# Основы программирования на языке Java

# Темы занятий

- ООП. Классы и объекты
- Работа с файлами. Сериализация
- Инкапсуляция, Наследование, Полиморфизм
- Абстрактные классы
- Интерфейсы
- Обобщения (Generics)
- Коллекции
- Сетевое программирование
- Многопоточные приложения
- Создание оконных приложений
- Оконные приложения. Создание графики

# Занятие 1. Темы

- Общие сведения о языке Java
- Базовые типы данных
- Классы-обертки
- Вывод/вывод данных
- Работа со строками. Класс String
- Генерация случайных чисел. Класс Random
- Работа с массивами
- Обработка исключений
- Работа с текстовыми файлами
- Работа с двоичными файлами

# Java

- Java — объектно-ориентированный язык программирования. Первоначально разрабатывался компанией Sun Microsystems, впоследствии поглощенной компанией Oracle.
- В настоящее время Java это целая платформа и экосистема, используемая для решения самых разнообразных задач: от создания десктопных приложений до написания крупных веб-порталов и сервисов.
- Большинство программ для ОС Android пишутся на Java.
- Код на Java сначала транслируется в специальный байт-код, независимый от платформы, а затем этот байт-код выполняется виртуальной машиной JVM (Java Virtual Machine).
- Использование виртуальной машины JVM избавляет разработчиков от необходимости заботиться об особенностях аппаратной части.
- Виртуальная машина JVM также сама заботится о базовой безопасности, управлении памятью и системе исключений.
- Для того чтобы на компьютере могла быть запущена программа написанная на Java на нем должна быть установлена JRE (Java Runtime Environment).
  JRE это минимальная реализация виртуальной машины, и библиотека классов.

# Версии Java

Внутри Java существует несколько основных семейств технологий:

- **Java SE** — Java Standard Edition, основное издание Java, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем.
- **Java EE** — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия.
- **Java ME** — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например, в мобильных телефонах, КПК, встроенных системах;
- **Java Card** — технология предоставляет безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объёмом памяти и возможностями обработки.

# Базовые типы данных Java

Java содержит 8 базовых типов данных

| Имя | Занимает | Содержит |
|---|---|---|
| boolean | | true/false |
| byte | 1 байт | знаковое целое |
| char | 2 байта | символ в кодировке UTF-16 |
| short | 2 байта | знаковое целое |
| int | 4 байта | знаковое целое |
| long | 8 байт | знаковое целое |
| float | 4 байта | число с плавающей точкой |
| double | 8 байт | число с плавающей точкой |

- Тип boolean может принимать только два значения: true (истина) и false (ложь)
- Все операции сравнения двух величин — вещественных и целых переменных или константы с переменной возвращают в качестве результата тип boolean.
- Переменную типа boolean можно передавать в качестве условия в операторы цикла и ветвления. Если значение переменной типа boolean равно false значит условие не выполняется. Иначе выполняется.

# Классы-обертки

- У каждого базового типа данных есть также класс-обертка. Объект такого класса, кроме данных хранит также разные полезные методы и константы, например минимальное значение типа int можно узнать использовав константу Integer.MIN_VALUE.
- Класс-обертка позволяет передавать параметры в метод по ссылке, а не по значению.
- Объект класса-обертки занимает больше места в памяти и работает медленнее чем соответствующий ему базовый тип. Поэтому использовать классы-обертки следует только в случае необходимости.
- Получить базовый тип из объекта-обертки можно методом <имя базового типа>Value.
- Оборачивание базового типа в класс-обертку называется упаковкой (boxing), а обратный процесс распаковкой (unboxing).

| Тип | Класс-обертка |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |
| boolean | Boolean |

# Вывод данных

- Для операций вывода на консоль в Java используются методы объекта **out**, который находится в классе **System**
- Вывод данных на консоль осуществляется с помощью метода **print()**, которому в качестве параметра в двойных кавычках передают текст, который должен быть выведен на экран, или методом **printf()**, который похож на форматный вывод в C.
- В методе **print()** можно соединять текст и переменные с помощью оператора **+**
- Метод **println()** выводит текст на экран и переводит курсор на новую строку.

```
public class CA_JavaTestNB11
{
   public static void main(String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

```
public class CA_JavaTestNB11
{
   public static void main(String[] args)
   {
      int a, b, c;
      a = 5; b = 3;
      c = a + b;
      System.out.println(a + " + " + b + " = " + c);
   }
}
```

# Ввод данных

- Для получения данных с консоли в классе **System** определен объект **in**. Однако работать напрямую с **System.in** неудобно, поэтому обычно используют класс **Scanner**, который обращается к **System.in**.
- Класс **Scanner** находится в пакете **java.util**, и вначале его нужно подключить (импортировать) с помощью инструкции *import java.util.Scanner*
- В классе **Scanner** определен свой метод, для считывания каждого типа данных:
  - next(): считывает введенную строку до пробела
  - nextLine(): считывает всю введенную строку
  - nextInt(): считывает введенное число int
  - nextDouble(): считывает введенное число double
  - nextBoolean(): считывает значение boolean
  - nextByte(): считывает введенное число byte
  - nextFloat(): считывает введенное число float
  - nextShort(): считывает введенное число short

```
import java.util.Scanner;

public class CA_JavaTestNB11 {
   public static void main(String[] args) {
      int a, b, c;
      Scanner in = new Scanner(System.in);
      a = in.nextInt();
      b = in.nextInt();
      c = a + b;
      System.out.println(a + " + " + b + " = " + c);
   }
}
```

# Класс String

- Для работы с текстовой информацией в Java определен класс **String**, который предоставляет методы для манипуляции строками.
- String представляет собой ссылку на область в памяти, в которой размещены символы.
- Для создания новой строки можно использовать один из конструкторов класса String или напрямую присвоить строке значение текст в двойных кавычках.
- Объект String является неизменяемым (immutable). При любых операциях над строкой, которые изменяют эту строку, создается новая строка.
- Строки можно соединять оператором конкатенации +.
- Если в конкатенации со строкой соединяется не строка, то она будет преобразована к строке

```
import java.util.Scanner;

public class CA_JavaTestNB11
{
  public static void main(String[] args)
  {
      int len;
      String greet, full, name;
      Scanner in = new Scanner(System.in);
      greet = "Hello, ";
      System.out.println("What is your name?");
      name = in.nextLine();
      len = name.length();
      full = greet + name;
      System.out.print(full);
      System.out.print(" your name has “);
      System.out.println(len + " letters");
  }
}
```

# Класс String

- Основные методы класса String для работы над своим содержимым:

| Метод | Действие |
|---|---|
| length() | возвращает длину строки |
| toCharArray() | разбивает строку на массив символов |
| concat() | объединяет строки |
| join() | соединяет строки с учетом разделителя |
| compare() | сравнивает две строки |
| charAt() | возвращает символ строки по индексу |
| equals() | сравнивает строки с учетом регистра |
| indexOf() | находит индекс первого вхождения подстроки в строку |
| startsWith() | определяет, начинается ли строка с подстроки |
| replace() | заменяет в строке одну подстроку на другую |
| trim() | удаляет начальные и конечные пробелы |
| substring() | возвращает подстроку, начиная с определенного индекса |
| toLowerCase() | переводит все символы строки в нижний регистр |

# Генерация случайных чисел

- Генерация случайных чисел осуществляется с помощью класса **Random.**
- Класс Random находится в **java.util.Random** и сначала его нужно импортировать
- Нужно создать объект этого класса и вызвать у него подходящий метод

```
import java.util.Random;

public class CA_JavaTestNB11
{
   public static void main(String[] args) {

       int i, rNum;
       Random rnd = new Random();

       for (i=0; i<10; i++) {
          rNum = rnd.nextInt(90) + 10;
          System.out.print(rNum + "  ");
       }
    }
}
```

| Метод | Возвращаемое значение |
|---|---|
| nextBoolean() | случайное значение типа boolean |
| nextBytes() | заполняет массив byte случайными значениями |
| nextDouble() | случайное значение типа double |
| nextFloat() | случайное значение типа float |
| nextInt(int n) | случайное значение типа int в диапазоне от 0 до n |
| nextInt() | случайное значение типа int |
| nextLong() | случайное значение типа long |

# Массивы

- Массивы в Java можно задать двумя способами: **тип[] название**, либо **тип название[]**
- Размер массива задается с помощью оператора **new**: int[] arr = new int[n];
- Все элементы массива имеют значение по умолчанию, в зависимости от типа.
- Можно инициализировать массив значениями:

  int[] arr = new int[] { 1, 3, 5, 7, 9 };
- Свойство **length** возвращает длину массива, т.е. количество его элементов.
- Двумерный массив указывается с помощью двух пар квадратных скобок, а в операторе new указывается количество строк и столбцов:

  int matrix[][] = new int[3][4]; matrix[1][2] = 3;
- С помощью цикла **for (тип переменная : массив)** можно пройти по всем элементам массива.

