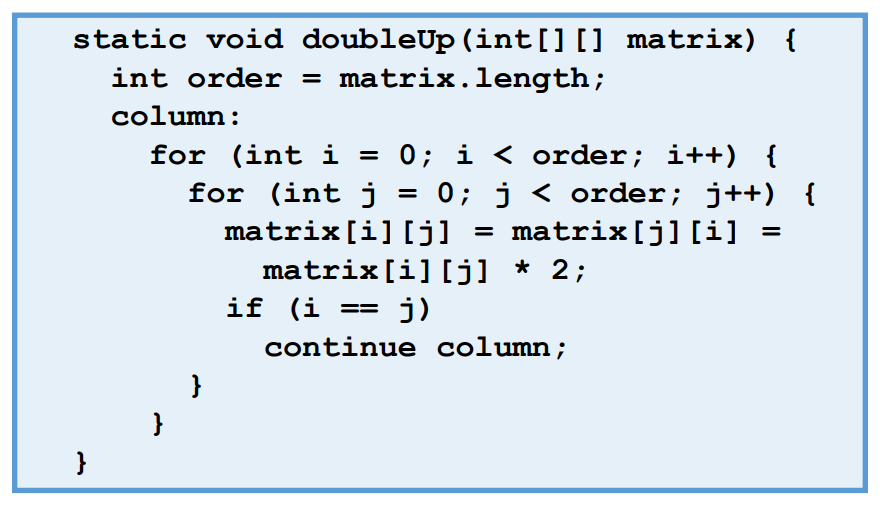
• Применяется только в контексте циклических конструкций

• Производит передачу управления в конец тела цикла

• Завершение витка текущего цикла (безымянная форма) -continue;

• Завершение витка указанного цикла (именованная форма) - continue метка;

• Завершить виток цикла, который сейчас не выполняется, нельзя!



25. **Отношения между классами: наследование, зависимость, метакласс.**

Наследование:

Особенности

* Использование одним классом структуры и/или поведения другого класса
* Дочерний класс специализирует родительский класс
* Образуется отношение «общее-частное» между классами

Реализация на практике

* Часть синтаксиса языка
* Возможен особый синтаксис для множественного наследования
* Особый синтаксис для абстрактных классов
* Возможен особый синтаксис для полностью абстрактных классов

Зависимость:

Особенности

* Изменение в одном классе (независимом) может влиять на другой класс (зависимый)
* Зависимый класс как-то использует независимый
* Отношение направленное по своей природе

Реализация на практике

* У зависимого класса есть операция, сигнатура которой содержит параметр, имеющий тип независимого класса
* В ходе выполнения операции зависимого класса иным способом получается и используется объект независимого класса:

Метакласс:

Особенности

* Метакласс – это класс, объектами которого являются классы
* Корневой метакласс – единственный объект, являющийся своим собственным классом

Реализация на практике

* Очень сильно зависит от языка
* В некоторых языках существует только один метакласс – корневой

26. **Отношения между классами: ассоциация, агрегация и композиция.**

Ассоциация:

Особенности

* Семантическая связь классов
* Направленность
  + Направленная
  + Ненаправленная
* Мощность
  + Один к одному
  + Один ко многим
  + Многие к одному
  + Многие ко многим

Реализация на практике

* У класса есть поле типа другого класса
* Направленность
  + Поле только в одном классе
  + Поля в обоих классах
* Мощность
  + «К одному» – поле имеет просто тип класса
  + «Ко многим» – поле имеет тип массива или коллекции класса

Агрегация:

Особенности

* Можно считать частным случаем ассоциации
* Образуется отношение «целое-часть» между объектами
* Агрегат (контейнер) – большой внешний объект
* Строгая направленность

Реализация на практике

* Всё почти так же, как и в случае ассоциации
* Ссылки между объектами не могут образовывать циклы

Композиция:

Особенности

* Можно считать частным случаем агрегации
* Время жизни объектов частей определяется объектом-контейнером и
* не может превышать время жизни объекта-контейнера
* Объект-часть не существует самостоятельно

Реализация на практике

* Всё почти так же, как в случае агрегации
* Создание и уничтожение объектов-частей происходит только в ходе выполнения операций объекта-контейнера



27. **Дизайн в ООП и его характеристики**:

Дизайн в ООП — это процесс проектирования структуры классов и их взаимодействия для решения задачи. Основные характеристики:

Coupling - (связанность, зависимость)

* Характеристика взаимосвязи модулей
* Степень того, насколько модуль зависит от других модулей
* Мера ресурсов, требующихся при внесении изменений

Cohesion - (связность, сцепленность, сфокусированность, сосредоточенность)

* Степень того, насколько модуль сфокусирован на решение одной задачи
* Степень того, насколько элементы модуля гармонизированы, подходят друг другу.

28. **Связанность в ООП и её виды.**

Связанность содержимого - (content coupling)

* Один модуль изменяет или полагается на внутренние особенности другого модуля (например, используются локальные данные другого модуля)
* Изменение работы второго модуля приведёт к переписыванию первого

Связность через общее - (common coupling)

* Два модуля работают с общими данными (например, глобальной переменной)
* Изменение разделяемого ресурса приведёт к изменению всех работающих с ним модулей

Связанность через внешнее - (external coupling)

* Два модуля используют навязанный извне формат данных, протокол связи и т.д.
* Обычно возникает из-за внешних сущностей (инструментов, устройств и т.д.)

Связанность по управлению - (control coupling)

* Один модуль управляет поведением другого
* Присутствует передача информации о том, что и как делать

Связанность по структурированным данным - (data-structured coupling, stamp coupling)

* Модули используют одну и ту же структуру, но каждый использует только её часть (части могут и не совпадать)
* Изменение структуры может привести к изменению модуля, который изменённую часть даже не использует

Связанность через данные - (data coupling)

* Модули совместно используют данные, например, через параметры
* Элементарные фрагменты маленькие и только они используются модулями совместно

Связанность по сообщениям - (message coupling)

* Модули общаются только через передачу параметров или сообщений
* Состояние децентрализовано

Отсутствие связанности - (no coupling)

* Модули вообще никак не взаимодействуют

29. **Сфокусированность в ООП и её виды.**

Случайная - (coincidental cohesion)

* Части модуля сгруппированы «от фонаря»
* Единственное, что их объединяет – сам модуль

Логическая - (logical cohesion)

* Части модуля логически относятся к одной проблеме
* При этом части могут различаться по своей природе

ВременнАя - (temporal cohesion)

* Части модуля обычно используются в программе в одно время, рядом

Процедурная - (procedural cohesion)

* Части модуля всегда используются в определённом порядке

По взаимодействию - (communication cohesion)

* Части модуля работают над одними и теми же данными

По последовательности действий - (sequential cohesion)

* Результат работы одной части модуля является исходными данными для другой

Функциональная - (functional cohesion)

* Части модуля направлены на решение одной чёткой задачи, за которую отвечает модуль

30. **Принципы SOLID. S, I, D.**

* Single responsibility principle
* Open/closed principle
* Liskov substitution principle
* Interface segregation principle
* Dependency inversion principle

Принцип единственности ответственности

Каждый класс должен иметь единственную ответственность. Эта ответственность должна быть полностью инкапсулирована в этом классе. Все сервисы класса должны быть направлены исключительно на обеспечение его ответственности. Принцип обеспечивает высокую сфокусированность. Класс имеет единственную причину для изменения. При плохой сфокусированности модификация различных ответственностей приводит к комбинаторному взрыву.

Принцип разделения интерфейсов

Клиенты не должны зависеть от методов, которые они не используют. Интерфейсы должны быть сфокусированными. Большие интерфейсы должны разделяться на более мелкие и узкоспециальные. Такие интерфейсы (полностью абстрактные типы) скорее имеют роль тегов, чем организуют свою иерархию наследования.

Принцип инверсии зависимостей

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Высокоуровневые и низкоуровневые компоненты взаимодействуют через абстрактный интерфейс. Интерфейс описывается как часть высокоуровневого компонента. Получается, что низкоуровневый компонент зависит от высокоуровневого, а не наоборот. Это позволяет заменять низкоуровневые компоненты, не изменяя высокоуровневых. Следование принципу снижает связанность. Значительно упрощает разработку сложных систем

31. **Принципы SOLID. O, L.**

* Single responsibility principle
* Open/closed principle
* Liskov substitution principle
* Interface segregation principle
* Dependency inversion principle

Принцип открытости/закрытости

Программные сущности (классы, модули и т.п.) должны быть открыты для расширения, но закрыты для изменения. Сущности могут изменять своё поведение без изменения их исходного кода.

Принцип открытости/закрытости по Мейеру:

Однажды разработанная реализация класса в дальнейшем требует только исправления ошибок. Новые или изменённые функции требуют создания нового класса. Рекомендуется наследование реализации. Реализация используется повторно. Тип не обязан использовать повторно

Полиморфный принцип открытости/закрытости

Используются полностью абстрактные типы. Реализация может быть изменена, или многие реализации могут использоваться полиморфно. Рекомендуется наследование от полностью абстрактных типов. Тип используется повторно и не изменяется. Реализации должны соответствовать типу.

Принцип подстановки Барбары Лисков

* Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) должно быть верным для объектов y типа S, где S является подтипом типа T
* (Р.С. Мартин) Использующие базовый тип функции должны иметь возможность использовать подтипы базового типа не зная об этом
* (Б. Мейер) Дочерний класс не должен нарушать контракт родительского класса

Требования к сигнатурам операций дочерних классов:

* Типы аргументов не должны быть уже
* Типы возвращаемых значений не должны быть шире
* Не должны выбрасываться новые типы исключений, кроме случаев, когда новое исключение является подтипом исключения из родительской сигнатуры
* Область доступа операции не должна сужаться

32. **Закон Деметры, принципы KISS и YAGNI.**

Простое объяснение: Закон Деметры говорит нам о том же, о чем в детстве говорили родители: «Не разговаривай с незнакомцами». А разговаривать можно вот с кем:  
  
— С методами самого объекта.  
— С методами объектов, от которых объект зависит напрямую.  
— С созданными объектами.  
— С объектами, которые приходят в метод в качестве параметра.  
— С глобальными переменными (что лично мне не кажется верным, так как глобальные переменные во многом увеличивают общую сложность).

Закон Деметры (принцип наименьшего знания)

* Каждый модуль должен обладать ограниченным знанием о других модулях: должен знать только о модулях, которые имеют к нему непосредственное отношение
* Каждый модуль должен взаимодействовать только с известными ему модулями и не должен «разговаривать с незнакомыми»
* Каждый модуль должен обращаться только к своим непосредственным друзьям

Объект A может вызвать сервис (метод) объекта B, но не может использовать объект B для получения доступа к объекту C, чтобы использовать его методы

Метод m() объекта O может вызывать только методы следующих объектов

* Сам объект O
* Объекты-параметры метода m()
* Любые объекты, созданные в ходе выполнения m()
* Объектов, непосредственно ассоциированных с O
* Глобальные переменные, доступные O в контексте m()

You ain’t gonna need it

Реализуйте что-то, только если оно вам действительно нужно.

Не реализуйте что-то, использование чего вы только предвидите.

Keep it simple, stupid (Keep it simple and stupid, Keep it short and simple)

Простота должна быть одной из основных целей в ходе разработки

Следует уходить от необоснованных сложностей

33. **Понятие исключения, причины возникновения, механизм обработки. Классификация исключений.**

Исключения в программировании (exceptions) — это механизм, который позволяет программе обрабатывать нетипичную ситуацию и при этом не прекращать работу.

Ошибки возникают в случае:

* некорректного ввода данных
* сбоев оборудования
* нарушения ограничений среды
* выполнения программного кода

Механизм обработки:

• Создается и «выбрасывается» объект исключения, содержащий информацию об ошибке

• Выполнение текущего потока вычислений приостанавливается

• Завершается выполнение блоков и методов в цепочке вызовов, пока не будет найден код, отлавливающий исключение

• Выполнение потока вычислений возобновляется, причем выполняется код обработчика исключения

Классификация исключений:

Объявляемые - Носят предсказуемый характер. Указываются в объявлении метода. Наследуют от класса Exception.

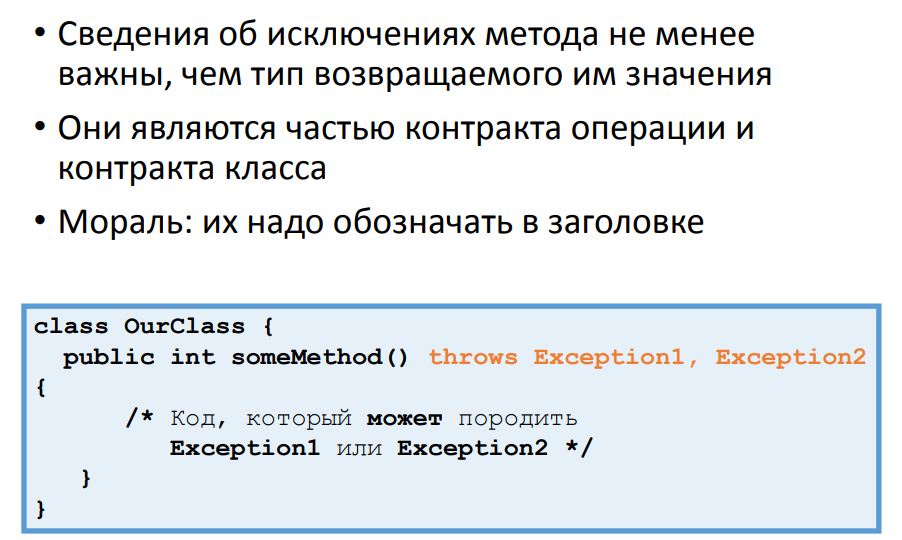
Необъявляемые - Обусловлены логикой кода. Не указываются в объявлении метода. Наследуют от классов RuntimeException, Error.

Синхронные - Непосредственный итог выполнения определенной инструкции

• Могут быть объявляемыми и необъявляемыми

Асинхронные - Не зависят от выполняемой инструкции. Внутренние ошибки JVM. Результат работы deprecated методов.

34. **Объявляемые исключения: синтаксис, особенности, порядок работы.**

****

Особенности:

В списке должны присутствовать те объявляемые исключения, которые не обрабатываются в теле самого метода

Метод вправе выбросить исключение типа, наследного от заявленного в throws

Запрещено генерировать объявляемые исключения типов, не заявленных в throws

Объявляются все объявляемые исключения, необработанные в теле метода

Статические блоки инициализации и инициализирующие выражения не могут выбрасывать исключения

Нестатические блоки инициализации могут генерировать объявляемые исключения, только если их тип указан во всех throws всех конструкторов класса заявленных в throws.

Порядок работы: Исключение выбрасывается в методе. Если исключение не обработано, оно передается в вызывающий метод. Если для исключения не найден подходящий блок catch во всей цепочке вызовов методов:

1. Выполняется блок finally (если есть)

2. JVM останавливает выполнение потока

3. Выводится stack trace — трассировка стека вызовов

4. В случае main потока — программа завершается

35. **Блок try/catch/finally, его предназначение и особенности.**

Блок try:

Заключает в себе блок кода, выполняемый успешно при нормальных обстоятельствах.

Тело выполняется вплоть до:

* Момента возникновения исключительной ситуации
* Благополучного достижения конца блока

Конкретный блок в процессе выполнения может выбросить только одно исключение

Блок catch:

«Внутренний метод» с параметром типа исключения, которое им обрабатывается

Способен:

* Выполнить некоторые восстановительные действия
* Выбросить собственное исключение
* Осуществить необходимые действия и передать управление последующим инструкциям

Количество блоков catch не регламентировано. Предложения catch рассматриваются последовательно до обнаружения среди них того, тип которого допускает присвоение выброшенного исключения. Использовать широкий тип (например, Exception) в качестве отлавливаемого – не лучшая мысль! Список предложений catch просматривается только один раз!

Блок finally:

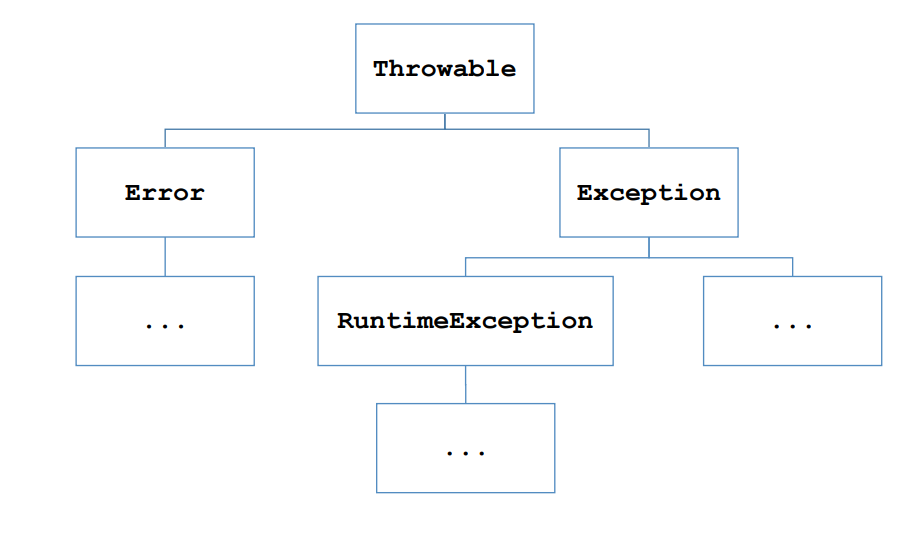
Блок finally необязателен

Если присутствует, то выполняется после завершения работы остальных фрагментов кода try

Блок finally выполняется в любом случае:

* Успешное выполнение
* С выбрасыванием исключения
* С передачей управления по break или return

36. **Иерархия базовых классов исключений. Наследование исключений. Выбрасывание исключений. Отладка приложений.**



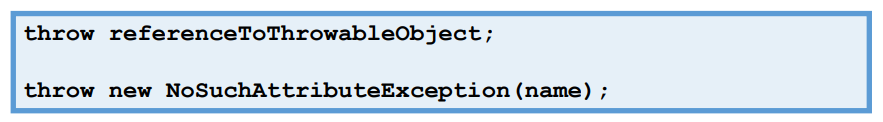
Создается новый тип, наследующий от более широкого типа, подходящего по смыслу (например,

java.lang.IndexOutOfBoundsException). Само то, что выбрасывается исключение более узкого типа,

несет в себе информацию. В состав нового типа могут вводиться новые поля и методы. Чаще всего класс содержит только 2 конструктора (по умолчанию и с параметром-строкой), просто вызывающие конструкторы родительского класса.

Объявляемые и необъявляемые исключения, выбрасываемые вызываемыми методами и операторами.

Явно (принудительно) выбрасываемые исключения



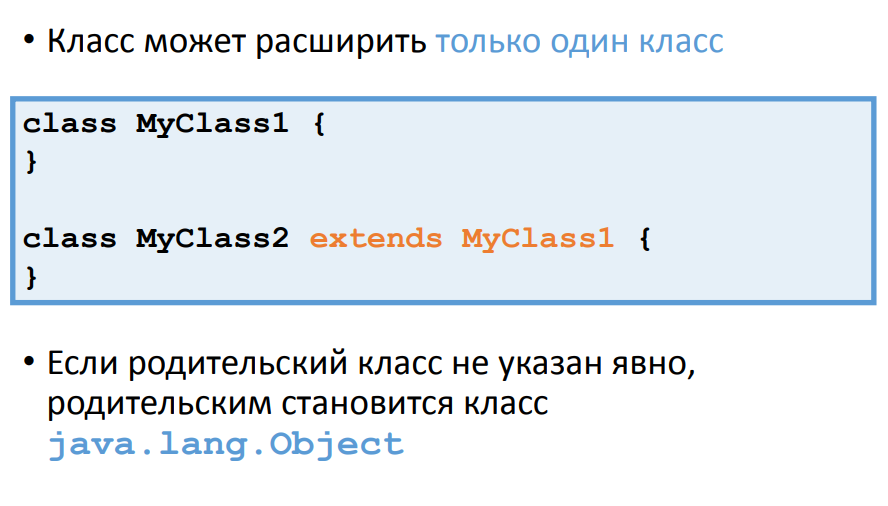
Собственные средства

* Добавление дополнительного кода
* Вывод данных на печать
* Вывод данных в системные журналы (logging)
* Создание дополнительных методов проверки

Отладчики (debuggers)

* В составе JDK, в составе среды разработки (IDE), отладчики сторонних компаний
* Использование точек останова, пошаговых режимов, просмотра состояния объектов

37. **Расширение классов. Порядок создания экземпляра дочернего класса.**



Расширение классов — это создание подкласса (дочернего класса) на основе существующего класса (родительского класса).

