**Основы ООП**

1. Понятие JVM, JRE, JDK

2. Компиляция и выполнение приложения java

3. Типы данных и операторы

4. Понятие класса и объекта

5. Члены класса

6. Ограничение доступа к членам класса

7. Конструкторы

8. Перегрузка методов

9. Статические члены класса

10. Константные члены класса

11. Перечисления

12. Наследование

13. Полиморфизм и позднее связывание

14. Абстрактные члены класса

15. Интерфейсы

**Структуры данных: определение и операции**

16. Массив

17. Вектор (динамический массив)

18. Список

19. Очередь

20. Множество

21. Хэш–таблица (отображение, карта)

**1. Понятие JVM, JRE, JDK**

**Java Runtime Environment**, сокращенно JRE – это исполнительная среда Java в которой выполняются программы, написанные на этом языке. Среда состоит из виртуальной машины – Java Virtual Machine(JVM) и библиотеки Java классов. JRE является частью JDK. По сути это минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Именно JRE или его аналог других фирм используется в браузерах, умеющих выполнять программы на Java, операционных системах и системах управления базами данных. Хотя JRE входит в состав JDK, фирма SUN распространяет этот набор и отдельным файлом. Это вызвано тем, что установка JRE является необходимым и достаточным условием для выполнения Java программ.  Однако для разработки программ JRE недостаточно, необходимо установить JDK, который может установить и JREи дополнительные компоненты.

**Java Virtual Machine**, сокращенно JVM – это виртуальная машина Java — основная часть исполняющей среды JRE. Виртуальная машина Java интерпретирует и исполняет байт-код Java. Байт код получают посредством компиляции исходного кода программы с помощью компилятора Java (стандартный - javac). В отличие от классических [runtime-библиотек](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29" \o "Библиотека (программирование)), библиотеки Java-классов входят в состав JRE.

**Java Development Kit**, сокращенно JDK – это бесплатно распространяемый корпорацией [Sun](http://www.java.sun.com/" \o "Sun) комплект разработчика приложений на языке Java, включающий в себя компилятор Java(javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE). В состав JDK не входит интегрированная среда разработки на Java (IDE), поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.

* JDK нужен для разработки (это компилятор, отладчик и т.д.).
* JRE нужен для запуска Java программ (содержит в себе JVM).
* JDK и JRE содержат JVM, которая нужна для запуска программ на Java.
* JVM является сердцем языка программирования Java и обеспечивает независимость от платформы.

**2. Компиляция и выполнение приложения java**

Чтобы скомпилировать программу Example, запустите компилятор (javac), указав имя исходного файла в командной строке:

C:\>javac Example.Java

Компилятор javac создаст файл Example.class, содержащий байт-кодовую версию программы. Как мы уже отмечали, байт-код Java — это промежуточное представление программы, содержащее инструкции, которые будет выполнять виртуальная машина Java. Следовательно, результат работы компилятора j avac не является непосредственно исполняемым кодом.

Чтобы действительно выполнить программу, необходимо воспользоваться программой запуска приложений Java, которая носит имя java. При этом ей потребуется передать имя класса Example в качестве аргумента командной строки, как показано в следующем примере:

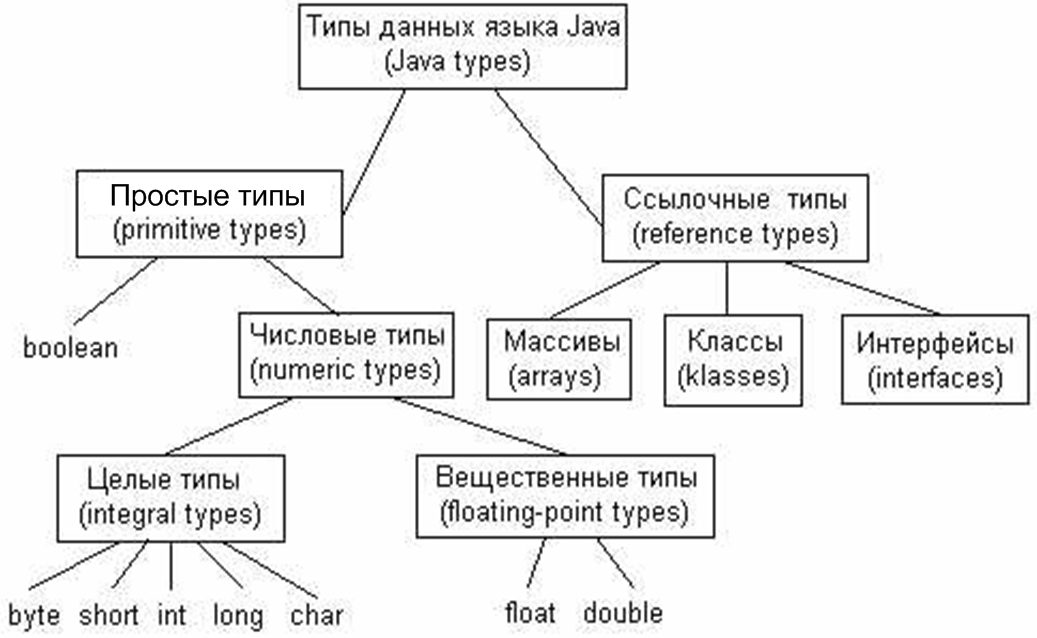
C:\>java Example

При выполнении программы на экране отобразится следующий вывод:

Простая Java-программа.