```
public class CA_JavaTestNB11
{
 public static void main(String[] args)
{
 int i;
 Scanner in = new Scanner(System.in);
 System.out.println("Size?");
 n = in.nextInt();

 int arr[] = new int[n];
 for (i=0; i<arr.length; i++)
   arr[i] = i*10;

 for (int val : arr)
     System.out.print(val +" ");
 }
}
```

# Обработка исключений

- Ошибки времени выполнения (**Runtime Exceptions**) это ошибки, которые невозможно отследить в момент компиляции, и которые проявляются уже в процессе работы программы (например в случае если пользователь ввел недопустимые данные)
- Для обработки таких ошибок используется конструкция **try-catch**.
- После ключевого слова **try** в фигурных скобках пишется код, который потенциально может вызвать ошибку.
- После блока **try** идет блок **catch**, у которого в круглых скобках указано ошибки какого рода он ловит, а после этого в фигурных скобках, код который будет выполнен если произошла ошибка.
- После одного блока **try** может быть несколько разных блоков **catch**, которые ловят разные ошибки.
- После блока/блоков **catch** может быть также блок **finally**, содержимое которого выполняется вне зависимости от того была ли ошибка в блоке **try**.

# Пример обработки исключений

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

    int age=0;
    Scanner in = new Scanner(System.in);
    System.out.println("What is your age?");

    try {
       age = in.nextInt();
    }
    catch(InputMismatchException e) {
        System.out.println("Please enter your age correctly");
        System.out.println(e.getMessage());
        age = 0;
    }
    finally {
      if (age > 65)
            System.out.println("Hello, pensioner!");
      else
            System.out.println("Hello, worker!");
     }
}
```

# Запись текста в файл. Класс FileWriter

- Для записи текстовой информации в файл можно использовать объект класса **FileWriter**, который находится в **java.io.FileWriter**
- При создании объекта, в качестве параметра передается имя файла включая путь, и необязательный boolean параметр: перезаписывать информацию или добавлять.
- Метод **write()** принимает в качестве параметра строку и записывает её в файл.
- После использования файл нужно закрыть с помощью метода **close()**
- Работа с файлом может вызвать исключение, поэтому она размещается внутри блока try.

```
import java.io.*;

public class CA_JavaTestNB11 {

public static void main(String[] args){
   FileWriter writer;

   try
   {
     writer = new FileWriter("file.txt");
     writer.write("Some text in file\n");
     writer.write("More text in file");
     writer.close();
   }
   catch(Exception ex)
   {
      System.out.println("Cannot open");
   }
}}
```

# Чтение текста из файла.

- Для чтения текста из файла можно использовать либо метод **read()** объекта **FileReader**, но тогда считывание будет происходить посимвольно, либо с помощью методов объекта **BufferedReader**, который при создании принимает в качестве параметра объект класса **FileReader**.
- Метод **readLine()** объекта **BufferedReader** считывает строку из файла.

```
public static void main(String[] args){
   FileReader reader;
   int num;

    try{
       reader = new FileReader("file.txt");
       while ((num = reader.read())!=-1)
           System.out.print((char)num +"  ");
        reader.close();
       }
    catch (Exception ex){
        System.out.println("No file!");
     }}
```

```
public static void main(String[] args){
   BufferedReader buffR;
   String str;
try {
   buffR = new BufferedReader(new FileReader("file.txt"));
   while((str = buffR.readLine())!=null)
      System.out.println(str);
    buffR.close();
     }
catch (Exception ex) {
      System.out.println("No file!");
   }}
```

# Запись двоичной информации в файл

- Для записи двоичной информации в файл используется объект класса **FileOutputStream**.
- При создании объекта, в качестве параметра передается имя файла включая путь, и необязательный boolean параметр: перезаписывать информацию или добавлять.
- Метод **write()** записывает массив байт в файл.
- Два варианта использования метода **write()** :
  - write(массив_для_записи)
  - write(массив, сдвиг, сколько_элементов)
- После использования файл нужно закрыть с помощью метода **close()**

```
import java.io.*;

public class CA_JavaTestNB11 {

public static void main(String[] args){

  byte arr[] = new byte[]{67, 68, 69, 70};
  FileOutputStream fos;
  try
  {
    fos = new FileOutputStream("fil.txt");
    fos.write(arr);
    fos.close();
   }
   catch (Exception ex)
   {
     System.out.println("Cannot open!");
    }
}}
```

# Чтение двоичной информации из файла

- Для чтения двоичной информации из файла используется объект класса **FileInputStream**.
- При создании объекта, в качестве параметра передается имя файла включая путь.
- Варианты использования метода **read()** :
  - read(массив_куда_считывать)
  - read(массив, сдвиг, сколько_элементов)
  - read() – возвращает очередной байт из файла
- После использования файл нужно закрыть с помощью метода **close()**

```
import java.io.*;

public class CA_JavaTestNB11 {

public static void main(String[] args){

  byte arr[] = new byte[8];
  FileInputStream fis;

  try {
    fis = new FileInputStream("fil.txt");
    fis.read(arr);
    fis.close();
    for (byte val: arr)
        System.out.print((char)val +" ");
   }
   catch(Exception ex) {
     System.out.println("Cannot open!");
   }
 }}
```

# Занятие 2. Темы

- Объектно-ориентированное программирование
- Классы и объекты
- Сокрытие данных (инкапсуляция)
- Конструкторы класса
- Инициализаторы
- Переопределение метода toString
- Статические элементы класса
- Обращение объекта к самому себе
- Вложенные классы
- Пакеты

# Процедурно-ориентированные языки

- C, Pascal, FORTRAN и другие сходные с ними языки программирования относятся к категории процедурных языков.
- Каждый оператор процедурного языка является указанием компьютеру совершить некоторое действие, например принять данные от пользователя, произвести с ними определенные действия и вывести результат этих действий на экран.
- Программы, написанные на процедурных языках, представляют собой последовательности инструкций, последовательность функций, которые вызывают друг друга.

**Недостатки процедурного подхода:**

- Процедурный подход не позволяет в достаточной степени упростить сложные программы
- Неконтролируемый доступ к данным. Есть неограниченность доступа функций к глобальным данным.
- Разделение данных и функций плохо отображает картину реального мира.

# Объектно-ориентированное программирование

- Идея ООП - объединить данные и действия, производимые над этими данными, в единое целое.
- Три кита объектно-ориентированного подхода:
  1. Инкапсуляция
  2. Наследование
  3. Полиморфизм
- В ООП вводится новое понятие – **класс**. Класс содержит в себе:
  1. поля (данные)
  2. методы (функции для работы с данными)
- **Класс** является по сути пользовательским типом данных. После объявления класса, можно создавать переменные типа этого класса. Такая созданная переменная называется **объектом** (экземпляром) класса.
- Используя имя объекта, можно обращаться к его **полям** и **методам**.
- Типичная программа на языке Java состоит из совокупности объектов, взаимодействующих между собой посредством вызова методов друг друга.

# Класс и объект

Объявление класса:

```
class firstClass {
  private int info;

   public void setInfo(int n) {
       info = n;
      }
  public void printInfo() {
       System.out.println ("info = " + info);
      }
}
```

- Доступ к полям и методам класса возможен только через конкретный объект этого класса.
- Для того чтобы получить доступ, необходимо использовать операцию точки (.), связывающую метод или поле с именем объекта.
- Для создания объекта из класса необходимо использовать оператор **new**.