Порядок создания экземпляра дочернего класса:

Выполняется статический блок инициализации родительского класса.

Выполняется статический блок инициализации дочернего класса.

Выполняется нестатический блок инициализации родительского класса.

Выполняется конструктор родительского класса.

Выполняется нестатический блок инициализации дочернего класса.

Выполняется конструктор дочернего класса.38. **Расширение классов. Переопределение методов.**

Сигнатуры различны

Перегрузка – добавляется метод с другими параметрами.

Сигнатуры совпадают

Переопределение – замещение версии метода, объявленной в базовом классе, новой, с точно

такой же сигнатурой. При обращении извне объекта производного класса к его методу всегда вызывается новая версия метода. Доступ к методу базового класса изнутри объекта дочернего класса может быть получен с помощью ключевого слова super. Уровень доступа при переопределении не может сужаться. Методы private не переопределяются. В предложении throws дочернего метода не может быть типов исключений, не совместимых с типами в предложении throws родительского метода. Переопределенный метод может быть снабжен модификатором abstract. Признаки synchronized, native и strictfp могут изменяться произвольно.

39. **Расширение классов. Сокрытие полей.**

В предложении throws дочернего метода не может быть типов исключений, не совместимых с

типами в предложении throws родительского метода

Переопределенный метод может быть снабжен модификатором abstract

Признаки synchronized, native и strictfp могут изменяться произвольно

Поля не переопределяются, но скрываются

Поле базового класса при сокрытии продолжает существовать, но недоступно непосредственно по имени

Доступ можно получить с помощью ключевого слова super либо через ссылочную переменную

родительского типа

40. **Ключевое слово super и его использование.**

Действует как ссылка на текущий экземпляр по контракту базового класса

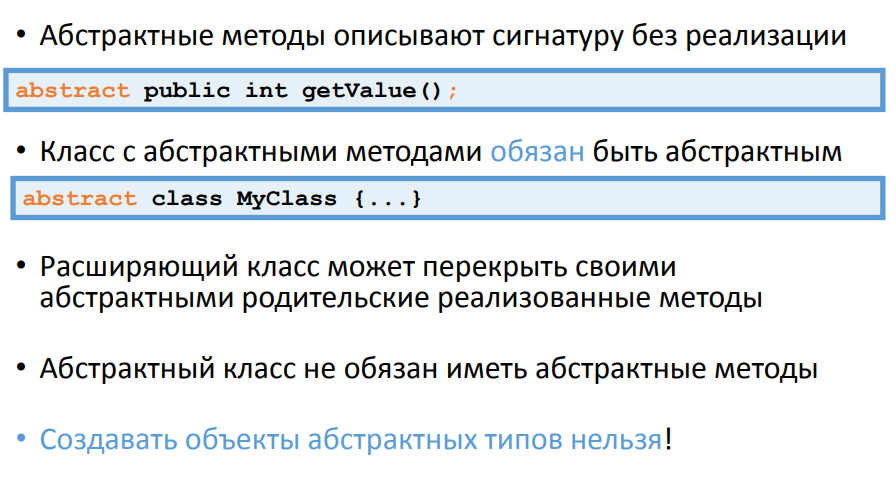
Может быть использовано в теле любого нестатического члена класса

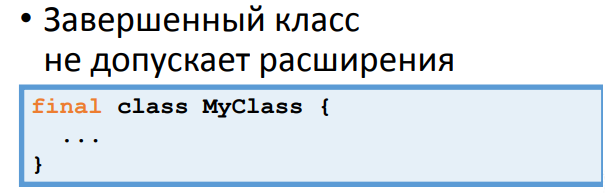
Формы использования

* Вызов конструктора родительского класса - super(...)
* Вызов метода родительского класса - super.method(...)
* Обращение к полю родительского класса - super.field

41. **Абстрактные и завершённые классы в Java**

Абстрактные классы — это классы, которые не могут быть инстанцированы и могут содержать абстрактные методы (без реализации). Завершённые (final) классы — это классы, которые не могут быть унаследованы

****

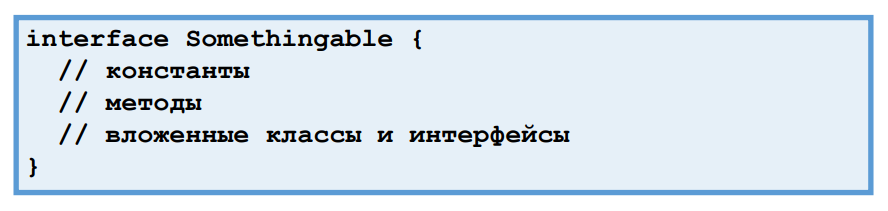


42. **Интерфейсы. Общий синтаксис. Пустые интерфейсы.**

Интерфейс — это контракт, который определяет методы, которые должны быть реализованы классами.

Позволяет описать тип в полностью абстрактной форме. Экземпляры интерфейсов создавать нельзя. Реализация классом интерфейса означает согласие класса на внешний контракт, описываемый реализуемым интерфейсом. Классы способны реализовывать один или несколько интерфейсов.

Все члены интерфейса по умолчанию обладают признаком public. Применение других модификаторов редко имеет смысл. Бывают пустые интерфейсы.



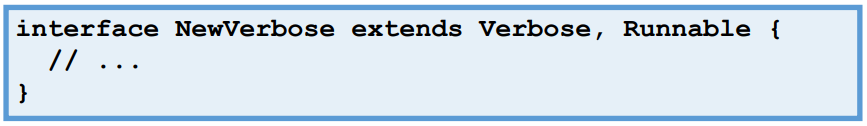
Такие интерфейсы называют маркерными. Они нужны для индикации чего-либо JVM, компилятору или какой-либо библиотеке. Пустые они потому, что позволяют добавить некоторый признак/маркер/индикацию к типам, *не меняя* контракты этих типов.

В Джаве такими интерфейсами являются, например, Serializable и Cloneable. Существование Serializable позволяет сделать проверку "можно ли сериализовать этот объект" более типизированным образом, сведя ее к проверке "имплементирует ли этот объект интерфейс Serializable".

Заменой маркерным интерфейсам также могут служить аннотации (например, @ThreadSafe).

43. **Интерфейсы. Расширение и реализация. Применение. Сравнение с абстрактными классами.**

Расширение:

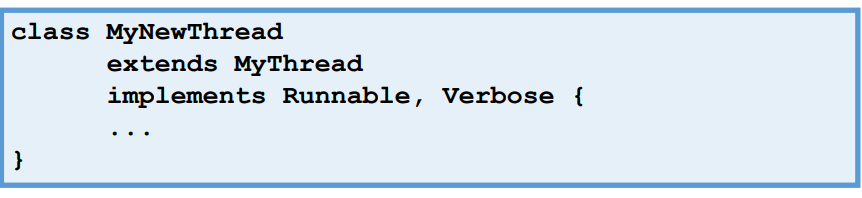
****

Допускается сокрытие констант.

Переопределение метода не несет семантической нагрузки.

Совпадение имен наследуемых методов не несет семантической нагрузки.

Реализация:



Интерфейсы реализуются классами.

Класс может реализовывать несколько интерфейсов.

Если класс не реализует все методы «наследуемых» интерфейсов, он является абстрактным.

Сравнение с абстрактными классами:

Интерфейсы обеспечивают инструментарий безопасного множественного наследования,

производный класс способен наследовать одновременно несколько интерфейсов.

Класс может расширять единственный базовый класс, даже если тот содержит только абстрактные методы.

Абстрактный класс частично может быть реализован, он вправе содержать члены, помеченные как protected и/или static и т.п.

Структура интерфейса ограничена объявлениями public-констант и public-методов без какой

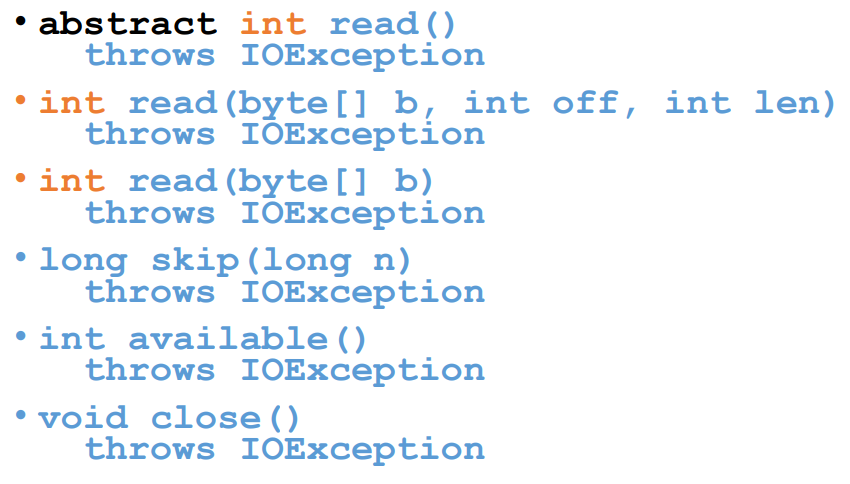
бы то ни было реализации.

44. **Потоки данных и их классификация. Базовые типы и их функциональность.**

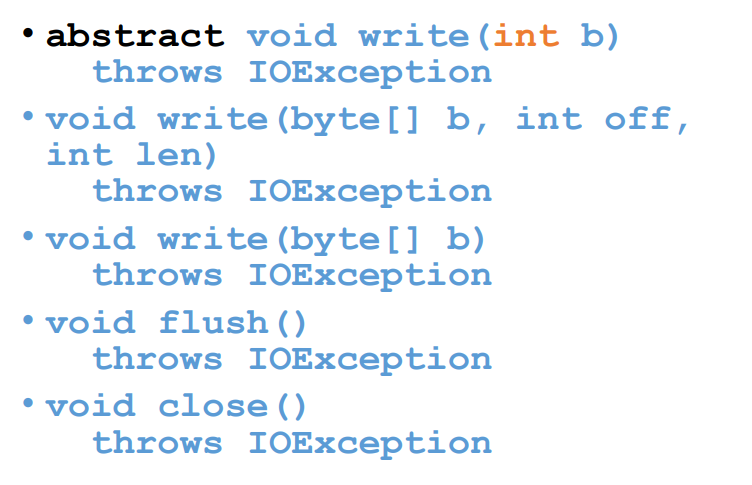
Потоки данных — это последовательности данных, которые могут быть считаны или записаны.

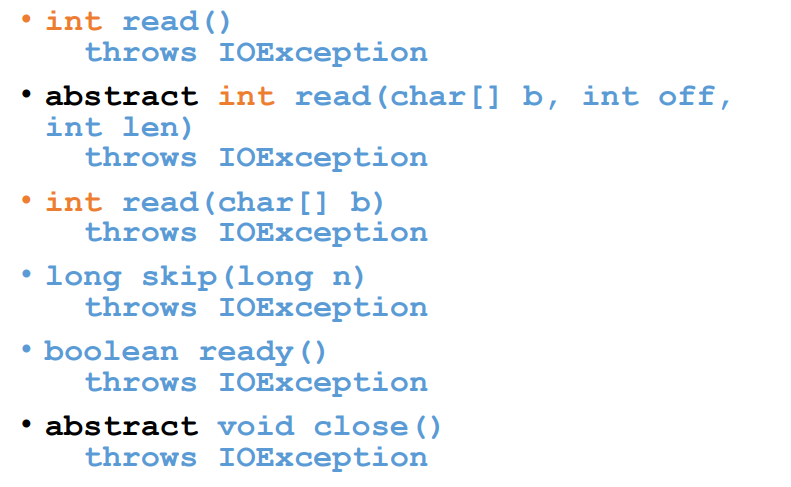
Разновидности потоков:

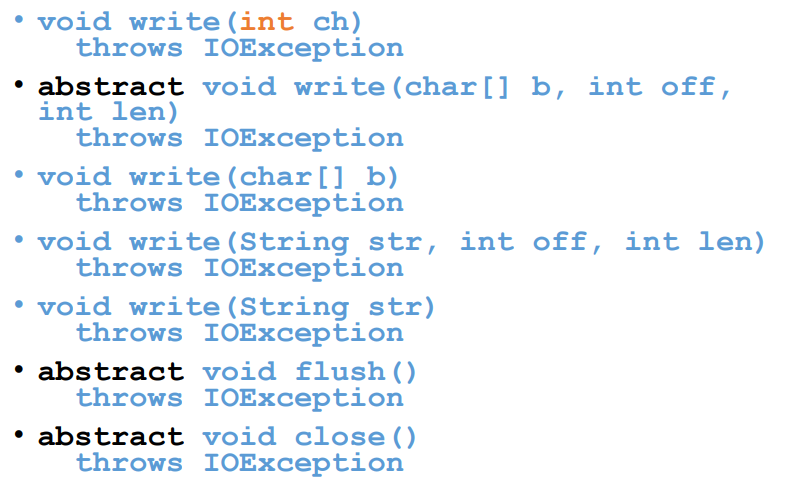


Input stream: 

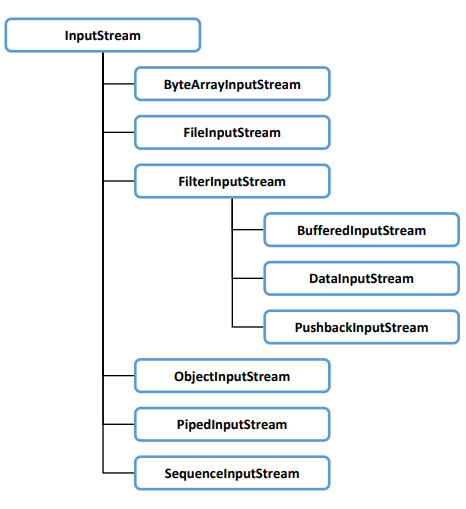
Output stream:



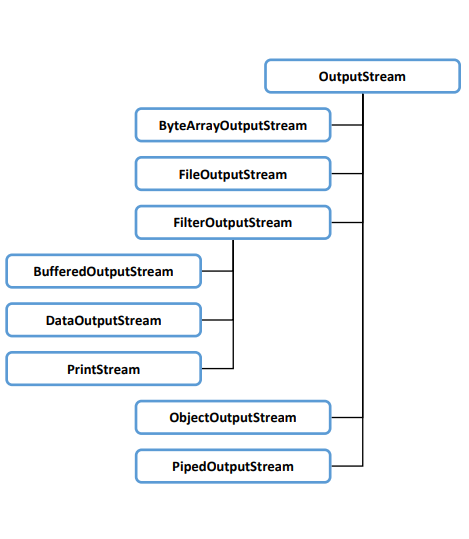
Reader: 

Writer: 

45. **Классы байтовых потоков ввода: иерархия и функциональность.**

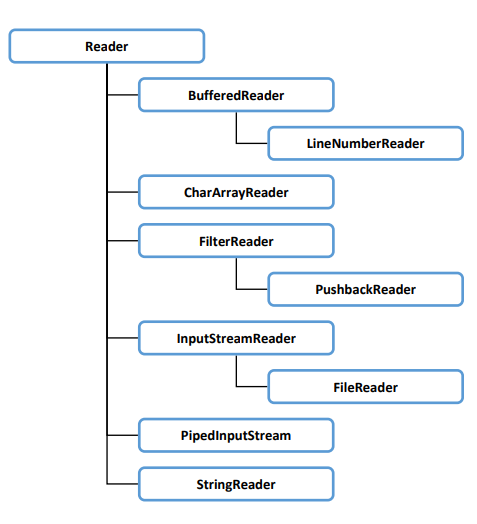


* **InputStream** — базовый абстрактный класс для чтения байтов.
* **ByteArrayInputStream** — читает данные из массива байтов в памяти.
* **FileInputStream** — читает байты из файла.
* **FilterInputStream** — базовый класс для декораторов, добавляющих функциональность.
* **BufferedInputStream** — буферизует ввод для повышения производительности.
* **DataInputStream** — читает примитивные типы данных.
* **PushbackInputStream** — позволяет возвращать байты в поток.
* **ObjectInputStream** — читает сериализованные объекты.
* **PipedInputStream** — используется для обмена данными между потоками.
* **SequenceInputStream** — объединяет несколько входных потоков в один.

46. **Классы байтовых потоков вывода: иерархия и функциональность.** 

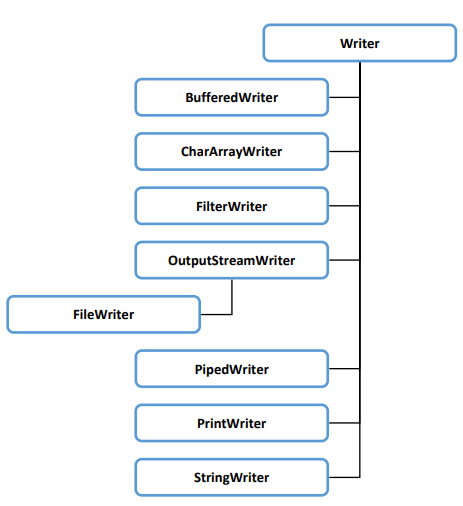
* **OutputStream** — базовый абстрактный класс для записи байтов.
* **ByteArrayOutputStream** — записывает данные в массив байтов в памяти.
* **FileOutputStream** — записывает байты в файл.
* **FilterOutputStream** — базовый класс для декораторов вывода.
* **BufferedOutputStream** — буферизует вывод для повышения производительности.
* **DataOutputStream** — записывает примитивные типы данных.
* **PrintStream** — предоставляет методы для записи форматированных данных.
* **ObjectOutputStream** — записывает сериализованные объекты.
* **PipedOutputStream** — используется для обмена данными между потоками.

47. Классы символьных потоков ввода: иерархия и функциональность.