В процессе компиляции исходного кода каждый отдельный класс помещается в собственный выходной файл, названный по имени класса и получающий расширение .class. Именно поэтому исходным файлам Java целесообразно присваивать имена, совпадающие с именами классов, которые они содержат — имя исходного файла будет совпадать с именем файла с расширением .class. При запуске Java описанным способом в командной строке в действительности указывают имя класса, который нужно выполнить. Программа будет автоматически искать файл с указанным именем и расширением . class. Если программа найдет файл, она выполнит код, содержащийся в указанном классе.

**3. Типы данных и операторы**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип** | **Размер, байт** | | **Диапазон значений** | **Значение по умолчанию** | **Описание** |
| *byte* | 1 | | -128 .. 127 | 0 | Самое маленькое целое — один байт |
| *short* | 2 | | -32,768 .. 32,767 | 0 | Короткое целое, два байта |
| *int* | 4 | | — 2\*109 .. 2\*109 | 0 | Целое число, 4 байта |
| *long* | 8 | | — 9\*1018 .. 9\*1018 | 0L | Длинное целое, 8 байт |
| ***float*** | 4 | | -10127 .. 10127 | 0.0f | Дробное, 4 байта |
| ***double*** | 8 | | -101023 .. 101023 | 0.0d | Дробное, двойной длины, 8 байт |
| boolean | 1 | | true, false | false | Логический тип (только true & false) |
| char | 2 | | 0..65,535 | ‘\u0000 ‘ | Символы, 2 байта, все больше 0 |
| Object | 4 | | Любая ссылка или null. | null | Хранит ссылки на объекты типа Object или классов наследников |
|  | | | | | |
| **Операторы** | | **Результат** | | | |
| **Арифметические и присваивания** | | | | | |
| **%** | | Остаток от деления по модулю (Modulus) | | | |
| **++** | | Инкремент (Increment) | | | |
| **+=** | | Присваивание со сложением (Addition assignment) | | | |
| **– =** | | Присваивание с вычитанием (Subtraction assignment) | | | |
| **\* =** | | Присваивание с умножением (Multiplication assignment) | | | |
| **/ =** | | Присваивание с делением (Division assignment) | | | |
| **% =** | | Присваивание с модулем (Modulus assignment) | | | |
| **- -** | | Декремент (Decrement) | | | |
| <<= | | Оператор присваивания «Сдвиг влево» | | | |
| &= | | Оператор присваивания побитового «И» («AND») | | | |
| ^= | | Оператор присваивания побитового исключающего «ИЛИ» («XOR») | | | |
| |= | | Оператор присваивания побитового «ИЛИ» («OR») | | | |
| **Операторы сравнения** | | | | | |
| **==** | | Проверяет, равны или нет значения двух операндов, если да, то условие становится истинным | | | |
| **!=** | | Проверяет, равны или нет значения двух операндов, если значения не равны, то условие становится истинным | | | |
| **>** | | Проверяет, является ли значение левого операнда больше, чем значение правого операнда, если да, то условие становится истинным | | | |
| **<** | | Проверяет, является ли значение левого операнда меньше, чем значение правого операнда, если да, то условие становится истинным | | | |
| **>=** | | Проверяет, является ли значение левого операнда больше или равно значению правого операнда, если да, то условие становится истинным | | | |
| **<=** | | Проверяет, если значение левого операнда меньше или равно значению правого операнда, если да, то условие становится истинным | | | |
| **Логические операторы** | | | | | |
| **&&** | | Называется логический оператор «И». Если оба операнда являются не равны нулю, то условие становится истинным | | | |
| **||** | | Называется логический оператор «ИЛИ». Если любой из двух операндов не равен нулю, то условие становится истинным | | | |
| **!** | | Называется логический оператор «НЕ». Использование меняет логическое состояние своего операнда. Если условие имеет значение true, то оператор логического «НЕ» будет делать false | | | |