Создание объекта из класса и вызов его методов:

```
public static void main(String[] args)
{
    firstClass fc = new firstClass();
    fc.setInfo(23);
    fc.printInfo();
}
```

# Сокрытие данных (инкапсуляция)

- Ключевой особенностью объектно-ориентированного программирования является возможность сокрытия данных.
- Данные и методы заключены внутри класса и защищены от несанкционированного доступа извне.
- Если необходимо защитить какие-либо данные, то их помечают ключевым словом **private**. Такие данные доступны только внутри класса.
- Данные, описанные с ключевым словом **public**, напротив, доступны за пределами класса.

```
class secondClass
{
   private int privInf;
   public int pubInf;
   public void printInfo()
  {
    System.out.println ("info = " + privInf);
  }
};
```

```
public static void main(String[] args)
{
    secondClass sc = new secondClass();
    sc.pubInf =12;
    sc.privInf = 5;       //ошибка! нет доступа
}
```

# Конструкторы класса

- Конструктор — это метод класса, выполняющийся автоматически в момент создания объекта.
- С помощью конструктора удобно инициализировать поля объекта автоматически в момент его создания.
- У конструктора есть несколько особенностей, отличающих его от других методов класса.
- Имя конструктора в точности совпадает с именем класса. Таким образом, компилятор отличает конструкторы от других методов класса.
- У конструктора не существует возвращаемого значения. Конструктор автоматически вызывается системой, и, следовательно, не существует вызывающей программы или функции, которой конструктор мог бы возвратить значение.
- Конструкторы бывают двух видов:
  1. Конструктор по умолчанию (не принимает параметров)
  2. Параметризованный конструктор (со списком параметров)

# Использование конструктора

```
class firstClass {
  private int info;


   public firstClass(){
       info = 0;
   }
   public void setInfo(int n) {
       info = n;
   }
   public void printInfo() {
     System.out.println("info = " + info);
   }
}
```

```
public static void main(String[] args)
{
    firstClass fc = new firstClass();
    fc.printInfo();
}
```

# Инициализаторы

- Кроме конструктора начальную инициализацию объекта можно выполнить с помощью инициализатора объекта.
- Инициализатор выполняется до любого конструктора, поэтому туда можно поместить код, общий для всех конструкторов.
- Инициализатором является блок кода, записанный в фигурных скобках, перед которыми нет никакого названия.

```
class firstClass {
  private int info;
  {
      info = 0;
  }
public void printInfo() {
     System.out.println("info = " + info);
  } }
```

```
public static void main(String[] args)
{
    firstClass fc = new firstClass();
    fc.printInfo();
}
```

# Переопределение метода toString()

- Для того чтобы установить вид вывода на экран объекта нужно переопределить его метод **toString**().
- Метод toString() должен быть публичным и возвращать значение типа String.

```
class Car {
        private int number;
        private string vendor;

       public Car(){
           number = 0;
           vendor = "vaz";
       }


public String toString(){
            String str;
            str = "num: "+number+"\nven:"+vendor;
            return str;
        }
    }
```

```
public static void main(String[] args){
        Car mycar=new Car();
        System.out.println(mycar)
}
```

# Статические элементы класса

- Элементы класса: поля и методы могут быть определены с использованием ключевого слова **static**. В этом случае будет существовать только один такой элемент для всех объектов данного класса.
- Статические поля класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение.
- Обращаться к статическому элементу можно, как через имя класса, так и через имя объекта.
- Инициализаторы также могут быть статическими, для инициализации статических переменных

```
class Car{
   static int total = 0;

   public Car(){
      total++;
    }


public static int getTotal(){
    return total;
   }
}
```

```
class Car{
   static int total;

   public Car()
   {
      total++;
   }
   static
   {
       total = 0;
   }
}
```

```
public static void main(String[] args)
{
  Car c1 = new Car();
  Car c2 = new Car();
  Car c3 = new Car();
  System.out.println("Created " + Car.getTotal() + " cars");
  System.out.println("Created " + c2.getTotal() + " cars");
}
```

# Обращение объекта к самому себе

- С помощью ключевого слова **this** объект может обратиться сам к себе.
- Одно из возможных применений ключевого слова this это разрешение неоднозначности контекста, когда входящий параметр назван так же, как поле данных класса.
- Также с помощью this объект может передать себя в качестве параметра в функцию.

```
class Car {
   private int number;
   private String vendor;

   public Car(int number, String vendor) {
      this.number = number;
      this.vendor = vendor;
   }

   public void show() {
       System.out.println("num: "+number+"\nven:"+vendor);
   }
}
```

# Вложенные классы

- В Java классы могут быть вложенными один в другой, т.е. одни классы, можно определять внутри других классов.
- Объекты внутреннего класса могут быть созданы только внутри внешнего класса.
- Внутренний класс имеет доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private. Внешний класс тоже имеет доступ ко всем членам внутреннего класса.
- Ссылку на объект внешнего класса из внутреннего класса можно получить с помощью выражения Внешний_класс.this
- Вложенные классы могут быть статическими. Статические вложенные классы позволяют скрыть некоторую комплексную информацию внутри внешнего класса

# Вложенные классы

```
class Car {
    private int number; private String vendor;
    private Motor moto;

 Car(int number, String vendor, int volume, int power) {
  this.number = number; this.vendor = vendor;
  moto = new Motor(volume, power);
    }

 public String toString(){
   String str;
   str = "num: "+number+"\nven:"+vendor+"\n";
   str+= moto.getInfo();
   return str;
  }

 class Motor {
   private int volume; private int power;

   Motor(int volume, int power) {
     this.volume = volume; this.power = power;
    }

   String getInfo() {
      return "volume: "+volume +"\npower: " + power;
       }
}}
```

```
public static void main(String[] args)
{
    Car myCar = new Car(12345, "Volvo", 10, 124);
     System.out.println(myCar);
}
```

# Пакеты

- Часто классы в Java объединяют в пакеты (package). Пакеты позволяют логически сгруппировать классы. Пакеты могут быть вложены друг в друга.
- Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом
- Чтобы указать, что класс принадлежит определенному пакету, надо использовать команду **package**, после которой указывается имя пакета: *package название;*
- Организация классов в пакеты позволяет избежать конфликта имен между классами. В разных пакетах классы могут иметь одинаковые названия. Принадлежность к пакету позволяет гарантировать однозначность имен.
- В случае необходимости использовать классы из других пакетов, надо подключить эти пакеты и классы и помощью команды **import**, после которого указывается класс из пакета, который надо подключить, или * чтобы подключить все классы.
- В Java имеется ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д.

# Объектно-ориентированное программирование в Java

- В Java реализованы все принципы объектно-ориентированного программирования: наследование, инкапсуляция и полиморфизм.
- В большинстве случаев, работа с объектами идет напрямую, а не через указатель, поэтому доступ к полям и методам класса осуществляется с помощью оператора точки (.)
- Модификатор доступа (public, protected, private) указывается отдельно для каждого поля и метода класса.
- Если модификатор доступа не указан, то по умолчанию будет public.
- Все методы в родительском классе сразу считаются виртуальными.

# Занятие 3. Темы

- Сериализация
- Наследование
- Полиморфизм
- Абстрактный класс
- Интерфейсы
- Паттерн Наблюдатель
- Сортировка и интерфейс Comparable

# Сериализация

- **Сериализация** это сохранение данных объекта в поток (файл) в виде линейной последовательности байт.
- Обратная операция – чтение данных из потока в объект, называется **десериализация**.
- Для сериализации объектов в поток используется класс **ObjectOutputStream**, и его метод **writeObject()**, который записывает объект в поток. В конструктор объекта ObjectOutputStream передается поток, в который производится запись.
- Для десериализации служит класс **ObjectInputStream** и его метод **readObject()**. В конструкторе объекта ObjectInputStream передается поток ввода из которого нужно считать данные.
- Для того чтобы объект класса можно было сериализовать нужно пометить класс, как реализующий интерфейс **Serializable** *class someClass implements Serializable*
- Поля, сохранять которые не требуется должны быть объявлены с модификатором **transient**.