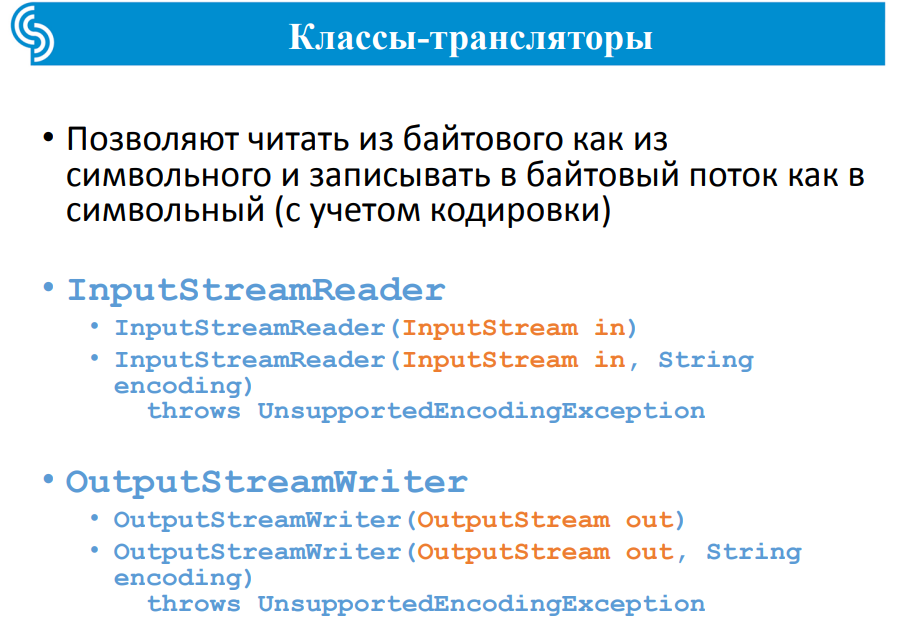
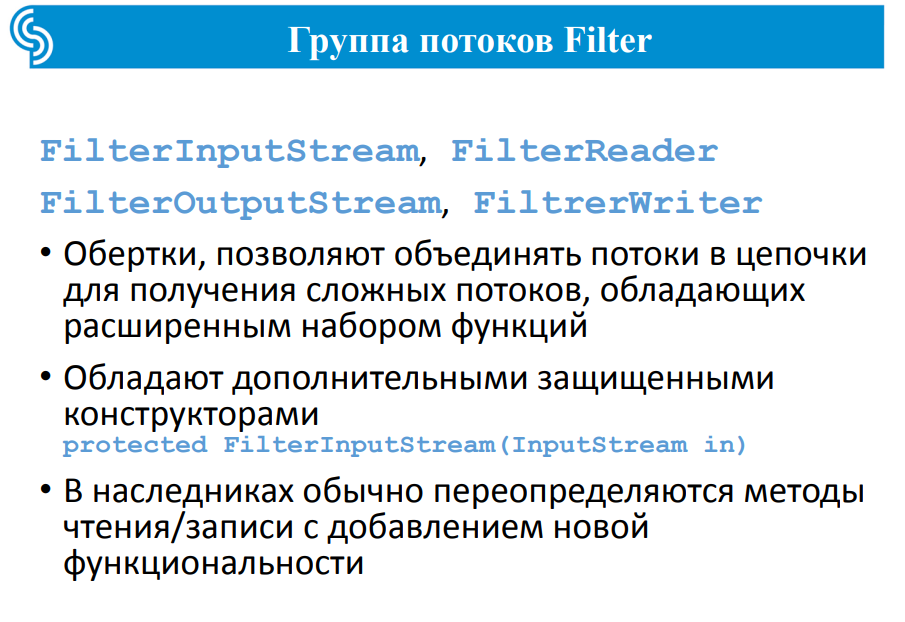
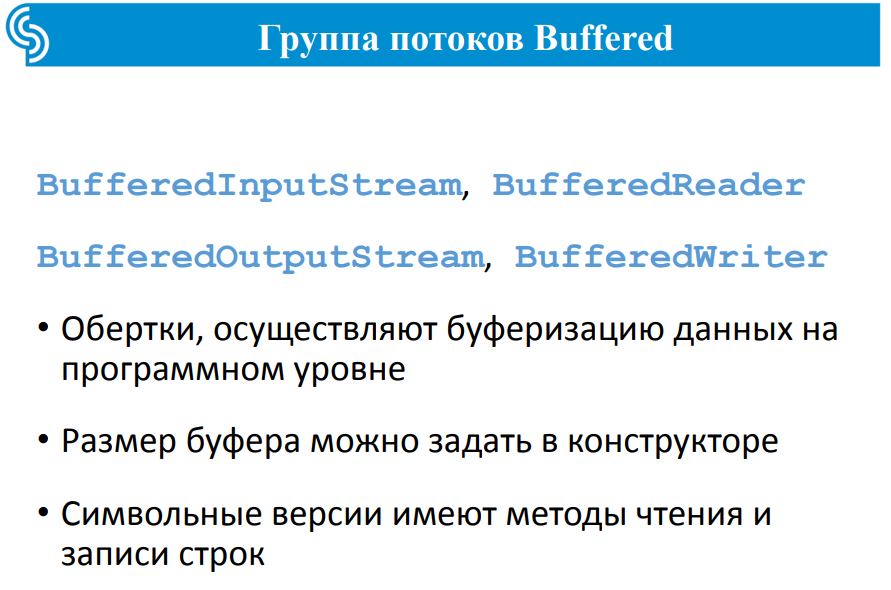
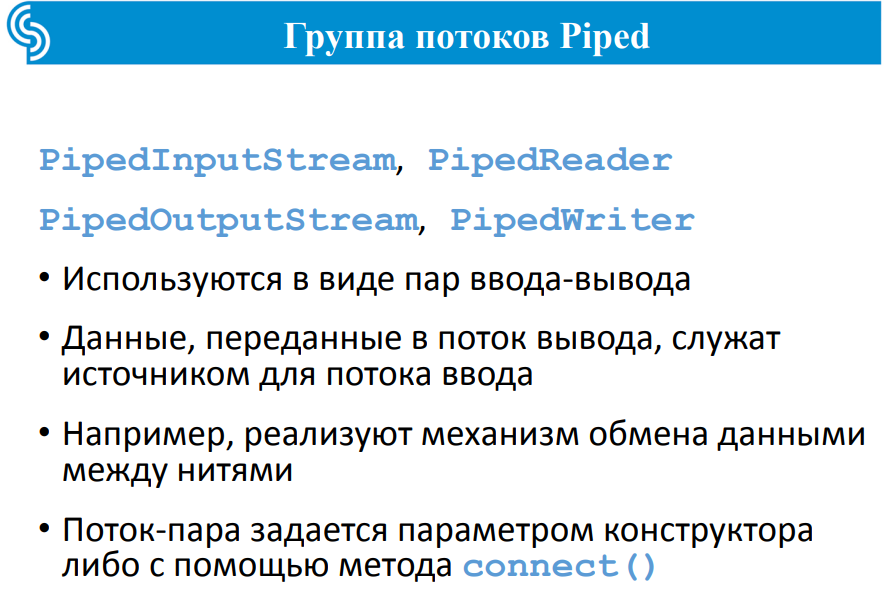
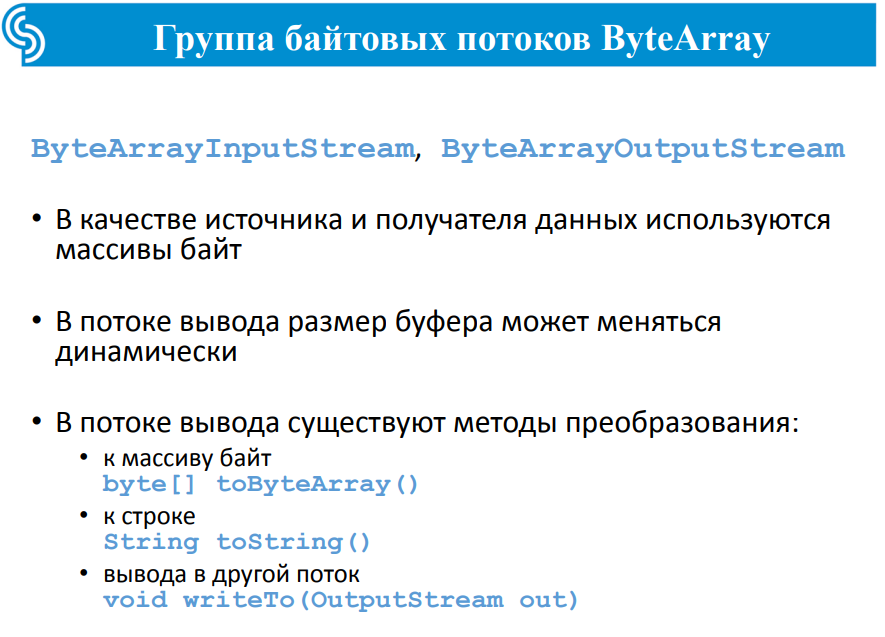
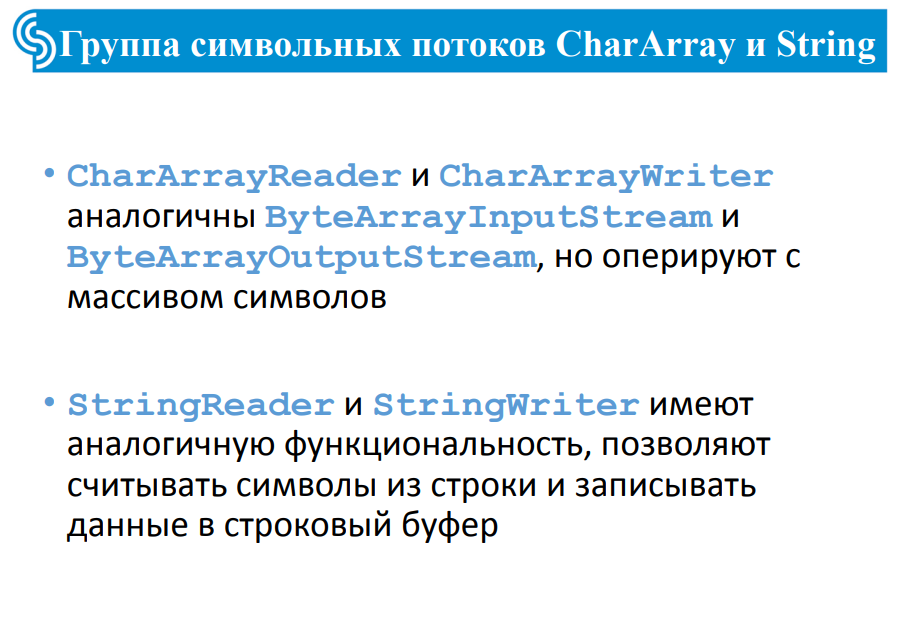
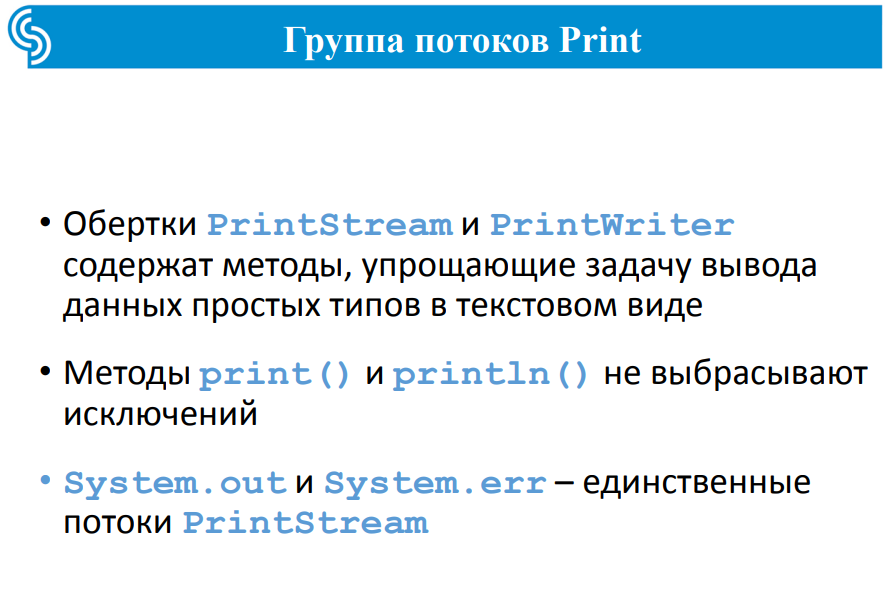
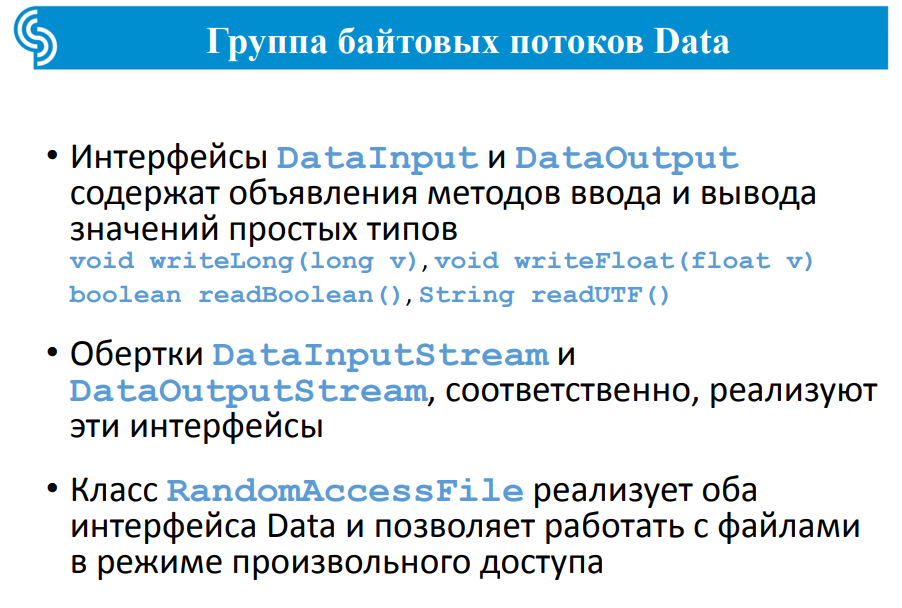
* **Reader** — базовый абстрактный класс для чтения символьных данных.
* **BufferedReader** — буферизует ввод для повышения эффективности чтения.
* **LineNumberReader** — отслеживает номера строк при чтении.
* **CharArrayReader** — читает символы из массива char в памяти.
* **FilterReader** — базовый класс для декораторов символьного ввода.
* **PushbackReader** — позволяет возвращать символы в поток.
* **InputStreamReader** — преобразует байтовый поток в символьный.
* **FileReader** — читает символы из файла (частный случай InputStreamReader).
* **PipedReader** — используется для обмена символьными данными между потоками.
* **StringReader** — читает символы из строки.

48. Классы символьных потоков вывода: иерархия и функциональность.



* **Writer** — базовый абстрактный класс для записи символьных данных.
* **BufferedWriter** — буферизует вывод для повышения эффективности записи.
* **CharArrayWriter** — записывает символы в массив char в памяти.
* **FilterWriter** — базовый класс для декораторов символьного вывода.
* **OutputStreamWriter** — преобразует символьный поток в байтовый.
* **FileWriter** — записывает символы в файл (частный случай OutputStreamWriter).
* **PipedWriter** — используется для обмена символьными данными между потоками.
* **PrintWriter** — предоставляет методы для записи форматированных символьных данных.
* **StringWriter** — записывает символы в строку.

Общая информация по потокам:

49. **Класс File. Работа с файловой системой.**

Инкапсулирует платформенно-независимые методы работы с файлами и директориями:

* создание
* проверка атрибутов
* удаление
* переименование
* Позволяет создавать временные файлы, удаляемые при завершении работы программыFileInputStream, FileReader. FileOutputStream, FileWriter

Позволяют трактовать файл как поток, предназначенный для ввода и вывода данных

Связаны с исключениями FileNotFoundException и SecurityException

Конструкторы могут получать параметры:

* Строку String, задающую имя файла
* Объект класса File
* Объект FileDescriptor (возвращается методом getFD() байтовых потоков)

50. **Форматированный ввод/вывод**

При выводе или отображении данных часто их нужно представить в удобочитаемом виде. Чтобы упростить процесс форматирования строковых данных в Java существует специальный класс Formatter, который располагается в пакете java.util. Основной метод — format() преобразует переданную строку в ее форматированное представление в соответствие с заданным шаблоном.

Класс Scanner является полезным дополнение к рассмотренному нами классу Formatter. И если второй предназначен для форматированного вывода данных, то первый для форматированного ввода. Данный класс имеет набор методов для сканирования текста, получения из него значений примитивных типов и строк, используя регулярные выражения.

51. **Понятие сериализации. Порядок сериализации и десериализации, их особенности.**

Сериализация – преобразование состояния объекта в поток байтов. Десериализация – восстановление состояния объекта из данных потока. Не все объекты могут быть сериализованы. Класс должен быть подготовлен к сериализации.

Класс ObjectOutputStream реализует сериализацию. Класс ObjectInputStream реализует десериализацию. Классы позволяют выводить и вводить графы объектов с сохранением структуры. Результатом десериализации является объект, равнозначный исходному.

Порядок сериализации и десериализации:

• В нисходящем порядке по древовидной иерархии типов: от первого сериализуемого класса до частного типа

• Объекты, на которые ссылаются поля, сериализуются в порядке обнаружения

• Перед десериализацией выполняется загрузка участвующих классов (возможен выброс исключения ClassNotFoundException)

52. **Понятие сериализации. Подготовка классов к сериализации. Принципы настройки сериализации.**

Должен реализовываться интерфейс-маркер java.io.Serializable. Все сериализуемые поля должны иметь сериализуемый тип. Родительский класс должен иметь конструктор по умолчанию (без параметров) или быть подготовленным к сериализации. Сериализуются поля объекта, не обозначенные как transient или static.

Для изменения работы механизма сериализации на уровне вашего класса в самом классе надо описать методы:

* реализация сериализации

private void writeObject(ObjectOutputStream out)

throws IOException

* реализация десериализации

private void readObject(ObjectInputStream in) throws

IOException, ClassNotFoundException

Уровень доступа методов позволяет им независимо существовать в различных классах в иерархии наследования

Можно не переписывать чтение/запись полностью, а лишь изменить порядок записи полей и их формат (см. методы ObjectOutputStream.writeFields() и ObjectInputStream.readFields())

53. **Понятие сериализации. Интерфейс Externalizable**

Интерфейс Externalizable расширяет Serializable и добавляет методы записи и чтения writeExternal и readExternal. Входной и выходной потоки-аргументы в них представлены более абстрактно чем в специальных методах – интерфейсами ObjectInput и ObjectOutput.

Этот интерфейс позволяет реализовать полностью свой механизм сериализации, стандартно запишется только идентификатор класса. Никакой автоматической работы с классом-родителем также не предусмотрено. Методы readObject и writeObject игнорируются. Ключевое слово transient эффекта на Externalizable не имеет.

Externalizable объект в отличие от Serializable десериализуется не в обход конструктора, так что должен иметь конструктор без аргументов.

54. **Структура пакета java.lang. Тип Object, его особенности и методы. Класс Class.**

Базовые классы

* Object
* Class
* Обертки примитивных типов
* Math
* Классы работы со строками
* Классы управления процессами и потоками
* Средства рефлексии
* И т.д.

Базовые интерфейсы

* Cloneable
* Comparable
* Runnable
* И т.д.

Основные исключения

* Exception
* Error
* RuntimeException
* SecutityException
* И т.д.

Этот пакет импортируется по умолчанию

Тип Object - является суперклассом для всех классов (включая массивы). Переменная этого типа может ссылаться на любой объект (но не на переменную примитивного типа). Его методы наследуются всеми классами. Реализует базовые операции с объектами.