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| & (побитовое и) | Бинарный оператор AND копирует бит в результат, если он существует в обоих операндах. | (A & B) даст 12, который является 0000 1100 |
| | (побитовое или) | Бинарный оператор OR копирует бит, если он существует в любом из операндов. | (A | B) даст 61 который равен 0011 1101 |
| ^ (побитовое логическое или) | Бинарный оператор XOR копирует бит, если он установлен в одном операнде, но не в обоих. | (A ^ B) даст 49, которая является 0011 0001 |
| ~ (побитовое дополнение) | Бинарный оператор дополнения и имеет эффект «отражения» бит. | (~ A) даст -61, которая является формой дополнением 1100 0011 в двоичной записи |
| << (сдвиг влево) | Бинарный оператор сдвига влево. Значение левых операндов перемещается влево на количество бит, заданных правым операндом. | A << 2 даст 240, который 1111 0000 |
| >> (сдвиг вправо) | Бинарный оператор сдвига вправо. Значение правых операндов перемещается вправо на количество бит, заданных левых операндом. | A >> 2 даст 15, который является 1111 |
| >>> (нулевой сдвиг вправо) | Нулевой оператор сдвига вправо. Значение левых операндов перемещается вправо на количество бит, заданных правым операндом, а сдвинутые значения заполняются нулями. | A >>> 2 даст 15, который является 0000 1111 |

### **Тернарный оператор или условный оператор (?:)**

переменная x = (выражение) ? значение if true : значение if false

**4. Понятие класса и объекта**

Java является объектно-ориентированным языком, поэтому такие понятия как "класс" и "объект" играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, туловища и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Класс определяется с помощью ключевого слова сlass.

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

Для хранения состояния объекта в классе применяются поля или переменные класса. Для определения поведения объекта в классе применяются методы.

Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person

**5. Члены класса**

Класс может содержать члены (members) трех основных категорий.

·      Поля (fields) – переменные, относящиеся к классу и его объектам и в совокупности определяющие состояние класса или конкретного объекта.

·      Методы (methods) – именованные фрагменты исполняемого кода класса, обусловливающие особенности поведения объектов класса.

·      Вложенные классы и вложенные интерфейсы – объявления классов или интерфейсов, размещенные в тексте объявлений других классов или интерфейсов.

**6. Ограничение доступа к членам класса.** модификаторы доступа

* private: члены класса доступны только внутри класса;
* default (package-private) (модификатор, по-умолчанию): члены класса видны внутри пакета (если класс будет так объявлен он будет доступен только внутри пакета);
* protected: члены класса доступны внутри пакета и в наследниках;
* public: члены класс доступны всем;

Последовательность модификаторов по убыванию уровня закрытости: private, default ,protected, public).

Во время наследования возможно изменения модификаторов доступа в сторону большей видимости

**7. Конструкторы**

Конструктор - это специальный метод, который вызывается при создании нового объекта. Не всегда удобно инициализировать все переменные класса при создании его экземпляра. Иногда проще, чтобы какие-то значения были бы созданы по умолчанию при создании объекта. По сути конструктор нужен для автоматической инициализации переменных.

Конструктор инициализирует объект непосредственно во время создания. Имя конструктора совпадает с именем класса, включая регистр, а по синтаксису конструктор похож на метод без возвращаемого значения.

**private int Cat()**; // так выглядит метод по имени Cat

**Cat();** // так выглядит конструктор класса Cat

В отличие от метода, конструктор никогда ничего не возвращает.

Конструктор определяет действия, выполняемые при создании объекта класса, и является важной частью класса. Как правило, программисты стараются явно указать конструктор. Если явного конструктора нет, то Java автоматически создаст его для использования по умолчанию.

**8. Перегрузка методов**

В программе мы можем использовать методы с одним и тем же именем, но с разными типами и/или количеством параметров. Такой механизм называется **перегрузкой методов** (method overloading).

Перегрузка методов — один из способов поддержки полиморфизма в Java. 

**9. Статические члены класса**

Кроме обычных методов и полей класс может иметь статические поля, методы, константы и инициализаторы. Например, главный класс программы имеет метод main, который является статическим. Для объявления статических переменных, констант, методов и инициализаторов перед их объявлением указывается ключевое слово static.

**Статические поля**

При создании объектов класса для каждого объекта создается своя копия нестатических обычных полей. А статические поля являются общими для всего класса. Потому они могут использоваться без создания объектов класса.

Например, переменная counter статическая, то мы можем обратиться к ней в программе по имени класса:

|  |  |
| --- | --- |
| 1  2 | System.out.println(Person.counter); // получаем значение  Person.counter = 8;                 // изменяем значение |

**Статические константы**

Также статическими бывают константы, которые являются общими для всего класса.

**Статические методы**

Статические методы также относятся ко всему классу в целом. Например, в примере выше статическая переменная counter была доступна извне, и мы могли изменить ее значение вне класса Person.

При использовании статических методов надо учитывать ограничения: в статических методах мы можем вызывать только другие статические методы и использовать только статические переменные.

*Другая ссылка*

Иногда требуется определить член класса, который будет использоваться независимо от любого объекта этого класса. И его можно использовать самостоятельно без ссылки на конкретный экземпляр. Для создания подобного члена класса, нужно в начало его объявления перед объявлением типа поместить ключевое слово static. В этом случае он становится доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть и переменные и методы.