# Пример сериализации

**Сериализуемый объект**

```
class Person implements Serializable
{
    private int id;
    private String name;

    public Person(int id, String name)
    {
        this.id = id;
        this.name = name;
    }

    public void show()
    {
        System.out.println("Person “ + id +
        "\nName: “ + name);
    }
}
```

**Сериализация**

```
public static void main(String[] args){
 Person p1 = new Person(1, "Vasya");
 try {
        ObjectOutputStream oos = new
        ObjectOutputStream(new FileOutputStream("person.txt"));
        oos.writeObject(p1);
        oos.close();
    }
 catch(Exception ex){
       System.out.println("Cannot open file");
    }}
```

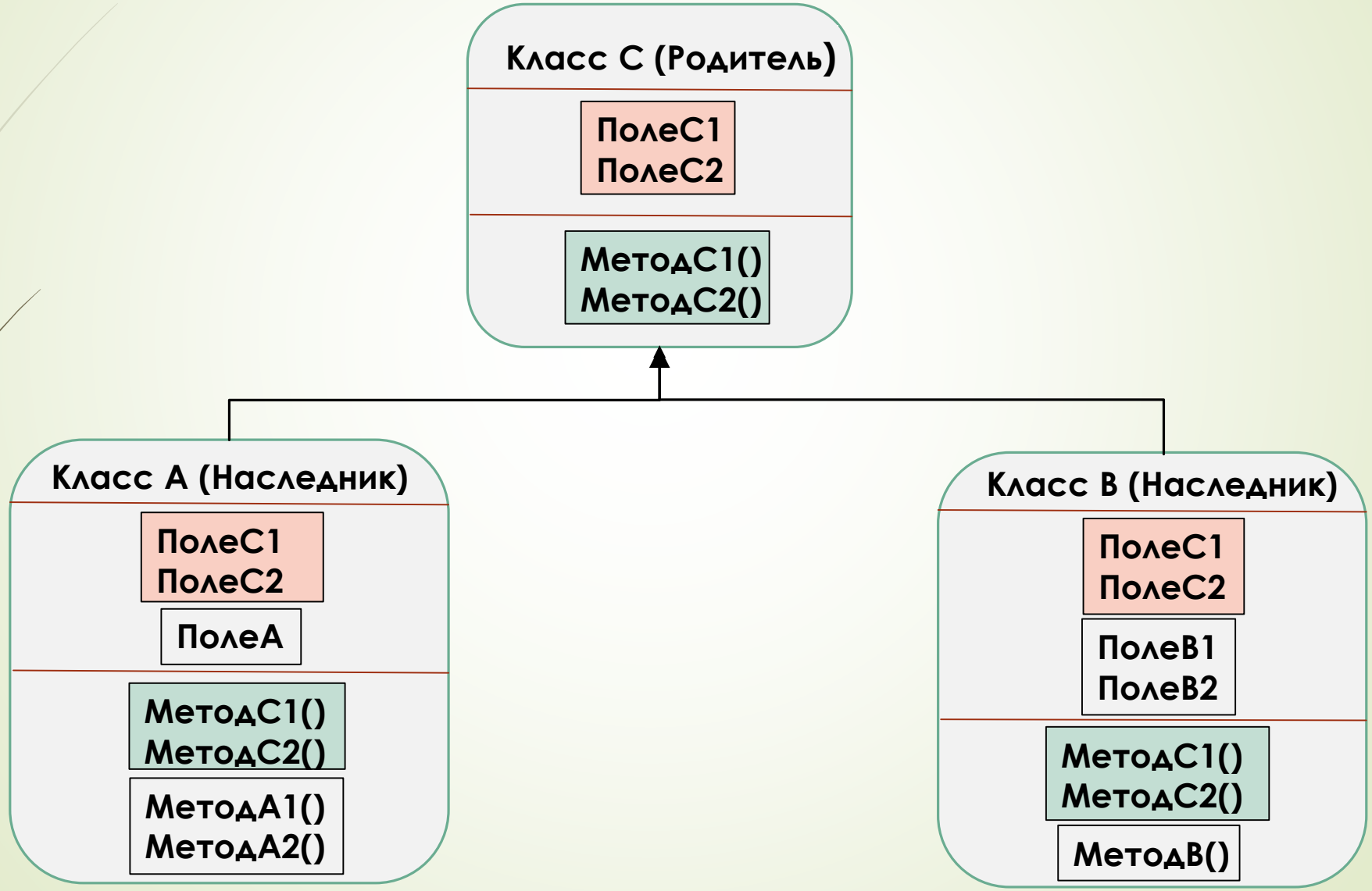
**Десериализация**

```
public static void main(String[] args){
  Person p1;
  try{
        ObjectInputStream ois = new
        ObjectInputStream(new FileInputStream("person.txt"));
        p1=(Person)ois.readObject();
        ois.close();
        p1.show();
     }
 catch(Exception ex) {
    System.out.println("Cannot open file");
  }}
```

# Наследование

- Если есть несколько классов у которых часть полей и методов одинаковая то для того чтобы не дублировать код можно использовать наследование.
- Наследование это создание новых классов, называемых ***наследниками*** или ***производными классами***, из уже существующих или базовых классов. Производный класс получает все возможности базового класса, но может также быть усовершенствован за счет добавления собственных полей и методов.
- Наследование позволяет многократно использовать существующий код. Имея написанный и отлаженный базовый класс, необязательно модифицировать его если нужно внести в код изменения. Вместо этого можно использовать механизм наследования.
- Наследование позволяет использовать классы, созданные кем-то другим, без модификации кода, просто создавая производные классы, подходящие для частной ситуации.
- Отношение наследования объявляется с помощью ключевого слова **extends**, сначала объявляется класс наследник, потом родительский класс:
  **class Наследник extends Родитель**
- Ключевое слово **super** используется чтобы обратиться из класса наследника к родительскому классу.
- Запретить наследовать от класса можно с помощью ключевого слова **final.**

# Наследование



# Пример наследования

**Базовый класс**

```
class Person {
  protected int id;
  protected String name;

  public Person()
  {
     id = 0;
     name = "Anonymus";
  }

  public Person(int id, String name)
  {
      this.id = id;
      this.name = name;
  }

  public void show()
  {
     System.out.println("Person:\n" + "ID:
     "+id+"\nName: "+name);
  }
}
```

**Класс наследник 1**

```
class Student extends Person {
  private int group;

  public Student()
  {
     group = 0;
  }

  public Student(int id, String name,
int group)
  {
     super(id, name);
     this.group = group;
   }

  public void show()
  {
     System.out.println("I'm
     Student:\nID: "+id+"\nName:
     "+name+"\nGroup: "+group);
  }
}
```

**Класс наследник 2**

```
class Teacher extends Person {
  private String rank;

  public Teacher()
  {
     rank = "Prof.";
  }

  public Teacher(int id, String name,
String rank)
  {
     super(id, name);
     this.rank = rank;
  }

   public void show()
  {
     System.out.println("I'm
     Teacher:\nID: "+id+"\nName:
     "+name+"\nRank: "+rank);
   }
}
```

# Полиморфизм

- Полиморфизм это свойство, которое позволяет одно и то же имя использовать для решения нескольких схожих, но технически разных задач.
- Концепцией полиморфизма является идея "один интерфейс, множество методов".
- В объект базового класса фактически можно записать любой объект наследник.
- Можно создать полиморфную функцию, с одинаковым именем в разных классах наследниках. Выбор конкретной полиморфной функции будет зависеть от того какой именно объект наследник записан в объект базового класса. Такой механизм вызова подходящей функции называется динамической диспетчеризацией методов.
- Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.
- Чтобы использовать полиморфизм в Java, необходимо выполнить три условия:
  1. Все классы должны являться наследниками одного и того же базового класса.
  2. Полиморфная функция должна быть объявлена в базовом классе
  3. Полиморфная функция должна быть переопределена в классах наследниках

# Пример полиморфизма

**Базовый класс**

```
class Person {
  protected int id;
  protected String name;

  public Person()
  {
     id = 0;
     name = "Anonymus";
  }

  protected void show()
  {
     System.out.println("Person:\n" +
     "ID: "+id+"\nName: "+name);
  }
}
```

**Класс наследник 1**

```
class Student extends Person {
   int group;

  public Student() {
    group = 0;
  }

  public void show() {
     System.out.println("I'm
    Student:\nID: "+id+"\nName:
   "+name+"\nGroup: "+group);
  }
}
```

**Класс наследник 2**

```
class Teacher extends Person {
private String rank;

public Teacher() {
   rank = "Prof.";
}

public void show() {
   System.out.println("I'm
   Teacher:\nID: "+id+"\nName:
   "+name+"\nRank: "+rank);
}
}
```

```
public static void main(String[] args){
    Person[] pArr = new Person[3];

     pArr[0] = new Teacher();
     pArr[1] = new Student();
     pArr[2] = new Student();

     for (i = 0; i < 3; i++)
         pArr[i].show();
}
```

# Абстрактный класс

- Абстрактным классом называется базовый класс из которого запрещено создавать объекты.
- Абстрактный класс может существовать с единственной целью — быть родительским по отношению к производным классам, объекты которых будут реализованы.
- Чтобы сделать класс абстрактным необходимо перед его объявлением указать ключевое слово **abstact**.
- Если в классе есть хотя бы одна абстрактная функция то класс должен быть объявлен абстрактным. Абстрактная функция содержит только объявление, без реализации.
- После указания класса как абстрактного создание из него объектов становится невозможным.