Методы Object’а:

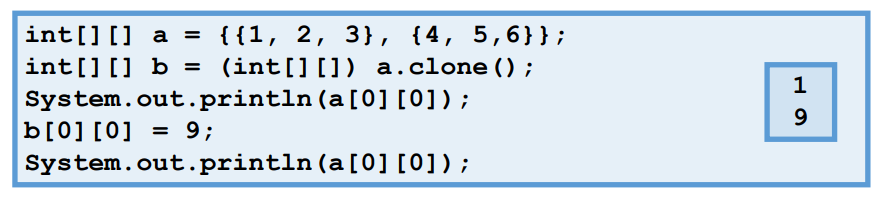
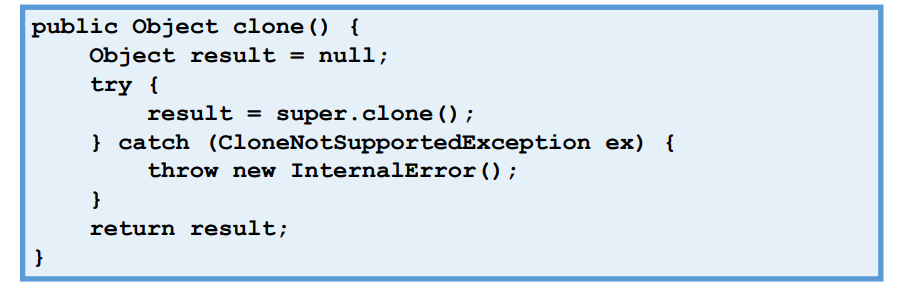
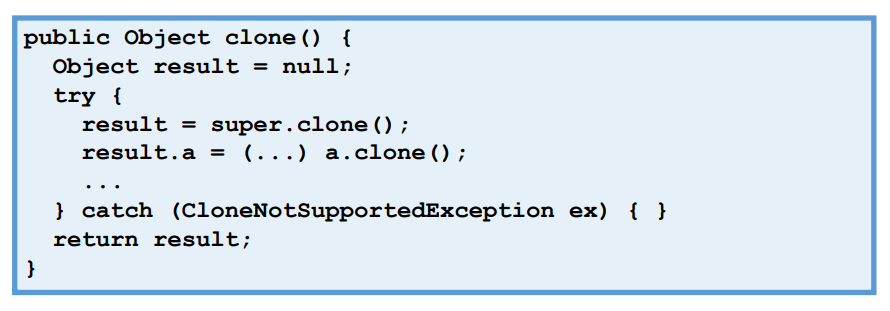
* Получение строкового представления объекта - String toString()
* Получение ссылки на описание класса объекта - final Class getClass()
* Клонирование объекта (получение копии) - protected Object clone()
* Проверка равенства объектов - boolean equals(Object obj)
* Получение хэш-кода объекта - int hashCode()
* Метод завершения работы с объектом - protected void finalize()
* Методы обслуживания блокировок в многопоточных приложениях - void wait(...), void notify(), void notifyAll()

Класс Class:

* Является метаклассом для всех классов Java
* Экземпляры содержат описания классов, загружаемых JVM
* Не имеет доступного конструктора
* Содержит методы для работы с классами и их методами
* Лежит в основе т.н. «рефлексии»

55. **Клонирование объектов.**

Считается, что результатом клонирования является копия объекта. Массивы поддерживают операцию клонирования. В классе Object метод clone() является защищенным. Метод clone() реализуется в конкретном классе. Никто не гарантирует того, что результатом его выполнения будет копия объекта, и даже того, что новый объект будет того же класса. Однако существует ряд соглашений, регламентирующих реализацию метода clone().

Класс должен переопределять метод clone().Класс должен реализовывать интерфейс-маркер Cloneable. Результат клонирования должен быть получен вызовом super.clone(). Результатом работы метода Object.clone() является точная копия объектаВ результате клонирования скопировался только сам объект a, но не объект, на которые он ссылался. При использовании результатов клонирования необходимо явное приведение типа. Начиная с Java5 для массивов можно не выполнять явное приведение типа, но только для массивов. Простого клонирования может быть недостаточно, если объект содержит ссылки на агрегированные объекты. В этом случае после процедуры простого клонирования необходимо создать и их копии тоже.C JavaRush:

Точной копией оригинала является его клон. В Java это означает возможность создавать объект с аналогичной структурой, как и у исходного объекта. Метод clone() обеспечивает эту функциональность. Поверхностное копирование копирует настолько малую часть информации, насколько это возможно. По умолчанию, клонирование в Java является поверхностным, т.е. Object class не знает о структуре класса, которого он копирует. При клонировании, JVM делает такие вещи:

1. Если класс имеет только члены примитивных типов, то будет создана совершенно новая копия объекта и возвращена ссылка на этот объект.
2. Если класс содержит не только члены примитивных типов, а и любого другого типа класса, тогда копируются ссылки на объекты этих классов. Следовательно, оба объекта будут иметь одинаковые ссылки.

Глубокое копирование дублирует все. Глубокое копирование — это две коллекции, в одну из которых дублируются все элементы оригинальной коллекции. Мы хотим сделать копию, при которой внесение изменений в любой элемент копии не затронет оригинальную коллекцию. **Глубокое клонирование требует выполнения следующих правил:**

1. Нет необходимости копировать отдельно примитивные данные;
2. Все классы-члены в оригинальном классе должны поддерживать клонирование. Для каждого члена класса должен вызываться super.clone() при переопределении метода clone();
3. Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

56. **Сравнение объектов. Хэш-функции. Интерфейс Comparable.**

Простого сравнения ссылок недостаточно для сравнения содержимого объектов. Для сравнения объектов по их содержимому применяется метод equals(Object obj). В классе Object метод реализован таким образом, что возвращает true только при сравнении с самим объектом. Конкретный класс должен переопределять метод equals(...). Метод equals(...) должен проверять эквивалентность объектов с точки зрения бизнеслогики. Отношение, задаваемое на множестве объектов этим методом, должно обладать следующими свойствами:

* рефлексивность
* симметричность
* транзитивность
* консистентность
* сравнение с null должно приводить к результату False

Метод int hashCode() предназначен для получения хэш-кода – числа, используемого для быстрого сравнения объектов. Если объект не изменял свое состояние, то значение хэш-кода не должно изменяться. Если два объекта эквивалентны (с точки зрения метода equals()), то хэш-коды объектов должны быть одинаковыми. Если хэш-коды объектов одинаковы, то это еще не значит, что объекты эквивалентны. Изменение реализации в классе метода equals() влечет за собой изменение реализации метода hashCode().

Метод compareTo()

Реализация интерфейса означает введение отношения порядка на множестве объектов класса.

a.compareTo(b) < 0 a < b

a.compareTo(b) > 0 a > b

a.compareTo(b) = 0 a = b

Настоятельно рекомендуется согласовывать работу методов compareTo() и equals().

И Comparable, и Comparator обычно используются в коллекциях Java для реализации функций сортировки. Comparable необходимо реализовать в классе, который требует сортировки, тогда как Comparator может быть реализован в другом классе или также может быть реализован с помощью лямбда-функции.

Если нам нужна сортировка по естественному упорядочению на основе одного атрибута, то вам следует отдать предпочтение Comparable, а если требуется несколько стратегий сравнения, вместо этого рекомендуется использовать интерфейс Comparator.

57. **Классы-обертки примитивных типов.**

Значения примитивных типов не могут быть непосредственно использованы в контексте, где требуется ссылка. Ссылочное представление значений примитивных типов является основной задачей т.н. классов-оберток. Экземпляр такого класса хранит внутри значение примитивного типа и предоставляет доступ к этому значению.

* Boolean
* Byte
* Character
* Double
* Float
* Integer
* Long
* Number
* Short
* Void

Наполнение классов-оберток:

•Константы типов

Integer.MAX\_VALUE, Double.NaN

• Конструкторы: по значению и строке

Float(float value), Float(String s)

• Методы получения значения

Boolean.booleanValue(), Float.floatValue()

• Методы преобразования типов

Integer.parseInt(String s), Float.byteValue()

• Методы проверки состояния и вида значения

compareTo(...), Double.isInfinite()

• Специальные методы, обусловленные спецификой типа

Double.longBitsToDouble(...), Integer.toHexString()

• Ссылочное представление значений примитивных типов

• Хранение вспомогательных функций для работы со значениями примитивных типов

• Представление примитивных типов и их значений в механизмах рефлексии

58. **Работа со строками. Классы String и StringBuffer.**

String:

Значение объекта класса String не может быть изменено без порождения нового объекта. Реализует операции для строки в целом. Экземпляры этого класса можно создавать без ключевого слова new. Каждый строковый литерал порождает экземпляр String. Значение любого типа может быть приведено к строке.

• Строковое представление

valueOf(), copyValueOf()

• Преобразование типов

getBytes(), getChars(), toCharArray(), toString()

• Сравнение

compareTo(), compareToIgnoreCase(), contentEquals(), equals(), equalsIgnoreCase(), intern()

• Выделение элементов

charAt(), substring(), split()

• Операции над всей строкой

concat(), replace(), replaceAll(), replaceFirst(), toLowerCase(), toUpperCase(), trim()

• Проверка содержимого строки

endsWith(), indexOf(), lastIndexOf(), length(), matches(), regionMatches(), startsWith()

StringBuffer:

Реализует методы модификации строки без порождения нового объекта. Реализует операции с элементами строки по отдельности. Используется по умолчанию при конкатенации строк. Для хранения строк использует буфер переменного объема.

• Добавление фрагментов

append(...), insert(...)

• Поиск вхождений

indexOf(), lastIndexOf()

• Извлечение фрагментов

charAt(), getChars(), reverse(), substring()

• Модификация строки

delete(), deleteCharAt(), replace(), setCharAt(), setLength()

• Состояние буфера

length(), capacity(), ensureCapacity(), trimToSize()

59. **Структура пакета java.util. Классы работы со временем, локализацией, Arrays, Random.**

java.util:

• Классы для работы со временем

• Классы для работы с локализацией

• Классы для работы с массивами

• Классы и интерфейсы коллекций

• Классы и интерфейсы для создания многопоточных приложений

• Прочие вспомогательные классы и интерфейсы

java.util.Arrays:

Содержит статические методы для работы с массивами

• Представление массива списком

List asList(Object[] a)

• Поиск элемента в массиве

int binarySearch(…[] a, … key)

• Сравнение массивов по элементам

boolean equals(…[] a1, …[] a2)

• Заполнение массива элементами

fill(…[] a, int from, int to, … val)

• Сортировка массива

sort(…[] a, int from, int to)

Классы работы со временем:

• Date

Отражает дату и время с точностью до миллисекунд. Не рекомендуется к использованию

• Calendar и сопутствующие

Содержит константы и методы для работы с датой и временем с учетом особенностей локализации

• Timer

Позволяет создавать задания для более позднего запуска (с использованием потоков инструкций).

Классы работы с локализацией:

• Locale

Содержит константы и методы для работы с языками и особенностями регионов

• TimeZone

Содержит методы для работы с часовыми поясами

• SimpleTimeZone

Реализует TimeZone для Григорианского календаря

java.util.Random:

• Экземпляр класса является отдельным генератором псевдослучайных чисел (ГПСЧ)

• Различные ГПСЧ позволяют формировать некоррелированные последовательности

• «Основание» имеет размерность 48bit

• Методы получения ПСЧ: nextBoolean(), nextByte(), nextDouble(), nextFloat(), nextInt(), nextLong(), nextGaussian()

• Метод настройки - setSeed(long seed)

60. **Коллекции. Основные типы и их особенности.**

Коллекции (контейнеры) – хранилища, поддерживающие разнообразные способы накопления и упорядочивания объектов с целью обеспечения возможностей эффективного доступа к ним.

В Java коллекции разделены на интерфейсы, абстрагирующие общие принципы работы с коллекциями, и классы, реализующие конкретную функциональность.

Не все методы, заявленные в интерфейсах, должны в действительности реализовываться классами. Часть методов может просто выбрасывать исключение UnsupportedOperationException.

Интерфейс Collection:

* Является образующим для интерфейсов коллекций
* Определяет базовую функциональность любой коллекции
* Подразумевает добавление, удаление, выбор элементов в коллекции
* Допускает дубликаты и пустые элементы

Методы:

• Добавление элементов

boolean add(Object o), boolean addAll(Collection c)

• Исключение элементов

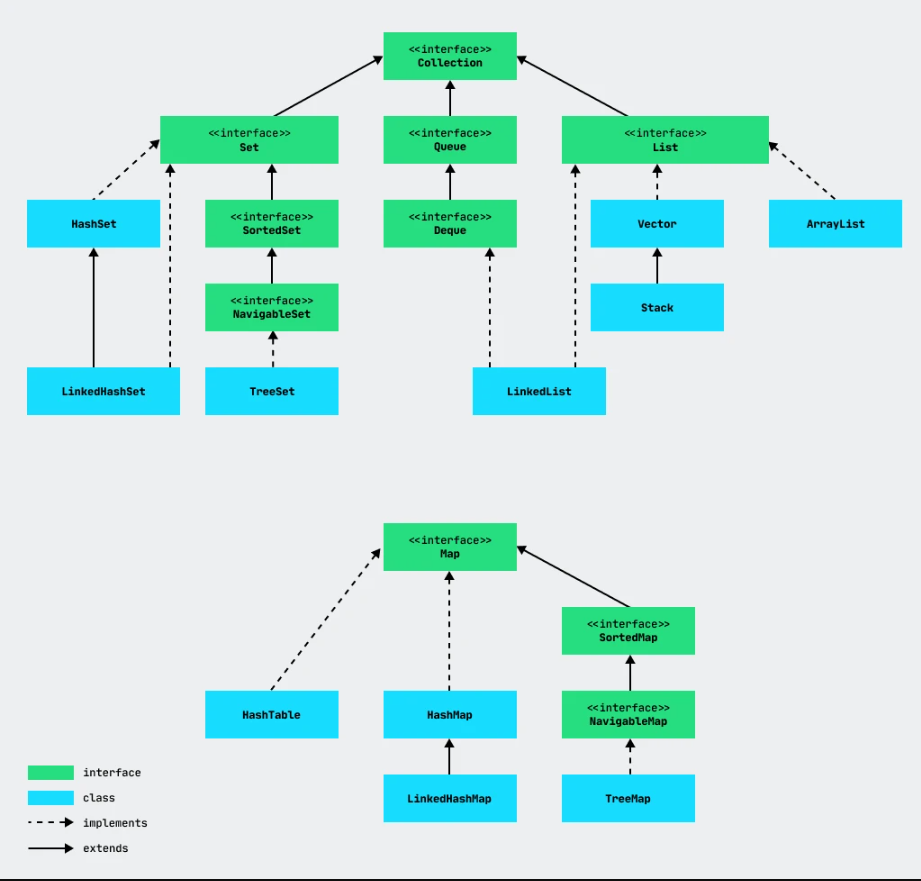
boolean remove(Object o), boolean removeAll(Collection c), boolean retainAll(Collection c), void clear()

• Состояние коллекции

boolean contains(Object o), boolean containsAll(Collection c), boolean isEmpty(), int size()

• Вспомогательные методы

Object[] toArray(), Iterator iterator()



<https://cdn.javarush.com/images/article/7dbef28e-e19b-423a-bc20-8892d85e2917/1024.webp>

Set - Расширяет интерфейс Collection. Не разрешает наличие дубликатов. Допускается наличие только одной ссылки null. Объекты коллекции должны корректно реализовывать метод equals().

List - Расширяет интерфейс Collection. Подразумевает хранение упорядоченной последовательности объектов. Порядок хранения определяется порядком добавления элементов. Позволяет обращаться к элементам по их номеру.

61. **Коллекции. Карты. Классы реализаций и вспомогательные классы.**

Не расширяет интерфейс Collection. Подразумевает хранение набора объектов парами ключ/значение. Ключи должны быть уникальными. Порядок следования пар ключ/значение не определен. Имеет расширение SortedMap, требующее упорядоченности по значениям ключей.

Методы Map:

• Добавление объектов

Object put(Object key, Object value), void putAll(Map t)

• Исключение объектов

Object remove(Object key), void clear()

• Доступ к объекту по ключу

Object get(Object key)

• Состояние

boolean containsValue(Object value), boolean containsKey(Object key), int size(), boolean isEmpty()

• Преобразование типа

Set entrySet(), Set keySet(), Collection values()

Некоторые классы коллекций

• Динамические массивы ArrayList (List), Vector (List)

• Двухсвязный список LinkedList (List)

• Упорядоченные множество и карта TreeSet (Set), TreeMap (StoredMap)

• Ряд других классов HashMap (Map), HashSet (Set)

Collections - Содержит ряд статических методов прикладного назначения, позволяющих оперировать объектами коллекций. Группы методов:

* Создание и поддержка оберток коллекций
* Прочие прикладные методы

Обёртки:

Синхронизированные - Обеспечивают механизмы синхронизации доступа для многопоточных приложений

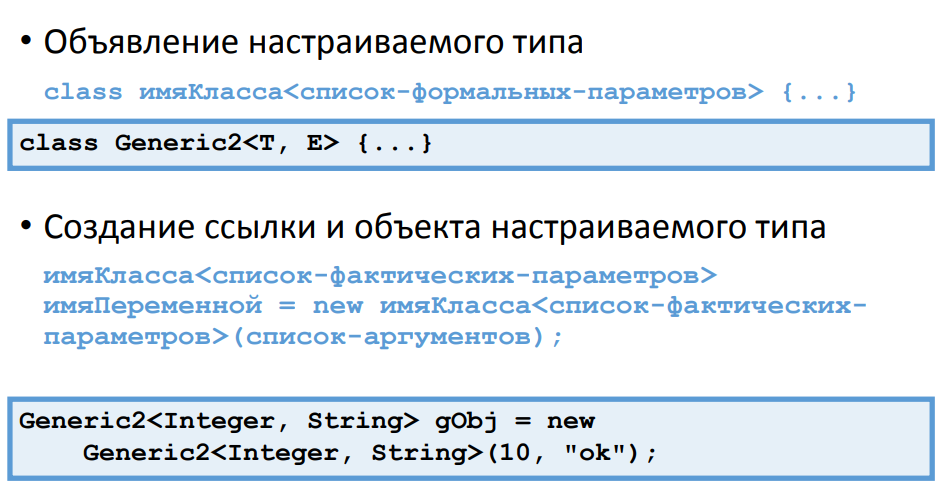
• List synchronizedList(List l), …

Неизменяемые - Запрещают использование методов модификации значений

• Map unmodifiableMap(Map m), …

62. **Настраиваемые типы. Общий синтаксис.**

Generics в Java — это особый способ описание используемых типов, который сможет использовать в своей работе компилятор кода для обеспечения типобезопасности.



Про такое поведение говорят: "Дженерики являются инвариантными". Что такое "инвариант"?

Таким образом, Инвариантность — это отсутствие наследования между производными типами. Если Кошка — это подтип Животные, то Множество<Кошки> не является подтипом Множество<Животные> и Множество<Животные> не является подтипом Множество<Кошки>.

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

**List**<**String**> lines = **new** **ArrayList**<>();

lines.add("Hello world!");

**System**.out.println(lines);

}

Компилятор по данному коду понимает, что если мы указали в левой части, что List будет содержать объекты типа String, то в правой части мы подразумеваем, что хотим в переменную lines сохранить новый ArrayList, который будет хранить тоже объект того типа, который указан в левой части. Таким образом, компилятор из левой части понимает или выводит тип для правой части. Именно поэтому такое поведение называется выведением типа или "Type Inference" на английском.

class MyClass<T> {  
 private T value;  
 MyClass(T value) {  
 this.value = value;  
 }  
 T getValue() {  
 return value;  
 }  
}

63. **Настраиваемые типы с ограничениями. Метасимвольный аргумент. Настраиваемые методы.**

Настраиваемые типы с ограничениями:

Ограничение типа позволяет использовать у ссылок методы и поля, доступные в типе-ограничителе. Типы, не наследующие от указанного, не могут быть использованы при создании объектов. Как имя типа может быть указан интерфейс!!! Как имя типа может быть указан ранее введенный параметр!!!

Например, наш класс Box мог бы содержать всех животных, тогда бы мы объявили дженерик следующим образом:

**public** **static** **class** **Box**<**T** **extends** **Animal**> {

То есть мы задали ограничили верхнюю границу классом Animal. Так же мы можем указать несколько типов после ключевого слова extends. Это будет означать, что тип, с которым мы будем работать, должен быть наследником некоторого класса и при этом реализовывать некоторый интерфейс. Например:

**public** **static** **class** **Box**<**T** **extends** **Animal** & **Comparable**> {

В этом случае, если мы попробуем положить в такой Box что-то, что не является наследником Animal и не реализует Comparable, то во время компиляции мы получим ошибку:

error: type argument **Cat** is not within bounds of type-variable **T**

Методы:

public class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

public void setKey(K key) { this.key = key; }

public void setValue(V value) { this.value = value; }

public K getKey() { return key; }

public V getValue() { return value; }

}



public static <T> T createObject(Class<T> object){  
 try{  
 return object.getDeclaredConstructor().newInstance();  
 } catch (Exception e){  
 throw new IllegalStateException(e);  
 }  
}

Метасимвольный аргумент:

Wildcard — удобный инструмент, чтобы смягчить некоторый ограничения дженериков. Например, как мы ранее разбирали, дженерики инвариантны. Это значит что хотя все классы являются наследниками (подтипами, subtypes) типа Object, но List<любой тип> не является подтипом List<Object>. НО, List<любой тип> является подтипом List<?>. Таким образом, мы можем написать следующий код:

**public** **static** **void** printList(**List**<?> list) {

**for** (**Object** elem: list) {

**System**.out.print(elem + " ");

}

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

}

Как и обычные дженерики (т.е. без использования wildcard'ов), дженерики с wildcard могут быть ограничены. Ограничение по верхней границе (Upper bounded wildcard) выглядит привычно:

**public** **static** **void** printCatList(**List**<? **extends** **Cat**> list) {

**for** (**Cat** cat: list) {

**System**.out.print(cat + " ");

}

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

}

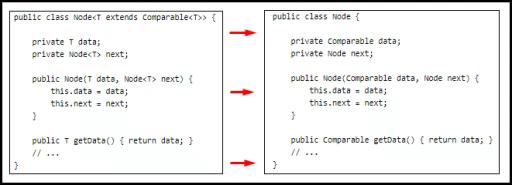
Но можно ограничить и по нижней границе (Lower bound wildcard):

**public** **static** **void** printCatList(**List**<? **super** **Cat**> list) {‘

Таким образом метод начнёт принимать всех котов, а так же по иерархии выше (вплоть до Object).