По существу переменные экземпляров, объявленные как static, являются глобальными переменными. При объявлении объектов их класса программа не создаёт никаких копий статической переменной. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

У статических методов есть ряд ограничений:

* они могут вызывать только другие статические методы
* они могут непосредственно осуществлять доступ только к статическим переменным
* они не могут ссылаться на члены типа **this** или **super**

В большинстве случаев среда разработки или компилятор заметят ошибку программиста и выведут предупреждение о неправильном использовании переменных.

Например, нельзя писать такой код.

public class HelloApp

{

int number = 5; // ошибка -- следует добавить ключевое слово static

public static void main(String[] args)

{

System.out.println(number); // не скомпилируется

}

}

В статическом методе **main()** мы пытаемся задействовать нестатическую переменную **number**.

Также следите, чтобы не объявлять статическую переменную в статическом методе - так тоже нельзя.

Если для инициализации статических переменных требуется произвести вычисления, то для этой цели можно объявить статический блок, который будет выполняться только один раз при первой загрузки класса.

**10. Константные члены класса**

Суть модификатора final - сделать дальнейшее изменение объекта невозможным.   
Вы можете применять этот модификатор тремя способами: для класса, для поля (переменной) и для метода.

**Final для полей**

Если вы хотите, чтобы **после инициализации** никто **не мог бы изменить** вашу переменную, напишите слово "final":

##### final int INTRVALS\_IN\_BRACES = 2; public static int PI = 3.14; Переменные с final - это константы. При этом их принято писать **заглавными буквами**

##### **Final для методов**

К методам тоже можно применить модификатор final:

Это будет значить, что при наследовании данный **метод нельзя переопределить**

**Final для классов**

Модификатор final может применяться к классам тоже. Это будет означать, что **нельзя создать наследников этого класса**

Из-за того, что класс объявляется final, можно сказать, что все его **методы тоже становятся final - их нельзя переопределить.**

Следует также отметить, что к абстрактным классам (с ключевым словом abstarct), нельзя применить модификатор final, т.к. это взаимоисключающие понятия.

**11. Перечисления**Кроме отдельных примитивных типов данных и классов в Java есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант. Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую

enum Day{

    MONDAY,

    TUESDAY,

    WEDNESDAY,

    THURSDAY,

    FRIDAY,

    SATURDAY,

    SUNDAY

}

Перечисление фактически представляет новый тип, поэтому мы можем определить переменную данного типа и использовать ее:  
public class Program{

    public static void main(String[] args) {

        Day current = Day.MONDAY;

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

    }

}  
***Методы перечислений***

Каждое перечисление имеет статический метод values(). Он возвращает массив всех констант перечисления:

Метод ordinal() возвращает порядковый номер определенной константы (нумерация начинается с 0)  
Конструкторы, поля и методы перечисления

Перечисления, как и обычные классы, могут определять конструкторы, поля и методы.

**12. Наследование**

***Наследование*** (англ. inheritance) - это механизм, позволяющий создавать классы на основе другого класса. Один из 4ех принципов ООП.  
Так, один класс может "наследовать" характеристики другого - его методы и переменные.

* Класс, являющийся основой, называют: **базовым, супер, родительским**.
* Класс, который создают, называют: **потомок, наследник или производный класс**.

Для того, чтобы унаследовать класс, нужно использовать ключевое слово **extends  
  
Правило 1. Наследуем только один класс.**

Java не поддерживает наследование нескольких классов. Один класс - один родитель.

**Обратите внимание**-нельзя наследовать самого себя!

**Правило 2. Наследуется все кроме приватных переменных и методов.**

Выше мы говорили, что класс-наследник будет иметь доступ ко всем переменным и методам родителя. Это не совсем так.

На самом деле, все методы и переменные, помеченные модификатором **private**, не доступны классу-наследнику.

**Правило 3. Переделать метод класса-родителя.**

Представим, что мы наследуем класс, но нам нравится не все, что мы унаследовали. Допустим мы хотим, чтобы определенный--- метод работал не так, как в родителе.

Для того, чтобы переопределить метод класса-родителя, пишем над ним **@Override**

**Правило 4. Вызываем методы родителя через ключевое слово super.**

Представим, что Вы хотите изменить метод родительского класса совсем чуть-чуть - буквально дописать пару строк. Тогда в своем методе мы можем вызвать родительский метод с помощью ключевого слова **super**.

**13. Полиморфизм и позднее связывание  
  
Полиморфизм**, если перевести, - это значит **"много форм"**часто этот принцип описывают фразой: Один интерфейс – много методов.  
  
Свойство, которое позволяет использовать одно и то же имя для решения 2ух или более схожих, но технически разных задач.

**Проявления**:  
Перегрузка методов

Переопределение методов родителя

Итак, фундаментальное различие между статическим и динамическим связыванием в Java состоит в том, что **первое происходит рано, во время компиляции на основе типа ссылочной переменной, а второе – позднее, во время выполнения, с использованием конкретных объектов.**

**14. Абстрактные члены класса**Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово **abstract**

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

Human h = new Human();

Кроме обычных методов абстрактный класс может содержать **абстрактные методы**. Такие методы определяются с помощью ключевого слова abstract и не имеют никакого функционала (public abstract void display();)  
  
Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.  
  