**Абстрактный базовый класс**

```
abstract class Person
{
   public abstract void show();
}
```

**Класс наследник**

```
class Student extends Person
{
    public void show()
    {
        System.out.println(group);
    }
}
```

**Создание объектов**

```
public static void main(String[] args)
{
  Student st = new Student();
  Person p = new Person(); //ошибка
}
```

# Интерфейсы

- Интерфейсы похожи на абстрактные классы. Они определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.
- Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**: **class Класс implements Интерфейс**.
- Класс не может наследовать от нескольких родительских классов, но может применить множество интерфейсов.
- Если надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова implements:

  **class Класс implements Интерфейс1, Интерфейс2**
- Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом.
- Интерфейсы могут иметь не только определения методов, но и их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Такая реализация метода по умолчанию помечается ключевым словом **default**.
- Если класс применяет интерфейс, то он должен реализовать все методы интерфейса не имеющие реализации по умолчанию. Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный.
- С помощью интерфейсов, так же как с помощью наследования можно реализовать полиморфизм.

# Паттерн наблюдатель



```
//интерфейс
interface IObserver
{
   void getInfo(int i);
}
```

```
//класс уведомляемый
class Observer1 implements IObserver {
  int num;

  Observer1(int num) {
    this.num = num;
  }

  public void getInfo(int i){
    System.out.println("Я 1тип №" +
    num + " получил "+i);
  }
}
```

```
//класс уведомляющий
class Informator {
  int curNum;
  IObserver obs[] = new
  IObserver[10];

  public Informator() {
    curNum = 0;
}

  public void
  addObserver(IObserver o) {
    obs[curNum] = o;
    curNum++;
  }

  public void sendInfo(int info) {
    int i;
    for (i=0; i<curNum; i++)
        obs[i].getInfo(info);
  }
}
```

```
class Observer2 implements IObserver {
  int num;

  Observer2(int num) {
    this.num = num;
  }

  public void getInfo(int i){
    System.out.println("Я 2тип №" +
    num + " получил "+i);
  }
}
```

```
public static void main(String[] args) {
  Observer1 ob1 = new Observer1(1);
  Observer2 ob2 = new Observer2(2);
  Observer1 ob3 = new Observer1(3);

  Informator inf = new Informator();

  inf.addObserver(ob1);
  inf.addObserver(ob2);
  inf.addObserver(ob3);

  inf.sendInfo(10);
}
```

# Сортировка встроенными средствами

- В Java массивы можно сортировать с помощью функции sort находящейся в классе Arrays пакета java.util. Но если для встроенных типов сортировка происходит автоматически, то чтобы такая сортировка стала возможной для массивов пользовательских типов (объектов) необходимы дополнительные действия:
  - Пользовательский класс должен применить интерфейс **Comparable<E>**.
  - В пользовательском класса нужно переопределить метод **compareTo**.
- Метод **compareTo** сравнивает текущий объект с объектом, который передается в качестве параметра. На выходе он возвращает целое число, которое может иметь одно из трех значений:
  - Меньше нуля - текущий объект меньше объекта, который передается в качестве параметра.
  - Равен нулю - оба объекта равны.
  - Больше нуля - текущий объект больше объекта, передаваемого в качестве параметра.

# Пример сортировки и Comparable

```
class Car implements Comparable<Car> {
   private int number;
   private String vendor;

   public Car(int number, String vendor) {
      this.number = number;
      this.vendor = vendor;
    }

   public int compareTo(Car c) {
        if (number < c.number) return -1;
            else return 1;
    }

   public void show() {
      System.out.println("num:
      "+number+"\nven:"+vendor);
    }
 }
```

```
import java.util.*;

public static void main(String[] args) {
   int i;

   Car[] cars = new Car[3];

   cars[0] = new Car(1, "BMW");
   cars[1] = new Car(3, "Lada");
   cars[2] = new Car(2, "Toyota");

   for (i=0; i< cars.length; i++)
       cars[i].show();

    Arrays.sort(cars);

    for (i = 0; i < cars.length; i++)
        cars[i].show();
}
```

# Сортировка по разным полям

- Если массив пользовательских объектов нужно сортировать в разных случаях по разным критериям, тогда можно воспользоваться интерфейсом **Comparator<E>:**
  - Для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий интерфейс Comparator.
  - В этом вспомогательном классе нужно переопределить метод **compare**. сравнивающий два объекта.
  - Объект этого вспомогательного класса нужно передать в стандартный метод сортировки массива в качестве второго параметра.
- Интерфейс Comparator находится в пакете java.util.

# Пример сортировки и Comparator

```
class Car {
   private int number;
   private string vendor;
   public Car(int number, string vendor) {
      this.number = number;
      this.vendor = vendor;
    }
}

 class CompByNum implements Comparator<Car> {
    public int compare(Car c1, Car c2)
     {
          if (c1.number < c2.number) return -1;
            else return 1;
      }
    }

class CompByVen implements Comparator<Car> {
    public int compare(Car c1, Car c2)
    {
        return c1.vendor.compareTo(c2.vendor);
     }
   }
}
```

```
public static void main(String[] args) {
{
   Car[] cars = new Car[3];

   cars[0] = new Car(1, "BMW");
   cars[1] = new Car(3, “Toyota");
   cars[2] = new Car(2, “Lada");

   for (i=0; i< cars.length; i++)
       cars[i].show();

 Arrays.sort(cars, CompByNum());
    for (i = 0; i < cars.length; i++)
        cars[i].show();

Arrays.sort(cars, new CompByVen());
    for (i = 0; i < cars.length; i++)
        cars[i].show();
}
```

# Занятие 4. Темы

- JAR файлы (Java ARchive)
- Обобщения (Generics)
- Коллекции
- Класс ArrayList
- Сетевое программирование

# JAR файлы (Java ARchive)

- Если программа на Java состоит из нескольких классов, то при компиляции каждый класс будет скомпилирован в отдельный файл вида ИмяКласса.class
- Если нужно запустить программу из командной строки, то нужно вызвать на выполнение тот файл (без расширения.class), в котором содержится метод main, например: *java Program*
- Чтобы итоговая программа состояла из одного файла, а не из нескольких её можно экспортировать в JAR-файл (Java ARchive). В IDE Eclipse: правой кнопкой по проекту и выбрать: Export -> Java -> Runnable JAR file. Выбрать Launch configuration и ввести имя файла в поле Export destination
- Запускать из командной строки программу, которая находится в JAR-файле нужно следующей командой: *java -jar ИмяФайла.jar*
- С помощью JAR-файлов можно создавать библиотеки классов Java и подключать их для использования в других проектах. Каждый подключаемый класс должен быть объявлен как public.
- Библиотека классов экспортируется как JAR File и не содержит метода main() поскольку она не запускается напрямую, а только подключается для использования в другом проекте.
- Для использования библиотеки в проекте ее необходимо подключить. В IDE Eclipse: правой кнопкой по проекту и выбрать Properties -> Java Build Path -> Libraries -> Add External JARs... выбрать файл jar.
- Также в программе необходимо импортировать пакет, в котором были созданы классы библиотеки