64. **Настраиваемые типы. Особенности и внутренняя реализация.**

Говоря про дженерики стоит знать про "Стирание типов". На самом деле стирание типов это про то, что дженерики — это информация для компилятора. Во время выполнения программы информации о дженериках больше нет, это и называется "стирание". Это стирание имеет тот эффект, что тип дженерика заменяется конкретным типом. Если дженерик не имел границы, то будет подставлен тип Object. Если же граница была указана (например <T extends Comparable>), то она и будет подставлена. Вот пример из Tutorial от Oracle: "[Erasure of Generic Types](https://docs.oracle.com/javase/tutorial/java/generics/genTypes.html)":

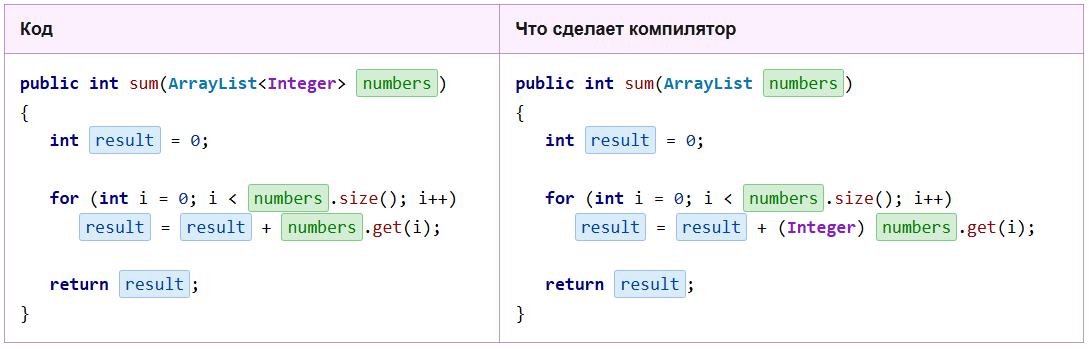


На самом деле Generics работают до ужаса примитивно.

Компилятор просто заменяет тип с параметром на него же, только без параметра. А при взаимодействии с его методами добавляет операцию приведения типа к типу-параметру:



Допустим, у нас был код метода, который суммирует числа в коллекции целых чисел:



Т.е. по сути дженерики — это такая разновидность синтаксического сахара, как и autoboxing, только побольше. При autoboxing компилятор за нас добавляет методы для преобразования типа int к Integer и обратно, а для generics добавляет операторы приведения типа.

После того, как компилятор скомпилировал ваш код с дженериками, в нем все классы с параметрами были преобразованы просто в классы и операторы приведения типа. Информация о том, какие изначально были типы-параметры у переменных сложных типов, потерялась. Этот эффект еще называют *стиранием типов*.

Иногда программистам, которые пишут свои классы с типами-параметрами, очень не хватает информации о типах, которые туда передаются в качестве параметров. Как с этим борются и что из этого выходит, вы узнаете в квесте Java Collections.

65. **Автоупаковка и автораспакова. Особенности автоупаковки/автораспаковки**

• Автоупаковка – процесс автоматической инкапсуляции данных простого типа в экземпляр соответствующего ему класса-обертки в случаях, когда требуется значение ссылочного типа

• Автораспаковка – процесс автоматического извлечения примитивного значения из объектаупаковки в случаях, когда требуется значение примитивного типа

Автоупаковка применяется компилятором Java в следующих условиях:

* Когда значение примитивного типа передается в метод в качестве параметра метода, который ожидает объект соответствующего класса-оболочки.
* Когда значение примитивного типа присваивается переменной, соответствующего класса оболочки.
* Когда объект передается в качестве параметра методу, который ожидает соответствующий примитивный тип.
* Когда объект присваивается переменной соответствующего примитивного типа.

А внутри магия выглядит вот так:

То есть по сути будет взят новый Integer или будет получен Integer из кэша (кэш ничто иное как просто массив из Integer) в зависимости от величины значения числа.

В первом случае, значение берётся из кэша Integer значений (см. объяснение Boxing выше), а во втором случае будет создаваться каждый раз новый объект. Но тут стоит оговориться. Это поведение зависит от верхней границы кэша (**java.lang.Integer.IntegerCache.high**). Кроме того, эта граница может измениться и из-за других настроек. На эту тему можно ознакомиться с обсуждением на stackoverflow: [How large is the Integer cache?](https://stackoverflow.com/questions/15052216/how-large-is-the-integer-cache) Естественно, объекты нужно сравнивать через equals:

Вторая проблема, связанная с этим же механизмом – производительность. Любая упаковка в Java равносильна созданию нового объекта. Если число не входит в значения из кэша (т.е. в -128 до 127), то будет каждый раз создаваться новый объект. Если вдруг упаковка (т.е. boxing) будет производиться в цикле, это вызовет огромный прирост ненужных объектов и потребление ресурсов на работу сборщика мусора. Поэтому, не стоит слишком безрассудно относится к этому.

66. **Компонентный подход в программировании.**

Объектный подход - Программа пишется как объектная модель реальной или воображаемой системы, объекты и их отношения легко понятны

Компонентный подход - Программа собирается из заранее созданных компонентов, которые могут быть достаточно сложны для понимания

Компонент – это элемент системы, представляющий определенную службу или сущность, допускающий взаимодействие с другими компонентами

Особенности компонентов:

* Многоразовое использование
* Независимость от контекста
* Кооперация с другими компонентами
* Инкапсуляция
* Самостоятельность как единицы программы
* Компонент – объект, написанный в соответствии со спецификацией

Требования к разработке компонентов:

* Полная документация
* Более строгое тестирование
* Надежная проверка достоверности входных данных
* Возврат полезных сообщений об ошибках
* Разработка с учетом непредусмотренного использования

67. **Компоненты JavaBeans. Требования к компоненту.**

Компонентная модель для языка Java. Компоненты называются Beans (бины). Компонент описывается как класс, удовлетворяющий определенным правилам. Следует различать компонент как класс компонента и компоненты как его экземпляры, использующиеся в программе. Для работы с компонентами используются специальные программные средства, позволяющие «визуально» настраивать компонент и его взаимодействие с другими компонентами (RAD-tools).

Интроспекция:

Средства работы с бинами подвергают бины интроспекции. Интроспекция – механизм, основанный на рефлексии, позволяющий определить характеристики компонента, основываясь на информации, заключенной в именах элементов класса компонента и вспомогательных классах.

Компоненты JavaBeans:

Компонент имеет свойства, которые представляют собой характеристики компонента и могут быть изменены в процессе сборки. Компонент использует события для взаимодействия с другими компонентами. Компоненты обладают свойством персистентности: могут сохранять свое состояние в долговременное хранилище и затем восстанавливать его. Методы компонентов являются обычными методами Java и могут использоваться другими компонентами

Требования к компоненту JavaBeans:

* Способность к инстанцированию нового экземпляра (бин – не интерфейс, не абстрактный класс)
* Наличие конструктора по умолчанию (конструктора без параметров)
* Возможность сериализации
* Следование соглашениям об именах и способах проектирования
* Использование мрефодели делегирования обработки событий

68. **Свойства компонентов JavaBeans. Классификация и особенности (без примеров реализации).**

Виды свойств по способу доступа

* Допускающие запись
* Допускающие чтение

Виды свойств по сложности

* Простые
* Индексированные
* Связанные
* Контролируемые

Простые свойства:

Доступны пользователю как пара методов доступа (setter/getter).

Любой из этих методов может отсутствовать, определяя таким образом доступность свойства для чтения/записи (read-only/write-only).

Соглашения именования:

Общий вид:

* public ТипСвойства getИмяСвойства()
* public void setИмяСвойства(ТипСвойства значение)

Логические свойства:

* public boolean isИмяСвойства()
* public void setИмяСвойства(boolean значение)

Индексированные свойства:

Определяют доступ к массиву свойств с возможностью доступа по номеру.

Соглашения именования:

* public ТипСвойства[] getИмяСвойства()
* public void setИмяСвойства (ТипСвойства[] значения)
* public ТипСвойства getИмяСвойства (int индекс)
* public void setИмяСвойства (int индекс, ТипСвойства значение)

Связанные свойства:

Поддерживают механизм оповещения об изменениях значений свойств.

Обработчики реализуют интерфейс PropertyChangeListener и принимают объекты PropertyChangeEvent, содержащие имя свойства и его старое и новое значения.

Используется вспомогательный класс PropertyChangeSupport.

Контролируемые свойства:

Похожи на связанные свойства, но перед присвоением нового значения оно должно быть проверено обработчиками.

Любой обработчик в праве запретить изменение, в этом случае значение свойства останется прежним.

Обработчики реализуют интерфейс VetoableChangeListener и принимают объекты PropertyChangeEvent, содержащие имя свойства и его старое и новое значения.

Используется вспомогательный класс VetoableChangeSupport.

69. **Графические компоненты (без примеров реализации). Упаковка в Jar. Персистентность.**

Если компоненты содержит средства визуализации, то его суперкласом должен быть:

• java.awt.Component - если создается компонент на базе AWT

• javax.swing.JComponent - если создается компонент на базе Swing

Хотя и компоненты AWT, и компоненты Swing являются компонентами JavaBeans, не рекомендуется смешивать их в одном контейнере.

Графические компоненты — это элементы интерфейса, такие как кнопки, текстовые поля, панели и т.д. Они используются в библиотеках AWT, Swing и JavaFX.

Класс компонента, вспомогательные классы, файлы помещаются в jarархив. В файле манифеста при этом указывается дополнительная информация. Упаковка в Jar: Jar-файл — это архив, содержащий классы, ресурсы и метаданные. Создается с помощью команды jar или через IDE.

Персистентность - Способность объекта существовать во времени, переживая породивший его процесс. Для JavaBeans реализуется за счет сериализации. Все классы компонентов должны быть подготовлены к сериализации. Существуют механизмы сохранения состояния бинов в xml-формат.

Персистентность в Java — это способность объектов сохранять свое состояние за пределами времени жизни программы, чаще всего путем хранения в базе данных, что обеспечивается спецификацией Java Persistence API (JPA) и ее реализациями (например, Hibernate)

70. **Принципы обработки событий компонентов.**

Обработка событий — это механизм, который позволяет компонентам реагировать на действия пользователя (например, клик мыши).

• Событие – объект, описывающий изменение состояния источника

• Источник – объект, генерирующий события

• Слушатель – объект, получающий уведомление о событии

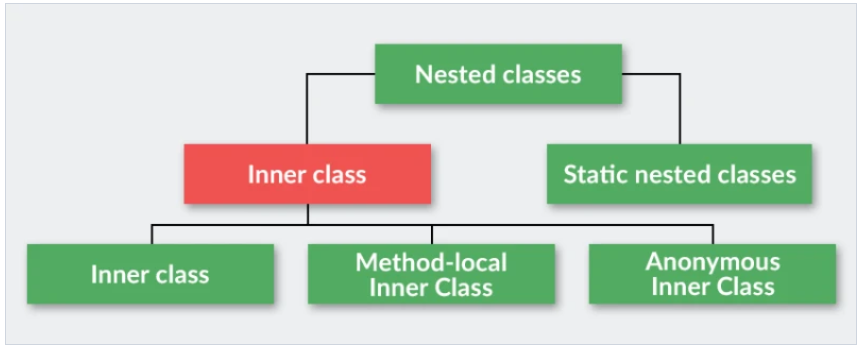
Источник генерирует событие и посылает его одному или нескольким слушателям. Слушатель просто ждет поступления события. Получив событие, слушатель обрабатывает его и затем возвращает управление.

Слушатели должны зарегистрироваться у источника – события посылаются только зарегистрировавшимся слушателям. Логика кода, обрабатывающего события, отделена от логики интерфейса, генерирующего события. Это реализация паттерна проектирования Observer.

Один источник может генерировать несколько типов событий. Один слушатель может быть слушателем нескольких видов событий. Один слушатель может получать события одного вида от нескольких источников. Слушатель может быть источником для других слушателей и для самого себя. Источник может быть слушателем других источников и самого себя.

71. **Вложенные типы, классификация и особенности. Вложенные классы и интерфейсы.**

Вложенные типы - считаются частью внешнего типа, со всеми вытекающими правами. Позволяют представить тип в виде логически связанных структурных групп и контекстов. Простой и эффективный инструмент объединения семантики соотносимых объектов.

****

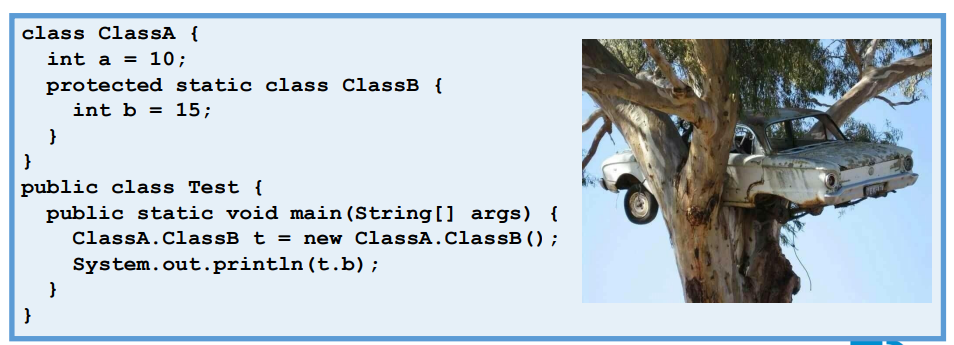
Особенности: Вложенные классы имеют доступ к членам внешнего класса. Статические вложенные классы не имеют доступа к нестатическим членам внешнего класса.

Статические вложенные классы:

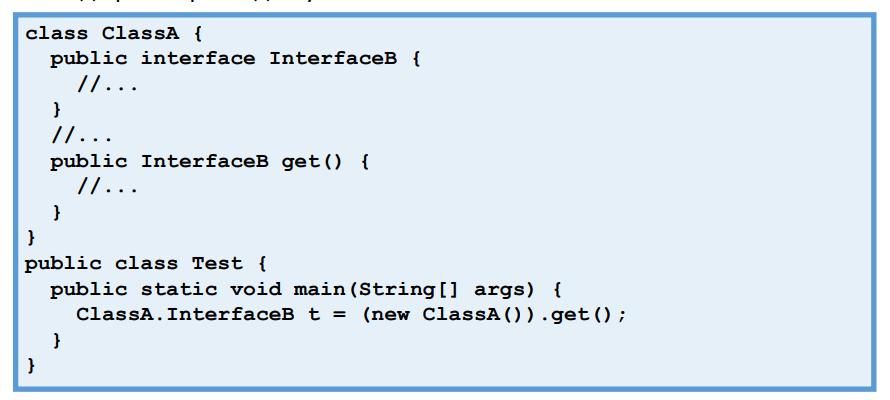
• Nested class

• Простейшая форма вложенного класса

• Может использоваться любыми классами, обладающими соответствующими правами доступа



Вложенные интерфейсы:

Всегда статичны (модификатор static по умолчанию). Доступ определяется доступом внешнего класса или интерфейса и модификаторами доступа.

72. **Внутренние, локальные и анонимные классы.**

Нестатические вложенные классы:

Внутренние классы — это классы для выделения в программе некой сущности, которая неразрывно связана с другой сущностью.

Особенности внутреннего класса:

* Объект внутреннего класса не может существовать без объекта «внешнего» класса.
* У объекта внутреннего класса есть доступ к переменным «внешнего» класса.
* Объект внутреннего класса нельзя создать в статическом методе «внешнего» класса.
* Внутренний класс не может содержать статические переменные и методы.
* При создании объекта внутреннего класса важную роль играет его модификатор доступа.
* Модификаторы доступа для внутренних классов работают так же, как и для обычных переменных.

Называются внутренними (inner) классами. Тип находится в контексте объекта. Объект внутреннего класса всегда ассоциируется с т.н. внешним объектом. Элементы внутреннего класса имеют доступ к полям и методам внешнего объекта. Это позволяет создавать семейства сильно связанных объектов.

Локальные классы:

Описываются в теле блока. Экземпляры имеют доступ к полям и методам внешнего объекта. Экземпляры имеют доступ к локальным переменным, снабженным модификатором final.

Однако, у локальных классов есть ряд важных особенностей и отличий от внутренних классов. Главное заключается в их объявлении:

* Локальный класс объявляется только в блоке кода. Чаще всего — внутри какого-то метода внешнего класса.
* Его можно объявить просто в блоке кода или даже в цикле for.

Итак, Объект локального класса не может создаваться за пределами метода или блока, в котором его объявили.

В Java 8 поведение локальных классов было изменено. В этой версии языка локальный класс имеет доступ не только к final-локальным переменным и параметрам, но и к effective-final. Effective-final называют переменную, значение которой не менялось после инициализации.

У локального класса есть доступ ко всем (даже приватным) полям и методам внешнего класса: и к статическим, и к нестатическим. Локальные классы похожи на внутренние классы, потому что они не могут определять или объявлять какие-либо статические члены. Локальные классы в статических методах могут ссылаться только на статические члены включающего класса.

Нельзя объявить интерфейс внутри блока; интерфейсы по своей природе статичны. Но если интерфейс объявлен внутри внешнего класса, локальный класс PhoneNumber может его реализовать. В локальных классах нельзя объявлять статические инициализаторы (блоки инициализации) или интерфейсы. Но у локальных классов могут быть статические члены при условии, что они постоянные переменные (static final).

Анонимные классы:

• Описываются непосредственно в выражении new и служат его частью

• Тип, указанный после new является базовым для объявляемого анонимного класса

• Могут расширять один класс или реализовывать один интерфейс

• Явно писать implements или extends нельзя

• Не могут иметь конструкторов

При создании анонимного класса внутри Java-машины происходит следующее:

* Создается безымянный Java-класс, реализующий интерфейс MonitoringSystem.
* Компилятор, увидев такой класс, требует от тебя реализовать все методы интерфейса MonitoringSystem (мы это и сделали 3 раза).
* Создается один объект этого класса.

MonitoringSystem generalModule = new MonitoringSystem() {

};

В конце стоит точка с запятой! Она стоит там не просто так. Мы одновременно объявляем класс (посредством фигурных скобок) и создаем его объект с помощью ();

В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным.

Есть у них кое-что общее и с локальными классами: они видны только внутри того метода, в котором определены.

И еще одно важное ограничение, которое досталось анонимным классам от их «предков» — внутренних классов: анонимный класс не может содержать статические переменные и методы.