Зачем нужны абстрактные классы? Допустим, мы делаем программу для обсулживания банковских операций и определяем в ней три класса: Person, который описывает человека, Employee, который описывает банковского служащего, и класс Client, который представляет клиента банка. Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы. И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

**15. Интерфейсы**Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.  
Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ**public**, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.  
interface Printable{

    void print();

}  
Чтобы класс применил интерфейс, надо использовать ключевое слово **implements.**Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут эти методы.  
  
Начиная с JDK 8 в интерфейсах доступны статические методы - они аналогичны методам класса. Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод: (Printable.read();)  
Ранее до JDK 8 при реализации интерфейса мы должны были обязательно реализовать все его методы в классе. А сам интерфейс мог содерать только определения методов без конкретной реализации. В JDK 8 была добавлена такая функциональность как **методы по умолчанию**. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

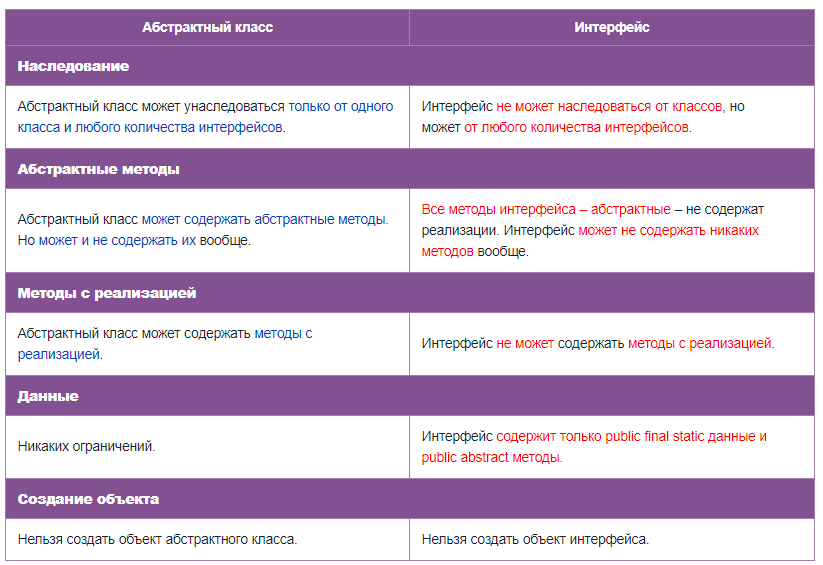
interface Printable {

**default** void print(){

        System.out.println("Undefined printable");

    }

}  
  
Кроме методов в интерфейсах могут быть определены статические константы. Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.  
  
Интерфейсы, как и классы, могут наследоваться.

****

Структуры данных : определение и операции

***Массив***  
Массив — это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется с помощью целого индекса. Например, если а — массив целых чисел, то значение выражения а [ i ] равно i-му целому числу в массиве.  
Объявление:  
**int**[] a;  
Однако этот оператор лишь объявляет переменную а, не инициализируя ее настоящим массивом. Чтобы создать массив, нужно применить оператор new.

**int**[] a = **new** int [100];

Этот оператор создает массив, состоящий из 100 целых чисел. Элементы этого массива нумеруются от 0 до 99 (а не от 1 до 100). После создания массив можно заполнять, например, с помощью цикла.

**int**[] а = **new** int[100];

**for** (**int** i = 0; i < 100; i++)

a[i] = i; //Заполняет массив числами от 0 до 99

Чтобы подсчитать количество элементов в массиве, используйте метод имя Массива .length  
Один массив можно скопировать в другой, но при этом обе переменные будут ссылаться на один и тот же массив.

**int**[] luckyNumbers = smallPrimes;

luckyNumbers[5] = 12;

Если необходимо скопировать все элементы одного массива в другой, следует использовать метод arraycopy из класса System. Его вызов выглядит следующим образом:

System.arraycopy(from, fromlndex, to, tolndex, count);  
  
Для создания многомерных массивов используются дополнительные скобки:

int[][] a = {

{ 1, 2, 3 },

{ 4, 5, 6 }

}

Также массив может создаваться ключевым словом **new**:

// трехмерный массив фиксированной длины

int[][][] b = new int[2][4][4];

* **copyOf()** − предназначен для копирования массива
* **copyOfRange()** − копирует часть массива
* **toString()** − позволяет получить все элементы в виде одной строки
* **sort()** — сортирует массив методом quick sort
* **binarySearch()** − ищет элемент методом бинарного поиска
* **fill()** − заполняет массив переданным значением (удобно использовать, если нам необходимо значение по умолчанию для массива)
* **equals()** − проверяет на идентичность массивы
* **deepEquals()** − проверяет на идентичность массивы массивов
* **asList()** − возвращает массив как коллекцию