# Обобщения (Generics)

- Обобщения или generics позволяют уйти от жесткого определения используемых типов. Если есть несколько классов с одинаковой функциональностью, которые различаются только типом полей, то можно вместо нескольких классов написать один обобщенный класс и только в момент создания объекта из этого класса определить тип его полей.
- Для объявления обобщенного класса после его имени в треугольных скобках указывается универсальный параметр, который затем будет замещен на конкретный тип: ***class имяКласса<T>***
- При создании объекта из обобщенного класса в треугольных скобках указывается конкретный тип, который будет подставлен вместо универсального параметра: ***Класс<Тип> объект = new Класс<Тип>(параметры)***
- Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. В этом случае универсальный параметр добавляется в объявление метода после всех модификаторов и перед типом возвращаемого значения:
  ***public <T> void метод(T параметр)***
- При вызове обобщенного метода метода перед его именем в угловых скобках указывается, какой тип будет передаваться на место универсального параметра:
  ***объект.<Тип>метод(параметр)***

# Пример использования обобщений

```
class Car
{
 private int number;
 private String vendor;
 private int maxSpeed;

 Car(int number, String
vendor, int maxSpeed)
{
 this.number = number;
 this.vendor = vendor;
 this.maxSpeed =
 maxSpeed;
}

public String toString()
{
  return "Car N."
  +number + " "+vendor
  +", max speed: " +
  maxSpeed + "\n";
 }
}
```

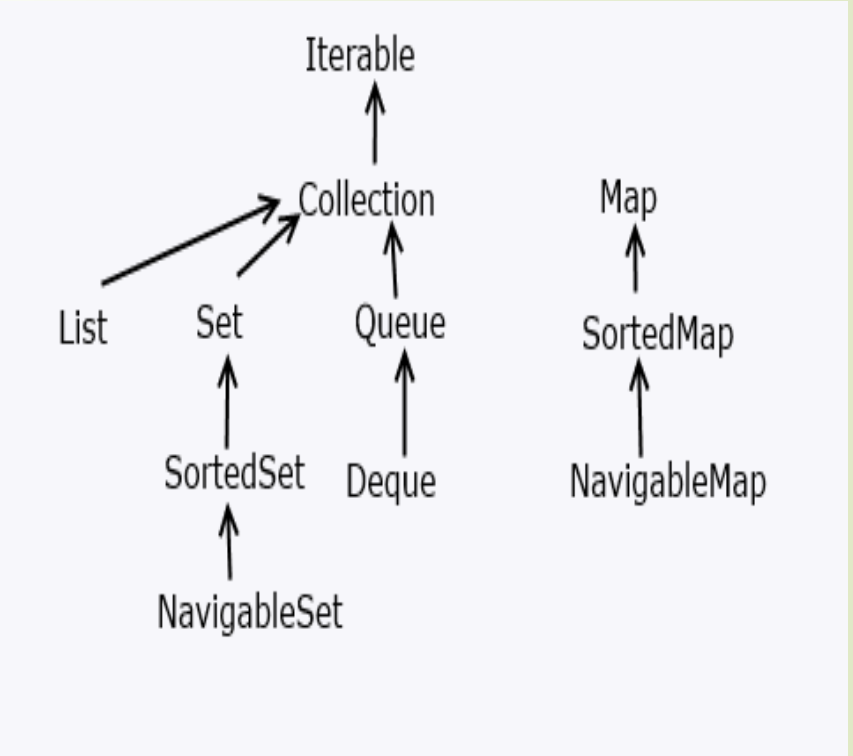
```
class Person
{
private int id;
private String name;

 Person(int id, String name)
 {
   this.id = id;
   this.name = name;
 }

public String toString()
{
  return "Person:\n" + "ID: " +
  id + "\nName: " + name;
 }
}
```

```
class MyList<T>
{
   T arr[];
   int total;

 MyList(T a[], int total)
 {
    arr = a;
    this.total = total;
 }

 void showAll()
 {
  for (int i=0; i<total; i++)
  System.out.println(arr[i]);
 }
}
```

```
public static void main(String[] args)
{
   Car cArr[] = new Car[6];
    cArr[0] = new Car(123, "BMW", 100);
    cArr[1] = new Car(432, "Lada", 80);
    cArr[2] = new Car(654, "Mercedes", 120);
    cArr[3] = new Car(873, "Toyota", 200);
  MyList<Car> carList = new MyList<Car>(cArr, 4);

    carList.showAll();

   Person pArr[] = new Person[3];
   pArr[0] = new Person(1,"Vasya");
   pArr[1] = new Person(2,"Petya");
   pArr[2] = new Person(3,"Kolya");
MyList<Person> peList = new MyList<Person>(pArr,3);

  peList.showAll();
}
```

# Коллекции

- Для хранения наборов данных предназначено такое встроенное средство языка как массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции.
- Коллекции являются классами используемыми для гибкого хранения набора данных. Кроме гибкого размера, они также реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.
- В Java существует множество коллекций, но все они образуют стройную и логичную систему. В основе каждой коллекции лежит применение того или иного интерфейса.
- Интерфейс Collection является базовым для всех коллекций, определяя основной функционал.
- Классы коллекций находятся в пакете java.util



# Основные классы коллекций

| Класс | Описание |
|---|---|
| ArrayList | простой список объектов |
| LinkedList | связанный список объектов |
| ArrayDeque | двунаправленная очередь. Вставка и удаление и с начала и с конца |
| PriorityQueue | очередь приоритетов |
| HashSet | набор элементов, где каждый имеет ключ - уникальный хеш-код |
| HashMap | структура данных в виде словаря, каждый элемент состоит из пары: ключ, значение |
| TreeSet | набор отсортированных в виде дерева объектов |
| TreeMap | структура данных в виде дерева, каждый элемент состоит из пары: ключ, значение |

# Класс ArrayList

- Класс ArrayList представляет собой список объектов, добавлять и удалять элементы из которого можно в любом месте.
- Основные методы класса ArrayList:
  - add(E obj): добавляет в список объект obj
  - get(int index): возвращает объект из списка по индексу index
  - set(int index, E obj): присваивает значение объекта obj элементу, который находится по индексу index
  - remove(int index): удаляет объект из списка по индексу index, возвращая при этом удаленный объект
  - indexOf(Object obj): возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1
  - lastIndexOf(Object obj): возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1
  - listIterator (): возвращает объект ListIterator для обхода элементов списка
  - sort(): сортирует список
  - subList(int start, int end): возвращает набор элементов, которые находятся в списке между индексами start и end (не включая end)

# Пример использования класса ArrayList

```
public static void main(String[] args) {
    ArrayList<String> al = new ArrayList<String>();
    al.add("First");
    al.add("Second");
    al.add("Third");
    al.add("Fourth");
    al.add("Fifth");

    for (String str: al)
        System.out.print(str + " ");
    System.out.println();

    for (String str: al.subList(1, 4))
    System.out.print(str + " ");
    System.out.println();

    al.set(1, "NewInfo");
    al.remove(3);
    for (String str: al)
        System.out.print(str + " ");
    System.out.println();
}
```

# Сетевое программирование

- Для обмена данными между компьютерами существует несколько различных протоколов. Один из них это протокол TCP/IP, в котором для подключения нужно знать адрес и порт компьютера к которому происходит подключение. Адрес представляет собой 32-битную структуру для протокола IPv4, 128-битную для IPv6. Номер порта — целое число в диапазоне от 0 до 65535 (для протокола TCP).
- Пара состоящая из адреса и порта называется **socket** (разъем). Фактически socket обозначает точку, через которую происходит соединение по сети двух программ.
- В процессе обмена, как правило, используется два сокета — сокет отправителя и сокет получателя. В программе можно создать «слушающий» сокет (серверный сокет), который будет находиться в цикле ожидания и просыпаться при появлении нового соединения.
- Обмен данными с помощью сокетов осуществляется по клиент-серверной схеме: сервер находится в режиме ожидания и ждет подключения, клиент зная адрес и порт сервера (сокет) подключается к нему и начинает обмен информацией. Клиент посылает серверу запросы, а тот отправляет на них ответы.

# Сетевое программирование в Java



В Java для обмена данными с использованием механизма сокетов существуют классы **Socket** и **ServerSocket**, которые находятся в пакете java.net.