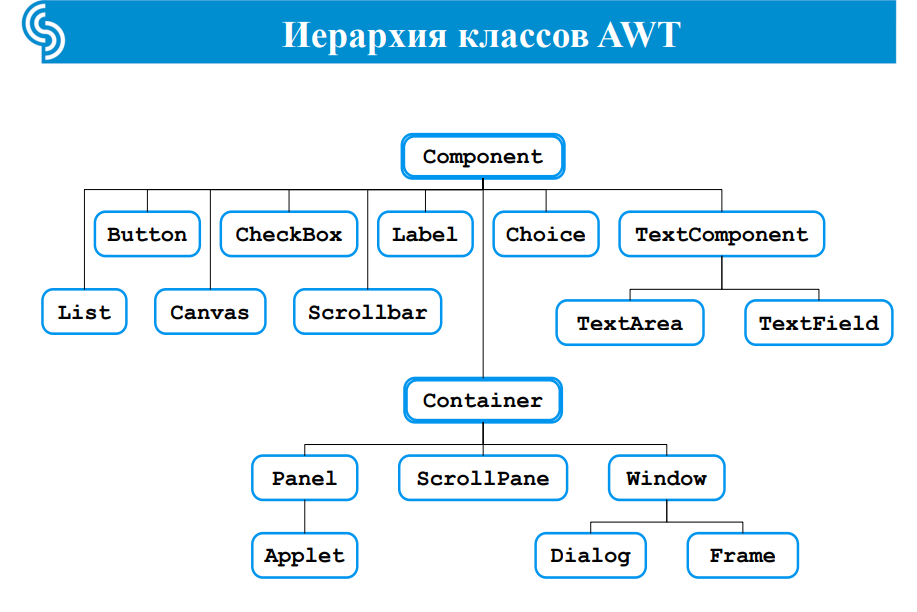
73. **Графические приложения: технологии AWT и Swing.**

Abstract Window Toolkit (AWT) - Технологии AWT (Abstract Window Toolkit) в Java — это *ранний набор классов для создания графических пользовательских интерфейсов (GUI)*, предоставляющий базовые элементы управления (кнопки, поля ввода), контейнеры и обработку событий; компоненты AWT *зависят от платформы* (отображаются нативно), а сама технология, хотя и уступила место более современным Swing и JavaFX, является основой для них и всё еще используется в устаревших проектах, предлагая управление окнами, цветами, шрифтами и отрисовкой графики через пакет java.awt.

Компоненты являются компонентами ОС. Большое количество native-кода. Отображение изменяется при смене ОС. Класс java.awt.Component определяет базовую функциональность компонентов.

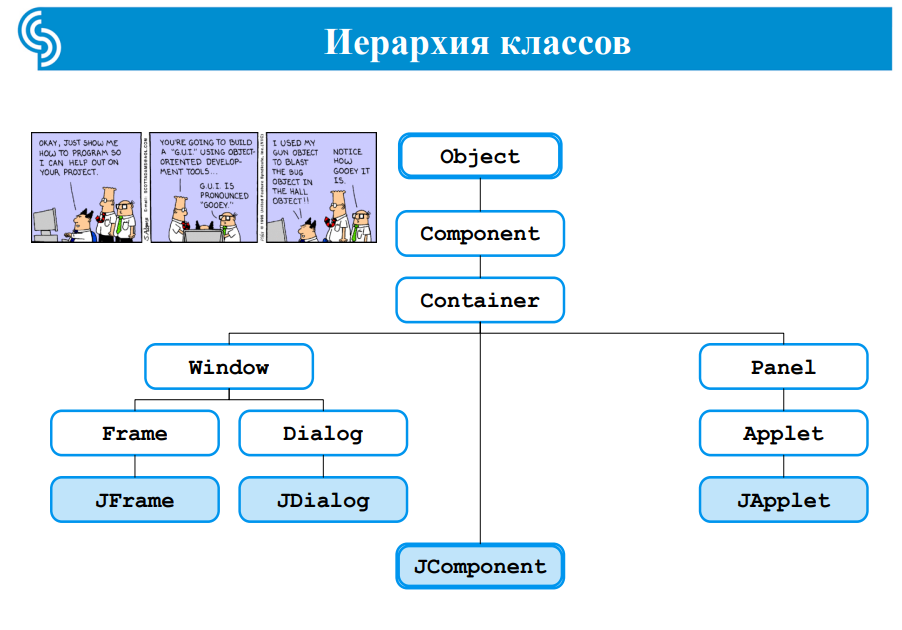
Проблемы AWT:

* Сходные элементы в различных ОС могут иметь некоторые различия
* ОС часто имеют элементы GUI, отсутствующие в других ОС
* Использование native-методов в AWT приводит к возникновению ошибок на конкретных платформах



Swing

Технологии Swing в Java — это мощная библиотека для создания графических пользовательских интерфейсов (GUI) с богатым набором компонентов, являющаяся частью Java Foundation Classes (JFC). Swing предоставляет множество виджетов (кнопки, поля ввода, таблицы), работает поверх AWT (Abstract Window Toolkit), но является более гибким и кроссплатформенным благодаря легковесным компонентам, написанным полностью на Java.

Элементы GUI отрисовываются в пустых окнах. «Нативные» функции используются только для вывода окна, отрисовки и получения информации о действиях пользователя. Набор элементов GUI более широк, чем в AWT, и может быть еще расширен. Сильная привязка к «нативным» методам отсутствует, что снижает вероятность возникновения ошибок. Отображение на различных платформах единообразно. 



74. **Отрисовка графических компонентов. Двойная буферизация. Pluggable look and feel.**

Отрисовка производится в методе paintComponent().Запрос на перерисовку:

* public void repaint()
* public void repaint(long tm)
* public void repaint(int x, int y, int width, int height)
* public void repaint(long tm, int x, int y, int width, int height)

Двойная буферизация:

В целях экономии времени на перерисовку логично запоминать однажды нарисованный статичный объект как рисунок в памяти. При использовании Swing для этого не надо предпринимать дополнительных действий по выделению памяти и.д. Используется т.н. механизм «двойной буферизации», реализующий сохранение информации на уровне механизмов отрисовки. Для одного участка «видимой области» приложения используется не более одного изображения-буфера.

Методы класса JComponent

• setDoubleBuffered(boolean aFlag) - Устанавливает, буферизует ли объект свой вывод

• boolean isDoubleBuffered() - Возвращает булевское значение, показывающее, используется ли двойная буферизация, или нетзуется не более одного изображения-буфера

Pluggable look and feel:

Программе можно придать различный вид, изменив «стиль» отрисовки компонентов

• Внешний вид программы может изменяться во время исполнения

• Перерисовка должна вызываться принудительно

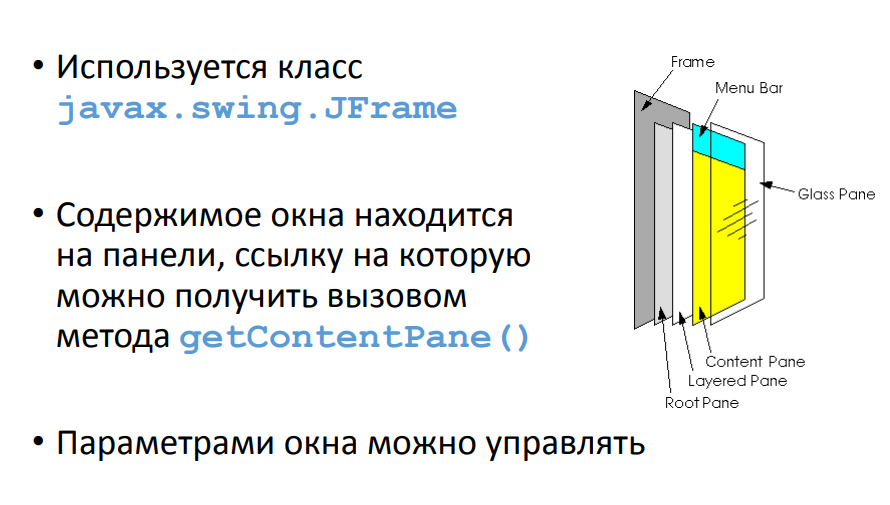
• Есть разработанные «стили» для базовых платформ

• Существует возможность создания собственных «стилей»Пример с кодом:

switch (value) {  
 case 0: s = "javax.swing.plaf.metal.MetalLookAndFeel";  
 break;  
 case 1: s = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";  
 break;  
 case 2: s = "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";  
 break;  
 default: s = UIManager.getSystemLookAndFeelClassName();  
 }  
UIManager.setLookAndFeel(s);  
SwingUtilities.updateComponentTreeUI(getContentPane());

75. **Оконные приложения, меню. Менеджеры компоновки.**

Оконные приложения — это приложения с графическим интерфейсом, использующие окна для отображения информации. Меню — это элемент интерфейса для навигации и выполнения команд. Менеджеры компоновки (Layout Managers) — это инструменты для управления расположением компонентов в контейнере.

Оконные приложения в Swing 

Пример кода:

import java.awt.\*;  
import javax.swing.\*;  
public class CenteredFrameTest {  
 public static void main(String[] args) {  
 CenteredFrame frame = new CenteredFrame();  
 frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
 frame.setVisible(true);  
 }  
}  
class CenteredFrame extends JFrame {  
 public CenteredFrame() {  
 Toolkit kit = Toolkit.getDefaultToolkit();  
 Dimension screenSize = kit.getScreenSize();  
 int screenHeight = screenSize.height;  
 int screenWidth = screenSize.width;  
 setSize(screenWidth / 2, screenHeight / 2);  
 setLocation(screenWidth / 4, screenHeight / 4);  
 Image img = kit.getImage("icon.gif");  
 setIconImage(img); setTitle("CenteredFrame");  
 }  
}

Работа с меню:

• javax.swing.JMenuBar - Панель меню в верхней части окна

• javax.swing.JMenu - Меню как таковое

• javax.swing.JMenuItem - Единичный элемент меню

• javax.swing.JCheckBoxMenuItem - Элемент меню «флажок»

• javax.swing.JRadioButtonMenuItem- Элемент меню «радио-кнопка»

• javax.swing.JPopup - Всплывающее меню

Менеджеры компоновки:

Управляют размещением компонентов в контейнере, учитывая параметры этих компонентов (например, предпочтительный размер). Реализуют интерфейс java.awt.LayoutManager. Устанавливаются с помощью метода setLayout() контейнера. Примеры простых компоновок:

* Простая поточная компоновка FlowLayout
* Граничная компоновка BorderLayout
* Компоновка в сетку GridLayout
* Компоновка «Колода карт» CardLayout
* Сетка с настраиваемыми размерами GridBagLayout

76. **Обработка событий, общие принципы. Участники и правила именования (с примерами). Классы-адаптеры.**

Обработка событий — это механизм, который позволяет компонентам реагировать на действия пользователя.

Участники:

Источник события (Event Source): Компонент, который генерирует событие.

* public void addTypeListener (TypeListener el) throws java.util.TooManyListenersException
* public void removeTypeListener(TypeListener el)

Слушатель события (Event Listener): Объект, который обрабатывает событие.

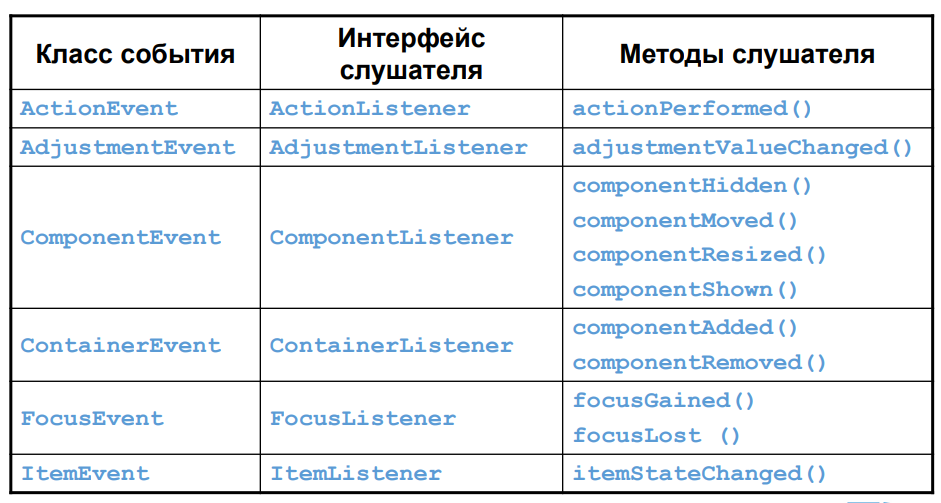
Должен реализовывать методы для приема и обработки уведомлений. Существует набор интерфейсов TypeListener, описывающих методы обработки событий. Класс слушателя должен реализовывать интерфейс, соответствующий событию.

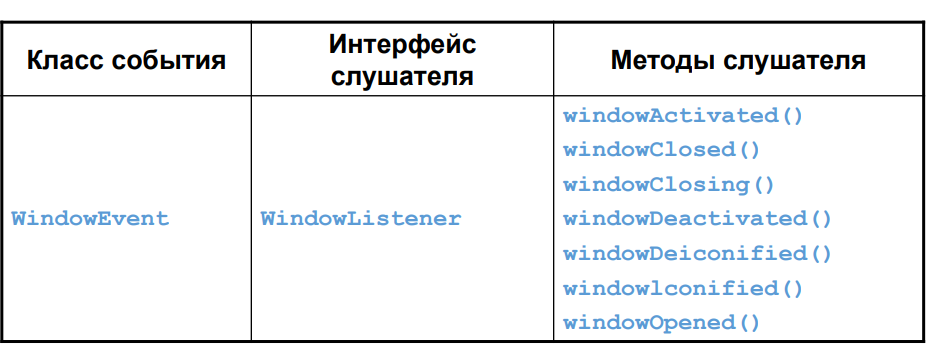
Объект события (Event Object): содержит информацию о событии.

Классы событий в пакетах

* java.awt.event
* javax.swing.event

Правила именования: Методы слушателя начинаются с actionPerformed, mouseClicked и т.д.





Классы адаптеры - Находятся в пакете java.awt.event. Определены для интерфейсов слушателей того же пакета, содержащих более одного метода. Являются пустыми реализациями соответствующего интерфейса. Наследники классов-адаптеров переопределяют необходимые методы.

77. **Апплеты, их особенности, структура и жизненный цикл. Тэг <applet>.**

Апплет (applet) – небольшое приложение, доступное на Интернет-сервере, транспортирующееся по Интернет, автоматически устанавливающееся и выполняемое как часть Web-документа. В HTML-документ апплет встраивается с помощью тегов <applet> и <object>. После доставки к клиенту апплет имеет ограниченный доступ к ресурсам системы .

Класс Applet:

java.applet.Applet

Является классом-предком для любого апплета, включаемого в web-страницу или просматриваемого в Java Applet Viewer. При наследовании обычно переопределяется ряд методов.

Структура Апплета:

* void init() - Вызывается один раз при инициализации.
* void start() - Вызывается каждый раз при выводе документа, содержащего апплет, на экран.
* void stop() - Вызывается каждый раз, когда браузер покидает документ, содержащий апплет.
* void destroy() - Вызывается один раз, когда выполнение апплета заканчивается.

Тег **<applet>** предназначен для вставки на страницу апплетов — небольших программ, написанных на языке Java. Этот тег является устаревшим, взамен необходимо использовать более гибкий тег **<object>**. Между открывающим и закрывающим тегом можно добавить текст, который будет отображаться в браузере, если он не поддерживает апплеты. В противном случае текст не выводится. Вместо <applet> в HTML5 следует использовать теги <object> (для встраивания различных объектов, включая плагины) или <embed> (для мультимедиа).

**Синтаксис**

**<applet** code="URL"**>**Текст**</applet>**

78. **Рефлексия, предназначение и возможности. Участники механизма рефлексии.**

• Рефлексия (от лат. Reflexio – обращение назад) – обращение субъекта на себя самого, на свое знание или на свое собственное состояние

• Рефлексия в Java – возможность программы анализировать саму себя, взаимодействуя с виртуальной машиной Java (JVM).

Возможности:

* Загрузка типов во время исполнения программы
* Исследование структуры типов и их элементов
* Создание экземпляров классов
* Вызов методов
* Загрузка классов из набора байтов

Класс java.lang.Class - Класс является метаклассом по отношению к другим типам. Экземпляры класса Class описывают классы и интерфейсы выполняемого приложения. Методы класса Class позволяют исследовать содержимое описываемого класса и его свойства.

Класс java.lang.ClassLoader - Реализует механизмы загрузки классов.

Пакет java.lang.reflect - Содержит ряд дополнительных и вспомогательных классов:

* Field - Описывает поле объекта
* Method - Описывает метод объекта
* Constructor - Описывает конструктор объекта
* Modifier - Инкапсулирует работу с модификаторами
* Array - Инкапсулирует работу с массивами

79. **Получение ссылки на описание типа. Получение информации о типе. Возможности класса Class.**

Метод Class Object.getClass() - Возвращает ссылку на представление класса, экземпляром которого является объект.

Псевдополе Object.class - Ссылка на представление указанного класса.

Метод static Class - Class.forName(...) Возвращает ссылку на представление класса, полное имя которого указывается параметром типа String.

Метод Class[] Class.getClasses() - Возвращает ссылку на массив ссылок на объекты Class вложенных типов.

Метод Class Class.getDeclaringClass() - Для вложенных типов возвращает ссылку на объект Class внешнего типа.

Метод Class[] Class.getInterfaces() - Возвращает ссылки на описания интерфейсов, от которых наследует тип.

Метод Class Class.getSuperclass() - Возвращает ссылку на описание родительского класса.

Пример кода с информацией о классе.

import java.lang.reflect.\*;  
class ListMethods {  
 public static void main(String[] argv)  
 throws ClassNotFoundException {  
 Class c = Class.*forName*(argv[0]);  
 Constructor[] cons = c.getConstructors( );  
 *printList*("Constructors", cons);  
 Method[] meths = c.getMethods( );  
 *printList*("Methods", meths);  
 Field[] fields = c.getFields();  
 *printList*("Fields", fields);  
 }  
 static void printList(String s, Object[] o) {  
 System.*out*.println("\*\*\* " + s + " \*\*\*");  
 for (int i = 0; i < o.length; i++)  
 System.*out*.println(o[i].toString( ));  
 }  
}

Возможности класса класс:

• Загрузка класса в JVM по его имени - static Class forName(String name).

• Определение вида типа - boolean isInterface(), boolean isLocalClass().

• Получение родительских типов - Class getSuperclass(), Class[] getInterfaces().

• Получение вложенных типов - Class[] getClasses().

• Создание объекта - Object newInstance().

• Получение списка всех полей и конкретного поля по имени - Field[] getFields(), Field getField(...).

• Получение списка всех методов и конкретного метода по имени и списку типов параметров - Method[] getMethods(), Method[] getMethod(...).

• Получение списка всех конструкторов и конкретного конструктора по списку типов параметров - Constructor[] getConstructors(), Constructor getConstructor(...).

80. **Создание экземпляров классов. Вызов методов.**

• Метод Object Class.newInstance() - Возвращает ссылку на новый экземпляр класса, используется конструктор по умолчанию.

Метод Object Constructor.newInstance(Object[] initArgs) - Возвращает ссылку на новый экземпляр класса, с использованием конструктора и указанными параметрами конструктора.

Прямой вызов - Если на момент написания кода известен тип-предок загружаемого класса. Приведение типа и вызов метода.

Вызов через экземпляр класса Method - Object Method.invoke(Object obj, Object[] args)

* obj – ссылка объект, у которого должен быть вызван метод
* принято передавать null, если метод статический
* args – список параметров для вызова методов

81. **Загрузчики классов и их возможности.**

Экземпляры класса отвечают за загрузку классов в виртуальную машину. Это абстрактный класс, не имеющий ни одного абстрактного метода. Классы-наследники должны в каком-то смысле расширять возможности виртуальной машины по загрузке классов. Объекты загрузчиков образуют иерархию (родительский объект указывается как параметр защищённых конструкторов).