***Вектор (динамический массив)***Vector — это способный увеличивать число своих элементов массив ссылок на объекты. Внутри себя Vector реализует стратегию динамического расширения, позволяющую минимизировать неиспользуемую память и количество операций по выделению памяти.

java.util   
**public class** Vector **extends** AbstractList **implements** List, Cloneable, Serializable

|  |
| --- |
| Конструкторы |
| **Vector()** – Создаёт пустой вектор размером 10 и с capacityIncrement = 0. |
| **Vector(Collection c)** – Создаёт вектор содержащий элементы определённой коллекции. |
| **Vector(int initialCapacity)** – Создает пустой вектор с заданным объемом памяти. |
| **Vector(int initialCapacity, int capacityIncrement)** – Создает пустой вектор с заданным объемом памяти (initialCapacity) и увеличением объема (capacityIncrement). |

|  |  |
| --- | --- |
| Методы | |
| void | **add(int index, Object element)**Добавляет в определённую позицию вектора определённый элемент. |
| boolean | **add(Object o)**Добавляет в конец вектора определённый элемент. |
| boolean | **addAll(Collection c)**Добавляет в вектор все элементы определённой коллекции. |
| boolean | **addAll(int index, Collection c)**Добавляет в вектор в определённую позицию все элементы коллекции. |
| int | **capacity()**Возвращает объем вектора. |
| void | **clear()**Удаляет все элементы вектора. |
| Object | **clone()**Возвращает аналог вектора. |
| boolean | **contains(Object elem)**Возвращает значение true если компонент содержится в векторе. |
| void | **copyInto(Object[] anArray)**Копирует элементы вектора в заданный массив. |
| Object | **elementAt(int index)**Возвращает компонент по индексу. |
| boolean | **equals(Object o)**Сравнивает определённый объект с вектором. |
| Object | **get(int index)**Возвращает элемент вектора из определённой позиции. |
| boolean | **isEmpty()** Возвращает true если вектор пуст. |
| Object | **remove(int index)**Удаляет объект из определённой позиции в векторе. |
| Object | **set(int index, Object element)**Заменяет объект в определянной позиции на другой. |
| void | **setSize(int newSize)**Устанавливает размер вектора. |
| int | **size()**Возвращает количество компонентов в векторе. |
| Object[] | **toArray()**Возвращает массив содержащий все элементы вектора. |
| String | **toString()**Возвращает представление вектора в виде строки. |

Пример использования:

Vector objects = new Vector();

objects.addElement(new Button("Hello"));

objects.addElement(new Panel());

***Список***

Для создания простых списков применяется интерфейс List, который расширяет функцональность интерфейса Collection.  
java.util.ArrayList - самый популярный вид списков  
ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Кроме того, вы без проблем можете вставить новый элемент в середину коллекции. А также спокойно удалить элемент из любого места. Элементы ArrayList могут быть абсолютно любых типов в том числе и null. Это удобно, когда вы не знаете точного размера массива. Для сравнения - гостиница для котов имеет фиксированное число номеров, массив использовать можно. Вы владелец преуспевающей компании и число наёмных работников постоянно увеличивается, обычный массив создавать для учёта сотрудников нецелесообразно. В этом случае удобнее работать со списочным массивом.  
  
Работать с ArrayList просто: создайте нужный объект, вставьте объект методом add(), обращайтесь к нему методом get(), используйте индексирование так же, как для массивов, но без квадратных скобок. ArrayList также содержит метод size(), который возвращает текущее количество элементов в массиве (напомню, что в обычном массиве используется свойство length).

Переменные принято называть во множественном числе.

Хотя мы можем свободно добавлять в объект ArrayList дополнительные объекты, в отличие от массива, однако в реальности ArrayList использует для хранения объектов опять же массив. По умолчанию данный массив предназначен для 10 объектов. Если в процессе программы добавляется гораздо больше, то создается новый массив, который может вместить в себя все количество. Подобные перераспределения памяти уменьшают производительность. Поэтому если мы точно знаем, что у нас список не будет содержать больше определенного количества элементов, например, 25, то мы можем сразу же явным образом установить это количество, либо в конструкторе:  ArrayList<String> people = new ArrayList<String>(25);, либо с помощью метода ensureCapacity: people.ensureCapacity(25);

***Очередь***

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

* В идеале, мы не можем встать в середину очереди - **только в конец**. А тот, кто приходит первым, первый получает мороженное  Или справку. Или что там еще. Точно так же работает и **Queue**. Мы не можем ничего вставить в середину очереди - только в конец.
* Тем не менее, убирать элемент можно и **из середины очереди** - так, например, человек в любой момент может махнуть рукой и сказать: "А, я уже три часа тут стою - пойду домой!"
* Более того, элемент **не всегда может влезть в очередь** (в смысле в **Queue**). Например, не сталкивались с  тем, что доходит очередь до Вас - и как раз начинается обед?  Или касса закрывается?