1. Сервер создает слушающий сокет, создавая новый объект типа ServerSocket и указав порт по которому будет происходить подключение: ***new ServerSocket(порт)***
2. У созданного объекта типа ServerSocket вызывается метод **accept()**, а результат присваивается переменной типа Socket. Сервер останавливается и ждет подключения. Когда клиент подключается информация о нем записывается в переменную типа Socket, с помощью которой с ним происходит обмен данными.
3. У полученного объекта типа Socket вызываются методы **getInputStream()** для получения потока входящих данных и **getOutputStream()** для получения потока исходящих данных.
4. Используя эти два потока сервер сначала считывает приходящие во входной поток данные, а потом отправляет ответ в выходящий поток.

1. Клиент создает объект типа **Socket** и указывает адрес и порт сервера к которому он хочет подключиться: **new Socket(адрес_сервера, порт)**
2. У полученного объекта типа Socket вызываются методы **getInputStream()** для получения потока входящих данных и **getOutputStream()** для получения потока исходящих данных.
3. Используя эти два потока клиент сначала отправляет запрос на сервер, а потом считывает его ответ.
4. После того как обмен данными завершен клиент закрывает подключение

# Сетевое программирование в Java

```
public static void main(String[] args) {          //Сервер
  Socket clientSocket;  ServerSocket server;
  BufferedReader in;  BufferedWriter out;
  String inputText, outputText;

  try {
     server = new ServerSocket(4004);
     clientSocket = server.accept();
     in = new BufferedReader(new
     InputStreamReader(clientSocket.getInputStream()));
     out = new BufferedWriter(new
     OutputStreamWriter(clientSocket.getOutputStream()));

     inputText = in.readLine();
     System.out.println(inputText);

     outputText = inputText+"sent";
     out.write(outputText + "\n");
     out.flush();
   }
  catch (Exception ex) {
              System.out.println("Error happend!");
       }}
```

```
public static void main(String[] args) {          //Клиент
  Socket clientSocket; String outText, inText;
  BufferedReader in; BufferedWriter out;
  Scanner scan = new Scanner(System.in);

  try {
      clientSocket = new Socket("localhost", 4004);
      in = new BufferedReader(new
      InputStreamReader(clientSocket.getInputStream()));
      out = new BufferedWriter(new
     OutputStreamWriter(clientSocket.getOutputStream()));

     outText = scan.nextLine();
     out.write(outText + "\n");
     out.flush();

      inText = in.readLine();
      System.out.println("Server sent me: " + inText);
      clientSocket.close();
      }
   catch (Exception ex) {
          System.out.println("Error happend!");
      }}
```

# Занятие 5. Темы

- Многопоточные приложения
- Оконные приложения. Библиотека Swing

# Многопоточные приложения

- В Java есть возможность писать многопоточные приложения, то есть такие программы, которые состоят из нескольких потоков, из нескольких частей кода выполняющихся практически одновременно.
- Основная цель создания многопоточных приложений это повышение общей производительности и сокращение времени реакции приложения.
- Многопоточные приложения создают как для многопроцессорных, так и для однопроцессорных систем.
- В однопроцессорной системе каждый поток получает некоторое количество квантов времени, по истечении которого управление передается другому потоку. Это создает впечатление одновременной работы нескольких потоков.
- Многопоточное приложение можно написать только в том случае, когда задачи решаемые программой можно распараллелить.
- Недостатки многопоточности:
  - Накладные расходы, связанные с переключением между потоками.
  - Проблемы синхронизации данных, возможность одновременного доступа к одним и тем же данным со стороны нескольких потоков.
  - Возможность взаимной блокировки потоков.

# Класс Thread

- Для работы с несколькими потоками в Java используются объекты класса Thread. Основные методы класса Thread:

| Метод | Описание |
|---|---|
| start | Запускает поток, вызывая его метод run() |
| sleep | Приостанавливает поток на заданное количество миллисекунд |
| run | Определяет точку входа в поток |
| join | Блокирует вызывающий поток до завершения другого потока |
| isInterrupted | Возвращает true, если поток был прерван |
| isAlive | Возвращает информацию о том, запущен поток или нет |
| getName | Возвращает имя потока |
| setName | Устанавливает текстовое имя потока |
| getPriority | возвращает приоритет потока |
| setPriority | устанавливает приоритет потока (от 1 до 10) |
| currentThread | Возвращает ссылку на выполняющийся поток |

# Многопоточные приложения

- В Java для создания вторичного потока исполнения следует получить экземпляр объекта типа **Thread**. Это можно сделать следующими двумя способами:
  - реализовав интерфейс Runnable
  - наследовав класс Thread
- В любом из этих двух случаев класс должен реализовать у себя метод **run()**, который и будет запущен в отдельном потоке.
- В методе **run()** устанавливается точка входа в другой, параллельный поток исполнения, и из него можно вызывать другие методы, использовать другие классы, объявлять переменные таким же образом, как и в главном потоке.
- После того как новый поток исполнения будет создан, он не запускается до тех пор, пока не будет вызван метод **start()**, объявленный в классе Thread. Вызов метода start() запускает метод run().
- После создания класса, реализующего интерфейс **Runnable**, следует создать объект этого класса и передать в конструктор класса Thread. Затем для объекта созданного из класса Thread вызывается метод start() для запуска второго потока.
- После создания класса, наследующего от Thread, следует создать объект этого класса и вызвать его метод start() для запуска второго потока.

# Пример реализации интерфейса Runnable

```
class SecondThread implements Runnable {
  public void run() {
  System.out.println("I'm second thread " +
  Thread.currentThread().getName());
  try {
    for (int i=0; i<10; i++)
    {
     System.out.println("S"+i);
     Thread.sleep(500);
    }
  }
  catch (InterruptedException ex)
  {
    System.out.println("Thread interrupted");
  }
  System.out.println(Thread.currentThread().getName() +
  " ended");
 }
}
```

```
public static void main(String[] args) {
  System.out.println("I'm main thread " +
  Thread.currentThread().getName());
  Thread st = new Thread(new SecondThread());
  st.start();
  try {
    for (i=0; i<10; i++)
    {
      System.out.println("F"+i);
      Thread.sleep(400);
    }
  }
  catch (InterruptedException ex) {
     System.out.println("Thread interrupted");
   }
}
```

# Пример наследования от класса Thread

```
class SecondThread extends Thread {
  int action, arr[];
  SecondThread(int a[], int action) {
     arr = a;
     this.action = action;
  }
  public void run() {
     System.out.println("I'm"+Thread.currentThread().getName());
     if (action==0)
        sum();
     else
        mul();
  }
 private void sum() {
     int i, sum=0;
     for (i=0; i<arr.length; i++)
         sum += arr[i];
     System.out.println("Sum = " + sum);
 }
private void mul() {
     int i, mul=1;
     for (i=0; i<arr.length; i++)
         mul *= arr[i];
     System.out.println("Mul = " + mul);
} }
```

```
public static void main(String[] args)
{
  int arr[] = new int[] {1,1,1,1,1,1,1,1,1,1};
  SecondThread st1 = new   SecondThread(arr, 0);
  SecondThread st2 = new SecondThread(arr, 1);
  st1.start();
  st2.start();
}
```

# Синхронизация потоков. Модификатор synchronized

- При построении многопоточных приложений необходимо гарантировать, что разделяемые потоками данные защищена от возможности их одновременного изменения разными потоками.
- Если один поток будет прерван до того, как завершит свою работу с разделяемыми данными, то второй поток после этого прочитает нестабильные данные.
- Для синхронизации доступа к разделяемым ресурсам в Java используется ключевое слова **synchronized**. Синхронизировать многопоточную программу можно двумя способами:
  - Используя синхронизированные методы
  - Используя оператор synchronized
- Для объявления синхронизированного метода перед его объявлением пишется ключевое слов synchronized: *доступ synchronized тип имяМетода (параметры)*
- Когда поток исполнения оказывается в теле синхронизированного метода, все другие потоки исполнения или любые другие синхронизированные методы, пытающиеся вызвать его для того же самого экземпляра, вынуждены ожидать.
- Блок команд после оператора synchronized может выполняться одновременно только одним потоком. (Первый поток, который входит в этот блок блокирует его для остальных).
- Формат оператора synchronized : *synchronized (выражение) блок_операторов*.
  В качестве выражения для блокировки используется или this или специально создаваемый объект.