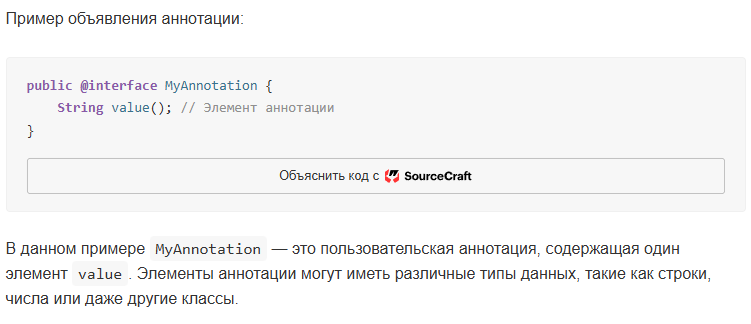
public Class loadClass(String name) - Проверяет, не был ли класс загружен раньше. Вызывает аналогичный метод родительского объекта. Вызывает метод findClass(), чтобы найти класс. Не стоит переопределять этот метод

protected Class findClass(String name) - Ищет и загружает класс по имени специфическим для данного загрузчика способом. Этот метод и нужно переопределять

protected final Class defineClass(String name, byte[] b, int off, int len) - Загружает класс из указанного набора байтов

82. **Аннотации.**

Аннотации в Java представляют собой метаданные, которые можно добавлять к классам, методам, полям и другим элементам кода. Они обеспечивают дополнительную информацию о коде, которая может быть использована компилятором, средствами разработки или даже во время выполнения программы. Для создания аннотации в Java используется аннотированный интерфейс, который определяет структуру аннотации.



Java предоставляет набор встроенных аннотаций, которые имеют специальное значение и часто используются в разработке.

— @Override: Эта аннотация указывает, что метод переопределяет метод из суперкласса. Она помогает предотвратить ошибки в случае, если вы ошибочно не переопределили метод.

— @Deprecated: Эта аннотация помечает элемент (класс, метод, поле и т. д.) как устаревший. Она предупреждает разработчиков о том, что использование этого элемента не рекомендуется, и в будущих версиях Java может быть удалено.

Пользовательские аннотации — это мощный механизм, который позволяет разработчикам внедрять собственные метаданные в код. Это открывает широкие возможности для улучшения читаемости, обеспечения безопасности и документирования кода, а также для создания собственных фреймворков и инструментов.

Применение аннотаций

* 1. Маркировка элементов
  2. Дополнительная информация
  3. Контроль компиляции
  4. Аннотации времени выполнения
  5. Обработка аннотаций

83. **Особенности однопоточного и многопоточного программирования. Реализация многопоточности. Многопоточность в Java.**

Проблемы однопоточного программирования:

* Монопольный захват задачей процессорного времени
* Смешение логически несвязанных фрагментов кода
* Попытка их разделения приводит к возникновению в программе новых систем и усложнению кода

Многопоточное программирование:

Последовательно выполняющиеся инструкции составляют поток. Потоки выполняются независимо. Потоки могут взаимодействовать друг с другом.

Квантование времени:

• Время разделяется на интервалы (кванты времени)

• Во время одного кванта обрабатывается один поток команд

• Решение о выборе потока принимается до начала интервала

• Переключения между потоками с высокой частотой

• Иллюзия одновременности!

Для создания нового потока (потока платформы) в Java мы можем использовать следующие варианты:

* Реализация интерфейса Runnable
* Создание класса, который наследуется от Thread

А ещё Runnable является функциональным интерфейсом начиная с Java 1.8. Это позволяет писать код задач для потоков ещё красивее:

public static void main(String []args){

Runnable task = () -> {

System.out.println("Hello, World!");

};

Thread thread = new Thread(task);

thread.start();

}

Особенности многопоточности:

* Простота выделения подзадач
* Более гибкое управление выполнением задач
* Более медленное выполнение
* Выигрыш в скорости выполнения при разделении задач по используемым ресурсам
* Выигрыш в скорости выполнения на многоядерных системах
* Недетерминизм при выполнении

84. **Класс Thread. Запуск нитей и управление нитями. Создание нитей с помощью интерфейса Runnable. Управление потоками.**

Использование класса Thread:

• Описание класса

public class ИмяКласса extends Thread {  
 public void run() {  
 // Действия, выполняемые потоком  
 }  
}

• Запуск потока

ИмяКласса t = new ИмяКласса();  
t.start(); // именно start(), а не run()!!!

Использование интерфейса Runnable:

• Описание класса

public class ИмяКласса implements Runnable {  
 public void run() {  
 // Действия, выполняемые потоком  
 }  
}

• Запуск потока

Runnable r = new ИмяКласса(); // Это ещё не поток  
Thread t = new Thread(r); // А вот это уже поток  
t.start();

Особенности Runnable:

Возможность создать класс, описывающий тело потока и наследующий от класса, отличного от Thread. Объект вашего класса не является объектом потока. Невозможно использовать напрямую

методы класса Thread. Можно получить ссылку на объект текущего потока с помощью статического метода currentThread() класса Thread.

Управление потоками:

* void start() - Запускает выполнение потока
* void stop() - Прекращает выполнение потока
* void suspend() - Приостанавливает выполнение потока
* void resume() - Возобновляет выполнение потока
* void join() - Останавливает выполнение текущего потока до завершения потока, у объекта которого был вызван метод
* static void sleep(long millis) - Останавливает выполнение текущего потока как минимум на millis миллисекунд
* static void yield() - Приостанавливает выполнение текущего потока, предоставляет возможность выполнять другие потоки

85. **Группы потоков. Приоритеты потоков. Демон-потоки и демон-группы потоков.**

• Каждый поток находится в группе

• Группы потоков образуют дерево, корнем служит начальная группа

• Поток не имеет доступа к информации о родительской группе

• Изменение параметров и состояния группы влияет на все входящие в нее потоки

Создание группы:

//Без явного указания родительской группы  
ThreadGroup group1 = new ThreadGroup("Group1");  
//С явным указанием родительской группы  
ThreadGroup group2 = new ThreadGroup(group1, "Group2");

Создание потока:

//Без явного указания группы  
MyThread t1 = new MyThread("Thread1");  
//С явным указанием группы  
MyThread t2 = new MyThread(group2, "Thread2");

Операции в группе потоков:

* int activeCount() - Возвращает оценку количества потоков
* int enumerate(Thread[] list) - Копирует в массив активные потоки
* int activeGroupCount() - Возвращает оценку количества подгрупп
* int enumerate(ThreadGroup[] list) - Копирует в массив активные подгруппы
* void interrupt() - Прерывает выполнение всех потоков в группе

Приоритеты потоков:

Приоритет – количественный показатель важности потока. Недетерминированно воздействуют на системную политику упорядочивания потоков. Базовый алгоритм программы не должен зависеть от схемы расстановки приоритетов потоков. При задании значений приоритетов рекомендуется использовать константы.

Константы в классе Thread

* MAX\_PRIORITY
* MIN\_PRIORITY
* NORM\_PRIORITY

Методы потока

* int getPriority()
* void setPriority(int newPriority)

Методы группы потоков

* int getMaxPriority()
* void setMaxPriority(int priority)

Демон-поток:

Демон-потоки позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них. Уничтожаются виртуальной машиной, если в группе не осталось не-демон потоков

void setDaemon(boolean on) - Устанавливает вид потока. Вызывается до запуска потока.

boolean isDaemon() - Возвращает вид потока: true – демон, false – обычный

Демон-группа:

Демон-группа автоматически уничтожается при остановке последнего ее потока или уничтожении последней подгруппы потоков.

void setDaemon(boolean on) -Устанавливает вид группы

boolean isDaemon() - Возвращает вид группы: true – демон, false – обычная

86. **Работа с разделяемыми ресурсами. Блокировки. Механизмы синхронизации, типичные ошибки.**

Неконтролируемое совместное использование ресурсов:

Недетерминизм программы - конечный результат работы программы непредсказуем.

Некорректность работы программы - возможность некорректной работы алгоритма, возникновения исключительных ситуаций.

Виды локов:

synchronized блок: это ключевое слово в Java, которое может использоваться для ограничения доступа к коду только одним потоком. Этот блок может быть использован для синхронизации методов или блоков кода.

ReentrantLock: это класс, реализующий интерфейс Lock, который предоставляет более гибкие возможности блокировки. Он позволяет использовать более сложные сценарии блокировки, такие как блокировка с ожиданием определенного времени, блокировка с попыткой прервать ожидание и другие.

ReadWriteLock: это интерфейс, предоставляющий две блокировки - одну для чтения и одну для записи. Эта схема позволяет нескольким потокам выполнять операции чтения одновременно, но блокирует доступ на запись во время обновления данных.

StampedLock: lock с поддержкой write/read и ultra-fast optimistic read, который ускоряет чтение и снижает contention в многопоточности.

• Только один поток в один момент времени может установить блокировку на некоторый объект.

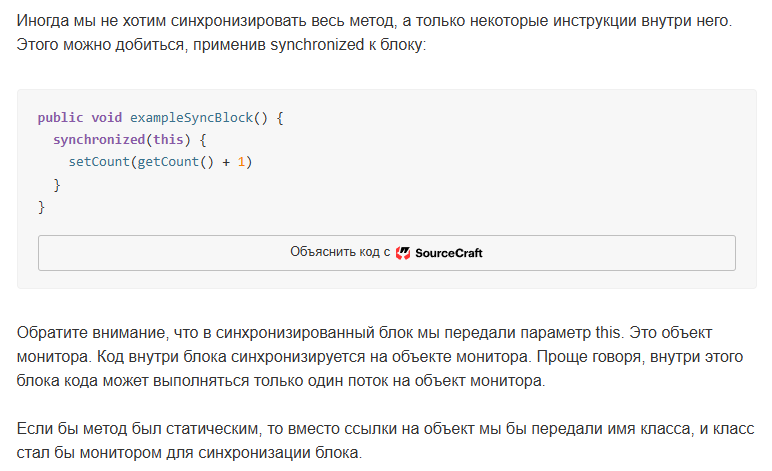
• Попытка блокировки уже заблокированного объекта приводит к остановке потока до момента разблокирования этого объекта.

• Наличие блокировки не запрещает всех остальных действий с объектом.



Синхронизированный блок:

//Блокируется указанный объект  
synchronized (ссылкаНаОбъект) {  
 // Тело блока синхронизации  
}



Синхронизованный метод:

//Блокируется объект-владелец метода  
public synchronized void метод() {  
 // Тело метода  
}

Новые виды ошибок:

* Отсутствие синхронизации
* Необоснованная длительная блокировка объектов
* Взаимная блокировка (deadlock)
* Возникновение монопольных потоков
* Нерациональное назначение приоритетов

87. **Методы класса Object для работы с потоками. Особенности использования. Прерывание потоков.**

Каждый объект имеет набор ожидающих потоков исполнения (wait-set). Любой поток может вызвать метод wait() любого объекта и попасть в его wait-set, остановившись до пробуждения. Метод объекта notify() пробуждает один, случайно выбранный поток из wait-set объекта. Метод объекта notifyAll() пробуждает все потоки из wait-set объекта.

Метод может быть вызван потоком у объекта только после установления блокировки на этот объект. Потоки, прежде чем приостановить выполнение после вызова метода wait(), снимают все свои блокировки. После вызова освобождающего метода потоки пытаются восстановить ранее снятые блокировки

Запрещенные действия над потоками:

• Thread.suspend(), Thread.resume() - Увеличивает количество взаимных блокировок

• Thread.stop() - Использование приводит к возникновению поврежденных объектов

Корректное прерывание потока:

• public void interrupt() - Изменяет статус потока на прерванный

• public static boolean interrupted() - Возвращает и очищает статус потока (прерван или нет)

• public boolean isInterrupted() = Возвращает статус потока (прерван или нет)

Поток должен в ходе своей работы проверять свой статус и корректно завершать работу, если его прервали.

В том случае, если в текущий момент поток выполняет методы wait(), sleep(), join(), а его прерывают вызовом метода interrupt()…

метод прерывает свое выполнение с выбросом исключения InterruptedException !

Потоку не сообщается, что его прервали!

88. **UML. Задачи, применение и структура. Виды диаграмм.**

UML (Unified Modeling Language) – унифицированный язык моделирования.

Язык моделирования - Нотация, используемая методами для описания проектов.

Процесс - Рекомендация относительно этапов, которые необходимо выполнить при разработке проекта.

Задачи UML (Unified Modeling Language) — это **визуализация, спецификация, конструирование и документирование сложных систем** в IT и бизнесе, позволяющие командам говорить на одном языке, проектировать структуру (классы, компоненты) и поведение (процессы, потоки) с помощью стандартных диаграмм (классов, вариантов использования, последовательности) для улучшения коммуникации и точности реализации.

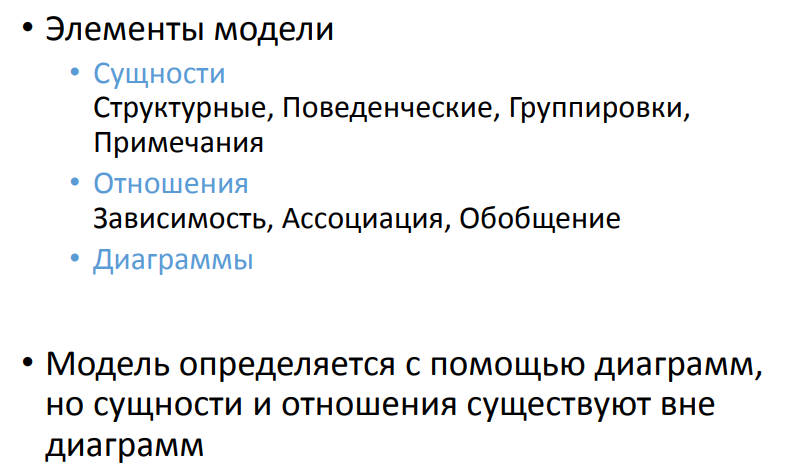
Области применения:

• Общение между разработчиками. Передача информации в графическом виде на различных уровнях разработки.

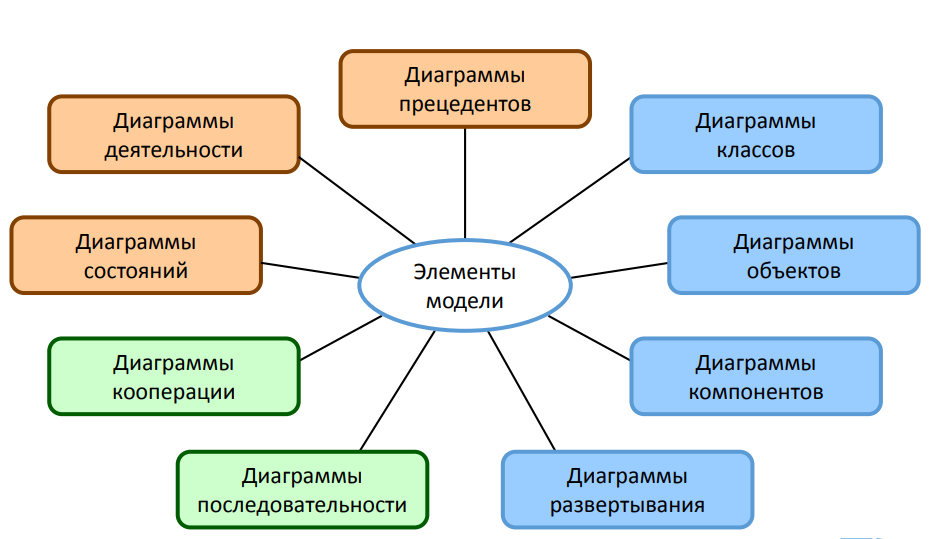
• Объектно-ориентированный подход. Работа с ОО-приемами в графичекой форме.

• Общение с экспертами предметной области. Описание вариантов использования (концептуальный подход).

Модель в UML:



Виды диаграмм:



89. **Диаграммы классов.**

Диаграмма классов (class diagram):

• Описывает типы объектов системы и различного рода существующие между ними статические отношения.

• Статические отношения: ассоциации и подтипы.

• Также отображаются атрибуты и операции классов и ограничения, накладываемые на связи между объектами.

Подходы к использованию:

• Концептуальная точка зрения - Представление понятий предметной области.

• Точка зрения спецификации - Программная система на уровне интерфейсов.

• Точка зрения реализации - Программная система на уровне реализации.

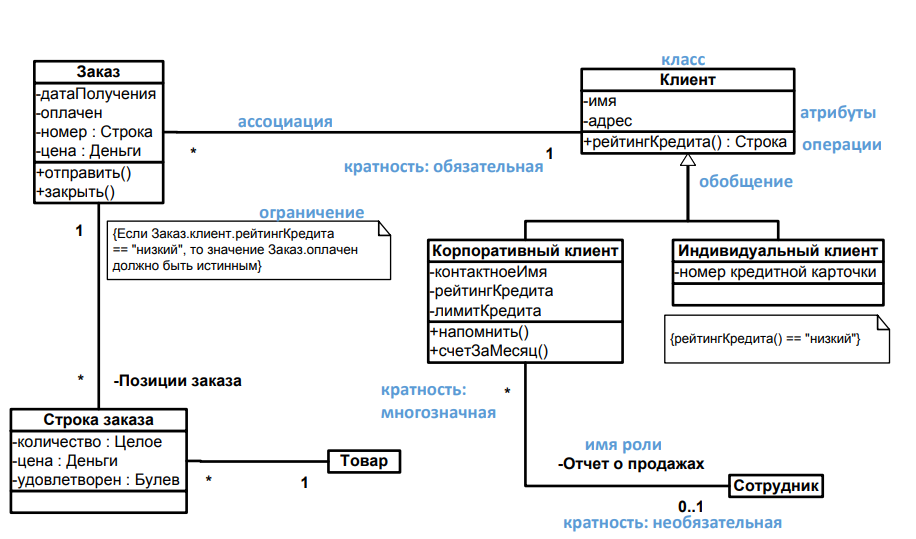
Применение диаграмм классов:

+ Являются основой почти всех объектноориентированных методов.

– Весьма богатая нотация.

• Выбор точки зрения модели должен соответствовать конкретному этапу работы над проектом.

• Следует концентрироваться на главных аспектах.

Пример диаграммы классов

90. **Диаграммы взаимодействия: диаграммы последовательности и кооперации.**

Диаграммы взаимодействия

Представляют собой модели, предназначенные для описания поведения взаимодействующих

групп объектов. Обычно описывают поведение только в рамках одного прецедента.

Существует два вида диаграмм взаимодействия:

* диаграммы последовательности
* диаграммы кооперации

Диаграмма последовательности

Порядок действий определяется расположением. Показывает скорее порядок действий, чем взаимодействующие объекты.

Элементы диаграммы:

* Линия жизни
* Сообщения
* Активность
* Условия сообщений
* Итерации
* Возврат
* Создание
* Удаление
* Самовызов

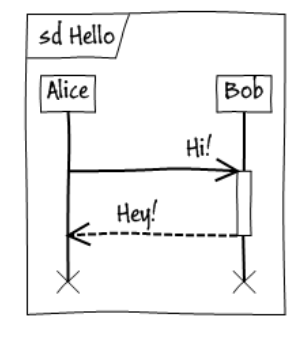
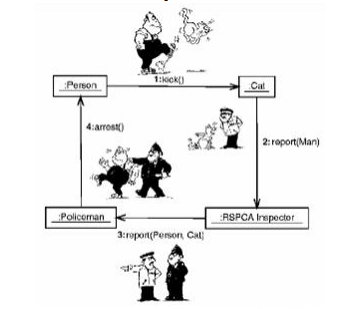


Диаграмма кооперации

Временная последовательность указывается нумерацией. Показывает скорее группы взаимодействующих объектов, чем порядок действий. Нумерация может быть вложенной. Допускается управляющая информация.