## **Какие есть виды очередей (Queue)**

Для начала Вам хватит знания одного типа - **PriorityQueue**.

Что интересного в **PriorityQueue**? Дело в том, что элементы в ней располагаются не только по порядку, но и **по приоритету**. Этот приоритет можно задавать самостоятельно, но для начинающих это может быть пока немного сложная тема. Пока Вам стоит знать, что если создавать очередь из стандартных типов данных (**Integer**, **Float**, **String**), у нее будет стандартный приоритет:

* для чисел по возрастанию
* для строк по алфавиту

## **FIFO и LIFO в Java**

Кстати, "принцип очереди" - то, что первый, кто пришел, первым получает мороженое - обозначается английской аббревиатурой **FIFO** (**"First in first out"**). Переводится как "первый пришел первый вышел". Кроме очереди можно привести пример с конвейером - то, что ты положил раньше на конвейерную ленту, первым станет готовым продуктом и поедет в магазин.

Также в Java есть структуры, которые работают чуть-чуть по-другому. В таких структурах тот элемент, который поступил первым, первым вынимается. Можно провести аналогию со стопкой тарелок - та тарелка, которую ты поставил первой, оказывается в самом низу. Очень неудобно доставать, правда?

## Поэтому мы моем тарелки, начиная с самой верхней - т.е. той, которую мы поставили последней. Этот принцип называется **LIFO** ("**Last in first out**", "последний пришел первый вышел"). По такому принципу работает еще одна структура данных, очень похожая на очередь - стэк. Тот что overflow

## **Операции с Queue**

1. **add()** - добавляет элемент в конец очереди.

* Поправка: если очередь с приоритетом - т.е. **PriorityQueue**- элемент ставится не обязательно в конец, а в соответствии со своим приоритетом

2. **remove()** и **poll()** - удаляет верхний элемент из очереди.

3. **offer()** - пытается вставить элемент в конец очереди.

4. **peek()** и **element()** - показывают верхний элемент очереди

**У метода remove() есть две формы - remove() и remove(Object o), а у poll() - только одна. В первой форме оба метода одинаковые - они убирают "голову" (первый элемент) очереди.**Но **remove(Object o)**позволяет убирать любой элемент, не только сверху.

Если мы попытаемся вызвать метод **remove()**на пустом листе, получим **Exception**. Если **poll()** - ничего не произойдет.   
Вывод:

* хотите **Exception** (т.е. если пустая очередь - недопустимое состояние)- берите **remove()**
* Если нет - берите **poll()**

**offer()** - пытается вставить в конец очереди. С английского "offer" переводится как "предложить". Почему "предлагаем"? Потому что в очередях с фиксированным размером у нас может не получится вставить элемент в очередь.  
**peek()** (от англ. "подсматривать"), и **element()** - показывают верхний элемент очереди.

Множество

**множество** (**Set**) - один из способов хранения данных в Java. Но особенность множества в том, что оно может хранить только уникальные значения. Таким образом, множество **обеспечивает** **уникальность своих элементов**.  
  
есть три основные виды множеств - **HashSet**, **LinkedHashSet** и **TreeSet**. Есть и другие - но пока мы только знакомимся с **Set**-ми, пока нам знать нам их совсем не обязательно.  
Среди **HashSet**, **LinkedHashSet** и **TreeSet**чаще всего используется **HashSet**.

## **Чем отличаются HashSet, LinkedHashSet и TreeSet**

* **HashSet**хранит элементы в произвольном порядке, но зато быстро ищет. Подходит, если порядок Вам не важен, но важна скорость. Более того, для оптимизации поиска, **HashSet**будет хранить элементы так, как ему удобно.
* **LinkedHashSet**будет хранить элементы в порядке добавления, но зато работает медленнее.
* **TreeSet** хранит элементы отсортированными.

**Операции с множествами**

1. **add()** - добавляет элемент в множество

2. **remove()** - удаляет элемент из множества

3. **contains()** - определяет, есть ли элемент в множестве

4. **size()** - возвращает размер множества

5. **clear()** - удаляет все элементы из коллекции

6. **isEmpty()** - возвращает true если множество пустое, и false если там есть хотя бы 1 элемент

 Элементы хранятся в произвольном порядке, не давайте себя обмануть. Всегда в порядке добавления будут храниться только элементы **LinkedHashSet**, а отсортировано - **TreeSet**.

***Хэш–таблица (отображение, карта)***

**Хеш-таблица (hash table)** — это специальная структура данных для хранения пар ключей и их значений. По сути это ассоциативный массив, в котором ключ представлен в виде хеш-функции.

В контейнерах **Map** (отображение) хранятся два объекта: ключ и связанное с ним значение. Иногда используют термин "ассоциативный массив" или "словарь".