# Пример синхронизации

```
class Work {
 int arr[];
 Work(int a[]) {
     arr = a;
 }

private synchronized void add(int elem) {
  arr[0]+=elem;
 }

void sum(int begin, int end) {
 try {
   for (int i=begin; i<end; i++) {
      add(arr[i]);
      Thread.sleep(200);
    }
  }
catch (InterruptedException ex) {
System.out.println("Thread interrupted");
}
}}
```

```
class SecondThreadSync extends Thread {
  int begin, end;
  Work w;

SecondThreadSync2(Work w, int begin, int end){
  this.w = w; this.begin = begin; this.end = end;
 }
 public void run()
 {
   w.sum(begin, end);
 }
}
```

```
public static void main(String[] args) {
int arr[] = new int[] {0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,};
Work w = new Work(arr);

try {
   SecondThreadSync st1 = new SecondThreadSync(w, 0, arr.length/2);
   SecondThreadSync st2 = new SecondThreadSync(w, arr.length/2, arr.length);
   st1.start();
   st2.start();

   st1.join();
   st2.join();
   System.out.println("arr[0] = " + arr[0]);
}
catch (InterruptedException ex) {
System.out.println("Thread interrupted");
}}
```

# Создание оконных приложений

- На языке Java можно разрабатывать также оконные приложение, где взаимодействие с пользователем осуществляется в графическом режиме, приложение выполняет вывод в отведенную ему прямоугольную область экрана, называемую окном, которое состоит из стандартных элементов, кнопок, надписей, полей для ввода данных, и т.д.
- При работе оконного приложения используется принцип событийного управления. После запуска программа ожидает действий пользователя и реагирует на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется событием.
- Для разработки оконных приложений на Java можно использовать разные библиотеки:
  - Библиотека AWT
  - Библиотека Swing
  - Библиотека JavaFX
- При создании проекта оконного приложения создается шаблон с пустым окном на которое можно добавлять необходимые элементы, перетаскивая их с панели элементов.
- Добавленным элементам можно затем прикреплять методы, которые будут вызваны при наступлении определенных событий.
- Формы (окна) в IDE Eclipse можно просматривать как в визуальном режиме (Design), так и в виде исходного кода (Source).
- Для создания оконного приложения Swing в Eclipse следует сначала создать обычный Java проект, а потом добавить к нему новое оконное приложение:
  New -> Other -> WindowBuilder -> Swing Designer -> Application Window

# Библиотека Swing. Класс JFrame

- В библиотеке Swing описан класс JFrame, представляющий собой окно с рамкой и строкой заголовка (с кнопками «Свернуть», «Во весь экран» и «Закрыть»). Оно может изменять размеры и перемещаться по экрану.
- Конструктор JFrame() без параметров создает пустое окно. Конструктор JFrame(String title) создает пустое окно с заголовком title.
- setSize(int width, int height) — устанавливает размеры окна. Если не задать размеры, окно будет иметь нулевую высоту независимо от того, что в нем находится и пользователю после запуска придется растягивать окно вручную. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.
- setDefaultCloseOperation(int operation) — позволяет указать действие, которое необходимо выполнить, когда пользователь закрывает окно нажатием на крестик. Обычно в программе есть одно или несколько окон при закрытии которых программа прекращает работу. Для того, чтобы запрограммировать это поведение, следует в качестве параметра operation передать константу EXIT_ON_CLOSE, описанную в классе JFrame.
- setVisible(boolean visible) — когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается данный метод с параметром true. Если вызвать его с параметром false, окно снова станет невидимым.

# Основные элементы оконного приложения

| Имя | Название | Описание |
|---|---|---|
| Метка | JLabel | Размещение текста на форме |
| Кнопка | JButton | Основное событие - щелчок мышью |
| Поле ввода | JTextField | Позволяет вводить и редактировать текст, который доступен через метод getText() |
| Многострочный ввод | JTextArea | Позволяет вводить многострочный текст |
| Флажок | JCheckBox | Для проверки, установлен ли флажок, анализируют его свойство Checked |
| Переключатель | JRadioButton | Позволяет выбрать один из нескольких предложенных вариантов |
| Меню | JMenuBar | Меню приложения, находящееся в верхней части программы |
| Панель элементов | JPanel | Пространство, на котором можно размещать другие элементы |
| Выбор файла | JFileChooser | Окно для выбора файла |

# Панель содержимого

- Напрямую в окне элементы управления не размещаются. Для этого служит панель содержимого, занимающая все пространство окна. Обратиться к этой панели можно методом **getContentPane()** класса JFrame. С помощью метода **add(Component component)** можно добавить на нее любой элемент управления.
- Также элементы можно размещать внутри панели **JPanel**, которая добавляется как один из элементов на окно.
- У каждой панели есть менеджер размещения, который определяет стратегию взаимного расположения элементов, добавляемых на панель. Его можно изменить методом **setLayout(LayoutManager manager)**. Есть следующие типы размещения элементов:
  - FlowLayout - Менеджер последовательного размещения.
  - BorderLayout - Менеджер граничного размещения
  - GridLayout - Менеджер табличного размещения
  - BoxLayout - Менеджер блочного размещения
- Если в качестве менеджера размещения панели установить null, элементы не будут расставляться автоматически. Координаты каждого элемента необходимо в этом случае указать явно.

# Диалоговые окна

- Диалоговое окно является особой разновидностью окна. У диалогового окна неизменяемые размеры и есть заданный набор кнопок, в зависимости от типа диалогового окна (OK, Cancel, Abort, Retry и т.д.)  При нажатии на любую из этих кнопок окно закрывается.
- После закрытия диалогового окна можно определить какая кнопка в нем была нажата и выполнить соответствующее действие.
- Открытие диалогового окна блокирует другие окна программы. Перейти к ним можно только после закрытия диалогового окна.
- Диалоговое окно открывается с помощью соответствующего статического метода класса JOptionPane:
  - showConfirmDialog
  - showInputDialog
  - showMessageDialog
  - showOptionDialog

```
btnDialog.addMouseListener(new MouseAdapter() {

public void mouseClicked(MouseEvent e) {
    if (JOptionPane.showConfirmDialog(null, "Вы
                                уверены, что хотите выйти?") ==
                                  JOptionPane.YES_OPTION)
        System.exit(0);
}
});
```

# Создание многооконной программы

- Оконное приложение может состоять из нескольких окон, вызывающих одно другое.
- Для создания второго окна следует добавить в приложение новый элемент типа «JFrame»
- Для отображения окна следует создать экземпляр объекта соответствующей формы, а затем сделать окно видимым методом setVisible(true).
- Для того чтобы закрыть окно нужно вызывать у метод dispose().

```
JButton btnFormAdd = new JButton("Добавить");
btnFormAdd.addMouseListener(new MouseAdapter() {

public void mouseClicked(MouseEvent arg0) {
    SecondForm sf = new SecondForm();
    sf.setVisible(true);
  }
});
```

# Передача данных между окнами

- Поскольку вторичное окно создается как объект внутри основного окна, то передавать данные из вторичного окна в основное можно просто обращаясь к открытым элементам вторичного окна из основного.
- Для передачи данных из основного окна во вторичное нужно изменить конструктор вторичного окна так чтобы он принимал параметры и при создании вторичного окна передавать те параметры, которые должно получить вторичное окно.
- В конструктор вторичного окна можно передавать либо отдельные данные, либо все основное окно целиком с помощью this.

**Основная форма**

```
private String info;

JButton btnFormAdd = new JButton("Добавить");
btnFormAdd.addMouseListener(new MouseAdapter(){

public void mouseClicked(MouseEvent arg0) {
    SecondForm sf = new SecondForm(info);
    sf.setVisible(true);
  }
});
```

**Вторичная форма**

```
public class SecondForm extends JFrame {

 private String info;

 public SecondForm(String info) {
     this.info = info;
  }
}
```

# Пример заполнения таблицы JTable

```
DefaultTableModel tableModel = new DefaultTableModel();

String[] columnNames = {"ID", "Name", "Family", "Gender", "Age"};

String[][] arrs = {{"1", "Vasya", "Ivanov", "M", "23"},
                   {"2", "Petya", "Sidorov", "M", "54"},
                   {"3", "Olya", "Kot", "F", "32"}};

tableModel.setColumnIdentifiers(columnNames);

for (int i = 0; i < arrs.length; i++)
        tableModel.addRow(arrs[i]);

Jtable table = new JTable(tableModel);

JScrollPane jpane = new JScrollPane(table);
contentPane.add(jpane);
```