Применение диаграмм взаимодействия

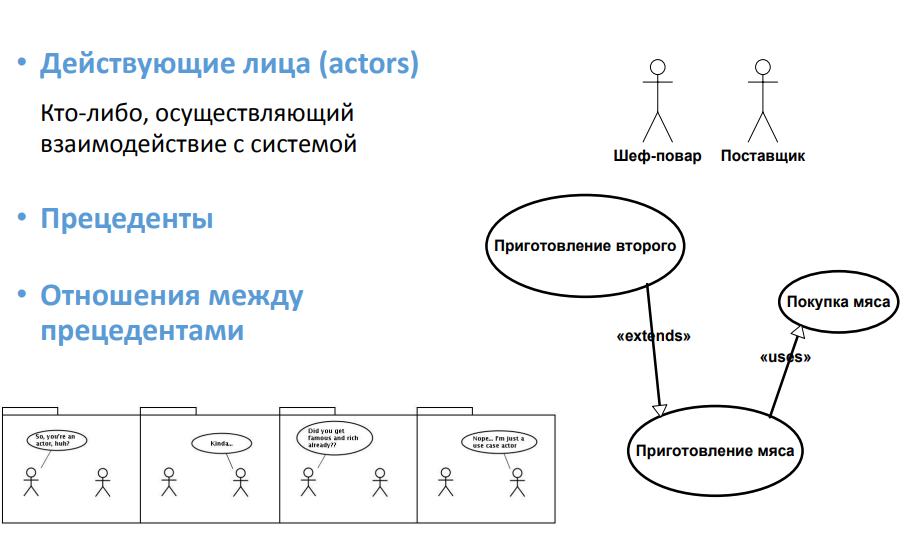
* Когда вы хотите описать поведение нескольких объектов в рамках прецедента
* Если необходимо описать один объект в различных прецедентах? - Диаграмма состояний
* Если необходимо описать несколько прецедентов или несколько нитей процесса? - Диаграмма деятельности

91. **Диаграммы прецедентов. Диаграммы состояний. Диаграммы действий.**

Прецедент (вариант использования, use case) – это модель поведения системы. Представляет собой последовательность действий системы и пользователя, направленных на осуществление бизнес-операции. Может включать в себя несколько сценариев

Диаграмма прецедентов (диаграмма вариантов использования, use case diagram) - Не описывает прецеденты как таковые Но описывает их взаимоотношения и участников

Элементы диаграммы прецедентов:



Пример:

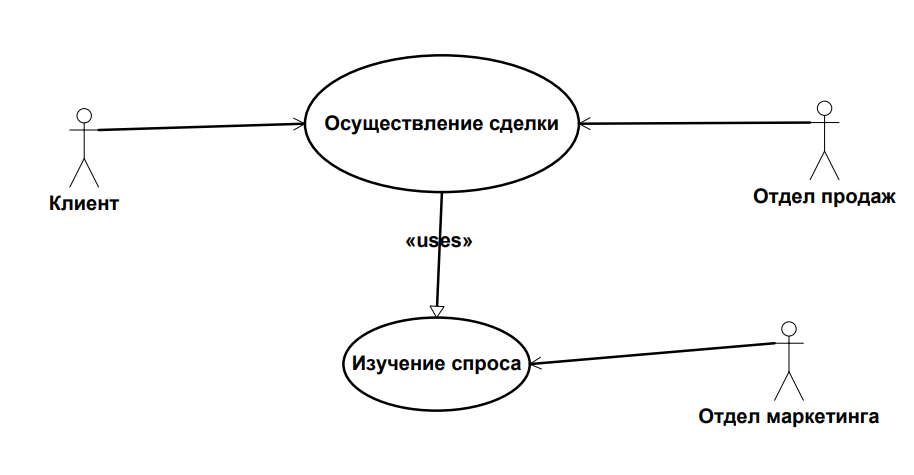


Диаграмма состояний (state diagram) - описывает состояния объекта и переходы между ними под воздействием внешних событийю Переходы могут сопровождаться деятельностью. Переходы могут иметь условия перехода. Состояния могут сопровождаться деятельностью.

Два выделенных состояния:

* начало (обязательно)
* и конец (необязательно)

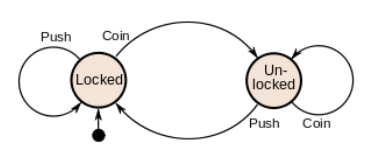
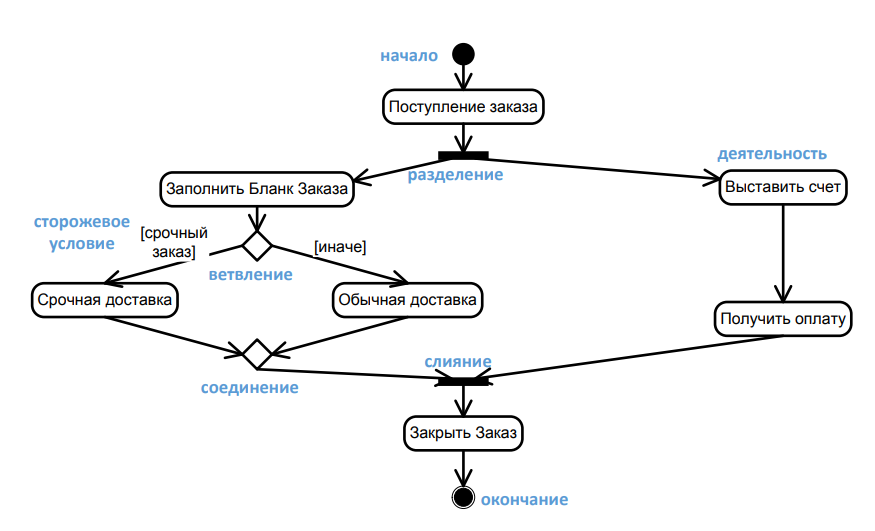


Диаграмма деятельности (activity diagram) - Описывает выполняемую деятельность и последовательность ее выполнения. Допускает выбор направления деятельности. Допускает одновременность выполнения различной деятельности. Сходна с диаграммой состояний, если рассматривать выполнение деятельности как состояние системы.



92. **Диаграммы компонентов. Диаграммы развертывания. Диаграммы объектов.**

Диаграмма компонентов(component diagram) - Показывает различные компоненты системы и зависимости между ними. Компонент представляет собой физический модуль программного кода. Зависимости между компонентами показывают, как изменения одного компонента могут повлиять на изменения других компонентов.

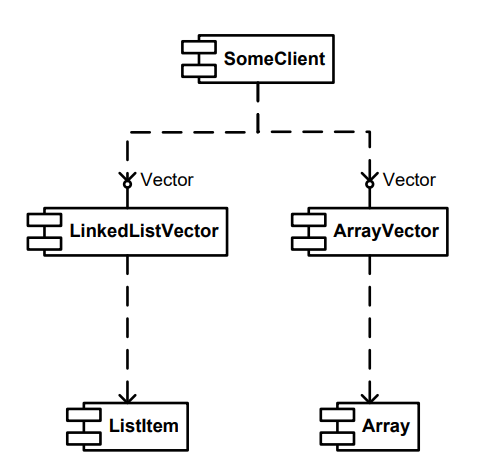


Диаграмма развертывания (deployment diagram) -

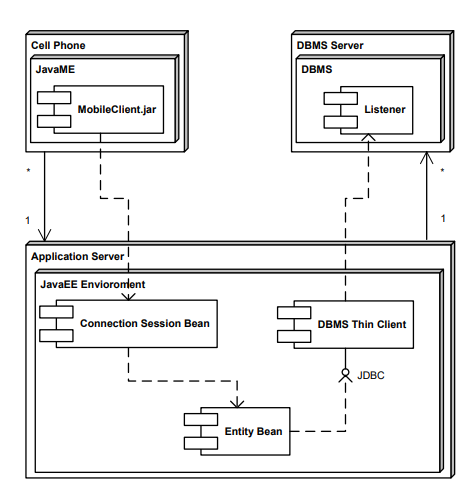
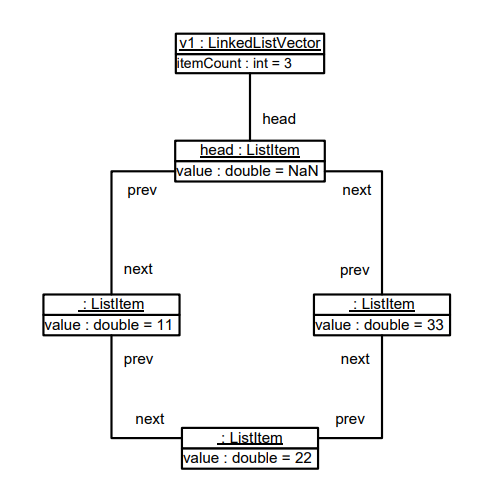
Отражает физические взаимосвязи между программными и аппаратными компонентами системы. Средство для представления маршрутов перемещения объектов и компонентов в распределенной системе. Узел представляет собой некоторый тип вычислительного устройства – в большинстве случаев самостоятельную часть аппаратуры.

Диаграмма объектов (object diagram) - Мгновенный снимок объектов системы с точки зрения времени. Могут быть указаны значения атрибутов и связей. 

93. **Повторное использование кода.**

Использование кода существующих приложений для создания новых приложений

Принципы создания хорошего повторно используемого кода

* Модульность (modularity)
* Слабая связность (low coupling)
* Высокая сфокусированность (high cohesion)
* Сокрытие информации (information hiding)
* Разделение ответственности (separation of concerns)

Примеры повторного использования

* «Копипаста» (copy-and-paste)
* Библиотеки (software libraries)
* Паттерны проектирования (design patterns)
* Фреймворки (software frameworks)

Механизмы повторного использования в ООП

* Наследование
* Композиция
* Шаблоны

94. **Паттерны и фреймворки.**

Фреймворк (Framework, каркас) – это набор взаимодействующих классов, составляющих повторно используемое проектное решение для конкретного класса программ. Диктует определенную структуру приложения или модуля. Определяет общую структуру, ее разделение на классы и объекты, основные функции тех и других, методы взаимодействия потоков и классов, потоки управления.

Паттерн проектирования (Design Pattern, образец проектирования, шаблон проектирования) – описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого проектного решения

Результат проектирования на уровне ООП – распределение ответственностей и активностей по классам. Паттерн – именованная конфигурация распределения ответственности по классам

Паттерны проектирования более абстрактны, чем фреймворки. Как архитектурные элементы паттерны проектирования мельче, чем фреймворки. Паттерны проектирования менее специализированны, чем фреймворки.

95. **Порождающие паттерны проектирования. Паттерн «одиночка» (Singleton).**

Одиночка – паттерн, порождающий объекты

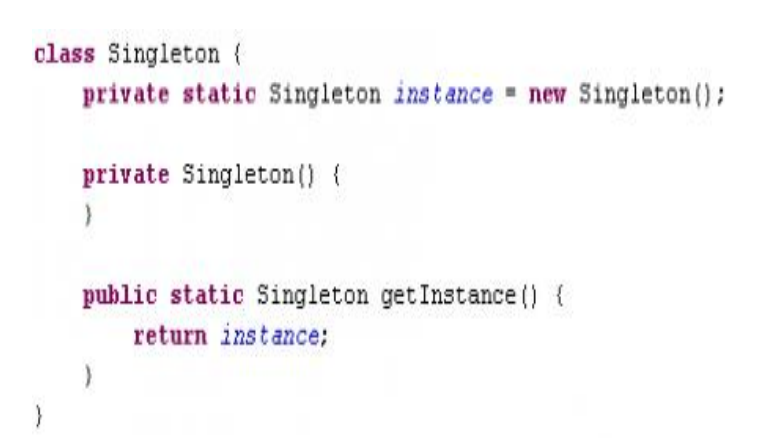
Назначение

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа

Применимость

* Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам.
* Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

Singleton – одиночка, определяет операцию Instance , которая позволяет клиентам получать доступ к единственному экземпляру.

****

96. **Порождающие паттерны проектирования. Паттерн «фабричный метод» (Factory method).**

Название и классификация

Фабричный метод – паттерн, порождающий объекты.

Назначение

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование в подклассы.

Известен также под именем Virtual Constructor.

Применимость

Используйте паттерн фабричный метод, когда:

* классу заранее неизвестно, объекты каких классов ему нужно создавать
* класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами
* класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя

97. **Структурные паттерн проектирования. Паттерн «адаптер» (Adapter).**

Адаптер – паттерн, структурирующий классы и объекты

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна

Известен также под именем Wrapper

Применимость

Применяйте адаптер классов, когда:

• хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям

• собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы

Применяйте адаптер объектов, когда:

• нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

98. **Структурные паттерны проектирования. Паттерн «декоратор» (Decorator).**

Декоратор – паттерн, структурирующий объекты

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Известен также под именем Wrapper

Применимость

Используйте паттерн декоратор:

• для динамического, прозрачного для клиентов добавления обязанностей объектам

• для реализации обязанностей, которые могут быть сняты с объекта

• когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно

99. **Структурные паттерны проектирования. Паттерн «заместитель» (Proxy).**

Заместитель – паттерн, структурирующий объекты

Назначение

Является суррогатом другого объекта и контролирует доступ к нему

Известен также под именем Surrogate

Применимость

Заместитель применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простую ссылку.

Удаленный заместитель - предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве.

Виртуальный заместитель - создает тяжелые объекты по требованию

Защищающий заместитель - контролирует доступ к исходному объекту

Умная ссылка - замена обычного указателя:

* подсчет числа ссылок на реальный объект
* загрузка объекта в память при первом обращении к
* нему
* проверка и установка блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его

100. **Поведенческие паттерны проектирования. Паттерн «итератор» (Iterator).**

Итератор – паттерн поведения объектов

Назначение

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления

Известен также под именем Cursor

Применимость

Используйте итератор:

• Для доступа к содержимому агрегированных объектов без раскрытия их внутреннего

представления

• Для поддержки нескольких активных обходов одного и того же агрегированного объекта

• Для предоставления единообразного интерфейса с целью обхода различных агрегированных структур (для поддержки полиморфной итерации)

101. **Поведенческие паттерны проектирования. Паттерн «наблюдатель» (Observer).**

Наблюдатель – паттерн поведения объектов

Назначение

Определяет зависимость типа “один ко многим” между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются

Известен также под именем Dependents, Publish-Subscribe, Listener

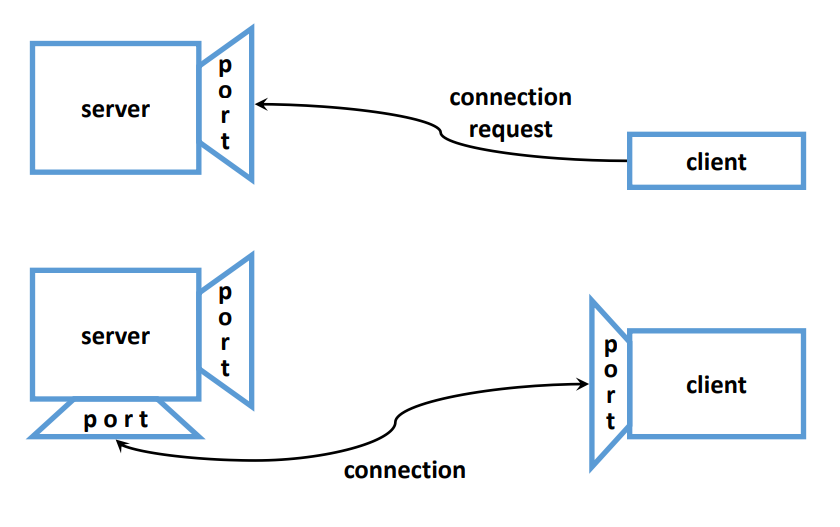
Применимость  
• Когда у абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо.

• Когда при модификации одного объекта требуется изменить другие и вы не знаете, сколько именно объектов нужно изменить.

• Когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, вы не хотите, чтобы объекты были тесно связаны между собой.

102. **Понятие сокета. Действия с сокетами и их особенности. Виды сокетов.**

Сетевое соединение – это процесс передачи данных по сети между двумя компьютерами или процессами • Сокет – конечный пункт передачи данных • Для программ сокет – одно из окончаний сетевого соединения • Для установления соединения каждая из сетевых программ должна иметь свой собственный сокет. • Связь между двумя сокетами может быть ориентированной на соединение • Связь между двумя сокетами может быть не ориентированной на соединение • Сокет связан с номером порта



**Понятие сокета**

**Сокет** — это программный интерфейс (объект), позволяющий приложениям обмениваться данными по сети. В ООП сокет инкапсулирует логику сетевого взаимодействия, предоставляя методы для подключения, отправки и получения данных. Он абстрагирует низкоуровневые детали сетевых протоколов и является конечной точкой связи между двумя программами, работающими в сети.

**Действия с сокетами и их особенности**

1. **Создание сокета**  
   Инициализация объекта сокета с указанием типа (например, потоковый или дейтаграммный) и протокола.
2. **Привязка (bind)**  
   Связывание сокета с конкретным IP-адресом и портом на локальной машине. Для клиентских сокетов часто выполняется автоматически.
3. **Прослушивание (listen, только для серверных TCP-сокетов)**  
   Переведение сокета в режим ожидания входящих подключений с указанием размера очереди запросов.
4. **Установка соединения (connect, для клиентских TCP-сокетов)**  
   Подключение к удалённому серверному сокету по указанному адресу и порту.
5. **Принятие соединения (accept, для серверных TCP-сокетов)**  
   Принятие входящего подключения от клиента, создание нового сокета для обмена данными с этим клиентом.
6. **Отправка и получение данных (send/write, recv/read)**
   * Для TCP: гарантированная передача потоковых данных через методы InputStream/OutputStream.
   * Для UDP: отправка/получение отдельных пакетов (дейтаграмм) без установки соединения.
7. **Закрытие сокета (close)**  
   Корректное завершение работы с освобождением ресурсов и разрывом соединения (для TCP).

Порядок работы с клиентским сокетом

* Открытие сокета
* Открытие потока ввода и/или потока вывода для
* сокета
* Чтение и запись в потоки согласно
* установленному протоколу общения с сервером
* Закрытие потоков ввода-вывода
* Закрытие сокета

**Особенности**:

* **Блокирующие и неблокирующие сокеты** — могут ожидать данные или возвращать управление немедленно.
* **TCP обеспечивает надёжность** (контроль целостности, порядок пакетов), UDP — нет.
* **Сокеты работают в рамках стека протоколов TCP/IP**.

**Виды сокетов**

1. **Стримовые сокеты (TCP-сокеты)**
   * Ориентированы на установление соединения (виртуальный канал).
   * Гарантируют доставку данных без потерь и в правильном порядке.
   * Используются там, где важна надёжность: веб-браузеры, FTP, передача файлов.
2. **Дейтаграммные сокеты (UDP-сокеты)**
   * Работают без установления соединения (отправка отдельных пакетов).
   * Не гарантируют доставку, порядок и целостность данных.
   * Используются там, где важна скорость и минимальные задержки: стриминг, онлайн-игры, DNS.

**Пример в Java**:

* java.net.Socket — для TCP.
* java.net.DatagramSocket — для UDP.

Существует несколько типов сокетов:

* **TCP-сокеты** — надёжная доставка данных с контролем последовательности
* **UDP-сокеты** — быстрая передача без гарантий доставки
* **Unix-сокеты** — локальное взаимодействие между процессами на одной машине

103. **Классы Socket и ServerSocket, работа с ними.**

Класс Socket - реализует клиентский сокет и его функции

Конструкторы

* Socket()
* Socket(InetAddress address, int port)
* Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
* Socket(String host, int port)
* Socket(String host, int port, InetAddress localAddr, int localPort)

Методы

* void close()
* InetAddress getLocalAddress()
* InputStream getInputStream()
* OutputStream getOutputStream()
* static void setSocketImplFactory(SocketImplFactory fac)
* И прочие…

Класс ServerSocket - Реализует серверный сокет и его функции

Конструкторы

* ServerSocket()
* ServerSocket(int port)
* ServerSocket(int port, int backlog)

Методы

* void close()
* Socket accept()
* void bind(SocketAddress endpoint)
* И прочие…

Порядок работы с клиентским сокетом:

* Открытие сокета
* Открытие потока ввода и/или потока вывода для сокета
* Чтение и запись в потоки согласно установленному протоколу общения с сервером
* Закрытие потоков ввода-вывода
* Закрытие сокета