**Map** позволяет искать объекты по ключу. Объект, ассоциированный с ключом, называется значением. И ключи, и значения являются объектами. Ключи могут быть уникальными, а значения могут дублироваться. Некоторые отображения допускают пустые ключи и пустые значения.

Классы для карт:

* **AbstractMap** - абстрактный класс, реализующий большую часть интерфейса **Map**
* **EnumMap** - расширяет класс **AbstractMap** для использования с ключами перечислимого типа **enum**
* [**HashMap**](http://developer.alexanderklimov.ru/android/java/hashmap.php) - структура данных для хранения связанных вместе пар "ключ-значение", применяется для использования хеш-таблицы
* **TreeMap** - для использования дерева, т.е. отображение с отсортированными ключами
* **WeakHashMap** - для использования хеш-таблицы со слабыми ключами, отображение со значениями, которые могут удаляться сборщиком мусора, если они больше не используются
* [**LinkedHashMap**](http://developer.alexanderklimov.ru/android/java/linkedhashmap.php)**-** отображение с запоминанием порядка, в котором добавлялись элементы, разрешает перебор в порядке вставки
* **IdentityHashMap** - использует проверку ссылочной эквивалентности при сравнении документов, отображение с ключами, сравниваемыми с помощью операции == вместо метода **equals()**

Метод **toString()** выводит содержимое в виде фигурных скобок, где ключи и значения разделяются знаком равенства. Ключи слева, значения справа.

Отображения не поддерживают реализацию интерфейса **Iterable**, поэтому нельзя перебрать карту через цикл **for** в форме **for-each**.

Интерфейс **Map** соотносит уникальные ключи со значениями. Ключ - это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект отображения. После того как это значение сохранено, вы можете получить его по ключу.

interface Map<K, V>

В параметре **K** указывается тип ключей, в **V** - тип хранимых значений.

Методы:

* **void clear()** - удаляет все пары "ключ-значение" из вызывающего отображения
* **boolean containsKey(Object k)** - возвращает значение *true*, если вызывающее отображение содержит ключ *k*. В противном случае возвращает *false*
* **boolean containsValue(Object v)** - возвращает значение *true*, если вызывающее отображение содержит значение *v*. В противном случае возвращает *false*
* **Set<Map.Entry<K, V>> entrySet()** - возвращает набор, содержащий все значения отображения. Набор содержит объекты интерфейса **Map.Entry**. Т.е. метод представляет отображение в виде набора
* **boolean equals(Object o)** - возвращает значение *true*, если параметр *o* - это отображение, содержащее одинаковые значения. В противном случае возвращает *false*
* **V get(Object k)** - возвращает значение, ассоциированное с ключом *k*. Возвращает значение *null*, если ключ не найден.
* **int hashCode()** - возвращает хеш-код вызывающего отображения
* **boolean isEmpty()** - возвращает значение *true*, если вызывающее отображение пусто. В противном случае возвращает *false*
* **Set<K> keySet()** - возвращает набор, содержащий ключи вызывающего отображения. Метод представляет ключи вызывающего отображения в виде набора
* **V put(K k, V v)** - помещает элемент в вызывающее отображение, переписывая любое предшествующее значение, ассоциированное с ключом. Возвращает *null*, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом.
* **void putAll(Map<? extends K, ? extends V> m)** - помещает все значения из *m* в отображение
* **V remove(Object k**) - удаляет элемент, ключ которого равен *k*
* **int size()** - возвращает количество пар "ключ-значение" в отображении
* **Collection<V> values()** - возвращает коллекцию, содержащую значения отображения.

Основные методы - **get()** и **put()**, чтобы получить или поместить значения в отображение.

Интерфейс **Sortedmap** расширяет интерфейс **Map** и гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Интерфейс **NavigableMap** (Java 7) расширяет интерфейс **Sortedmap** и определяет поведение отображения, поддерживающее извлечение элементов на основе ближайшего соответствия заданному ключу или ключам.

Интерфейс **Map.Entry** позволяет работать с элементом отображения.

**HashMap** обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден. **TreeMap**хранит ключи отсортированными по возрастанию, а **LinkedHashMap** хранит ключи в порядке вставки, но не обеспечивает скорость поиска **HashMap**.

**Map** – это множество пар. То же множество, но не одиноких элементов, а пар: ключ-значение. Единственное ограничение: первый объект в паре, называемый ключом, должен быть уникальным. В Map не может содержаться две пары с одинаковыми ключами.

|  |  |
| --- | --- |
| **Операция** | **Метод** |
| Получить множество всех пар | **entrySet()** |
| Получить множество всех ключей | **keySet()** |
| Получить множество всех значений | **values()** |
| Добавить пару | **put(key, value)** |
| Получить значение по ключу | **get(key)** |
| Проверить наличие «ключа» | **containsKey(key)** |
| Проверить наличие «значения» | **containsValue(value)** |
| Проверить что Map — пустой | **isEmpty()** |
| Очистить Map | **clear()** |
| Удалить элемент по ключу | **remove(key)** |