Введение в ООП

Березовская Юлия Владимировна
myumla.myu@gmail.com
Токаревская Светлана Анатольевна
s.tokarevskaya@narfu.ru

Содержание

Лекция 1

Высокоуровневые методы информатики и программирования

Программирование

- ✔ Программирование – технология создания программ.
- ✔ Программа представляет собой набор инструкций процессора.
- ✔ Чем выше уровень языка, тем в более простой форме записываются одни и те же действия.
- ✔ Высокоуровневые методы программирования:
  - методы структурного программирования;
  - ООП.

Парадигмы программирования

- некоторые методы и стили, которые становятся общепринятыми на некоторое время.

1) Первые программы, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста.
2) Создание машинного кода компилятором (FORTRAN, Algol и др.).
3) Процедурное программирование.
4) Структурное программирование (напр. Pascal).
5) Модульное программирование.
6) Объектное программирование.

Объектно-ориентированное программирование (ООП)

- технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств.

Этапы разработки ПО:

- Постановка задачи
- Анализ предметной области задачи;
- Проектирование системы;
- Реализация системы;
- Модификация.

Анализ предметной области

Цель:

Максимально полное описание задачи.

Выполняется:

Объектная декомпозиция системы

Основные особенности поведения объектов

Результат:

Диаграмма объектов: основные объекты и сообщения, передаваемые между ними.

Проектирование системы

Логическое проектирование: разработка структуры классов (поля, методы).

Результат: диаграмма классов, отражающая структуру классов и взаимоотношения между ними.

Физическое проектирование: объединение описаний классов в модули, определение способов взаимодействия с оборудованием, ОС, другим ПО и т.д.

Результат: схема объединения классов в модули, описание интерфейсов взаимодействия

Проектирование системы

Результат:

- Модель проектируемой системы, которая отображает только важные для поставленной задачи черты, а остальные представляет в упрощенном виде или вовсе отбрасывает.
- Модель содержит требования к иерархии классов представляющей предметную область, к разбиению системы на модули.

Для создания модели пользуются **специальным языком моделирования**, например, UML.

Эволюция (реализация) системы

– процесс поэтапной реализации классов и подключения объектов к системе.

- Абстракция;
- Инкапсуляция;
- Наследование;
- Полиморфизм;
- Модульность.

Объектно-ориентированные языки программирования

- ***Язык программирования*** (алгоритмический язык) – это набор правил, определяющих, какие последовательности символов составляют программу (синтаксические правила) и какие вычисления описывает программа (семантические правила).
- ***Программа*** – текст, задающий множество процессов вычислений, в соответствии с которым исполнитель, понимающий программу, разворачивает какой-то один из них.

Объектно-ориентированные языки программирования

- Объектно-ориентированные языки содержат конструкции, позволяющие определять объекты, принадлежащие классам и обладающие ***свойствами***:
  - **инкапсуляции,**
  - **наследования,**
  - **полиморфизма.**

Объектно-ориентированные языки программирования

Объектно-ориентированные языки можно разделить на **три группы**:

- **чистые**: Simula (1962); Smaltalk (1972); Beta (1975); Self (1986); Cecil (1992).
- **гибридные**: C++ (1983); Object Pascal (1984).
- **урезанные**: Java (1995); C# (2000).

Лекции 2-3

Технология Java
Классы и объекты

ОБЗОР ТЕХНОЛОГИИ JAVA

Язык Java

Язык программирования Java – язык программирования высокого уровня, обладающий характеристиками:

- объектно-ориентированный;
- простой;
- распределенный;
- многопоточный;
- динамичный;
- архитектурно (аппаратно) независимый;
- переносимый;
- высокопроизводительный;
- надежный (устойчивый к сбоям);
- безопасный.

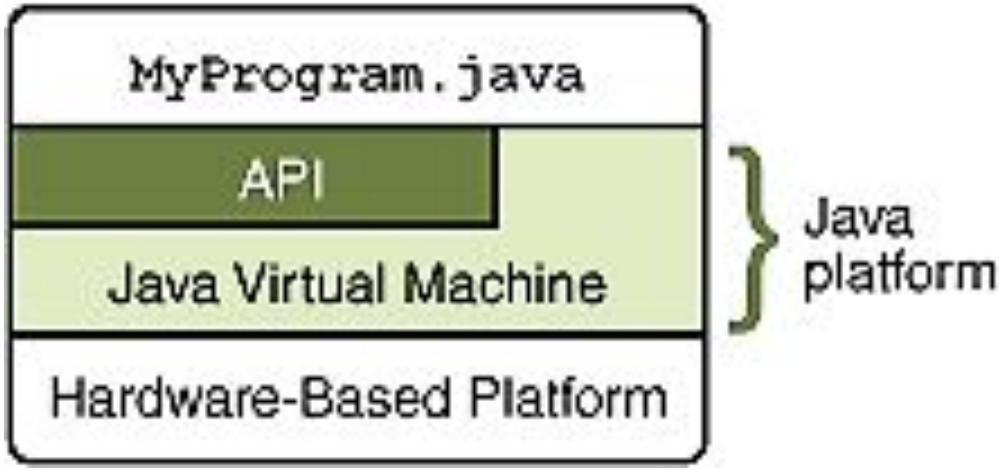
Подробнее: http://java.sun.com/docs/white/langenv/

Платформа Java

Платформа – окружение из аппаратного или программного обеспечения, в котором выполняется программа. В большинстве случаев платформа рассматривается как объединение ОС и аппаратного обеспечения (железа), на котором функционирует ОС.

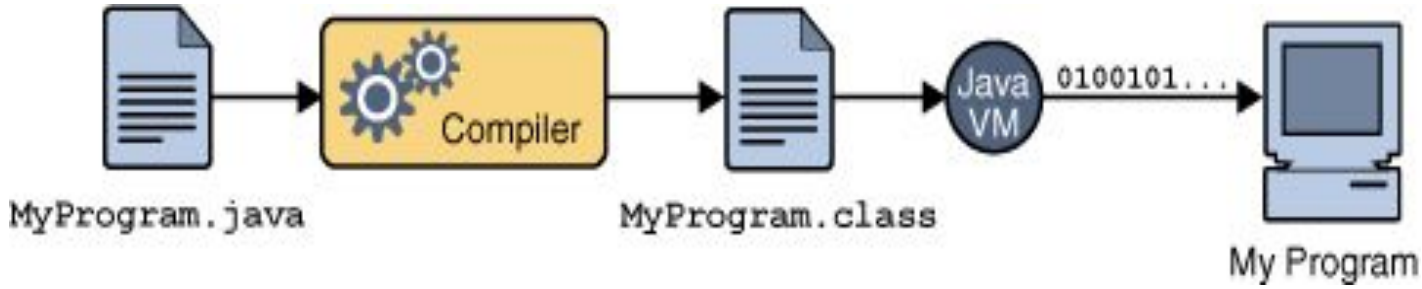
Java-*платформа* – исключительно программное обеспечение, функционирующее над платформой, основанной на аппаратном обеспечении, позволяющее исполнять Java-программы.

Платформа Java



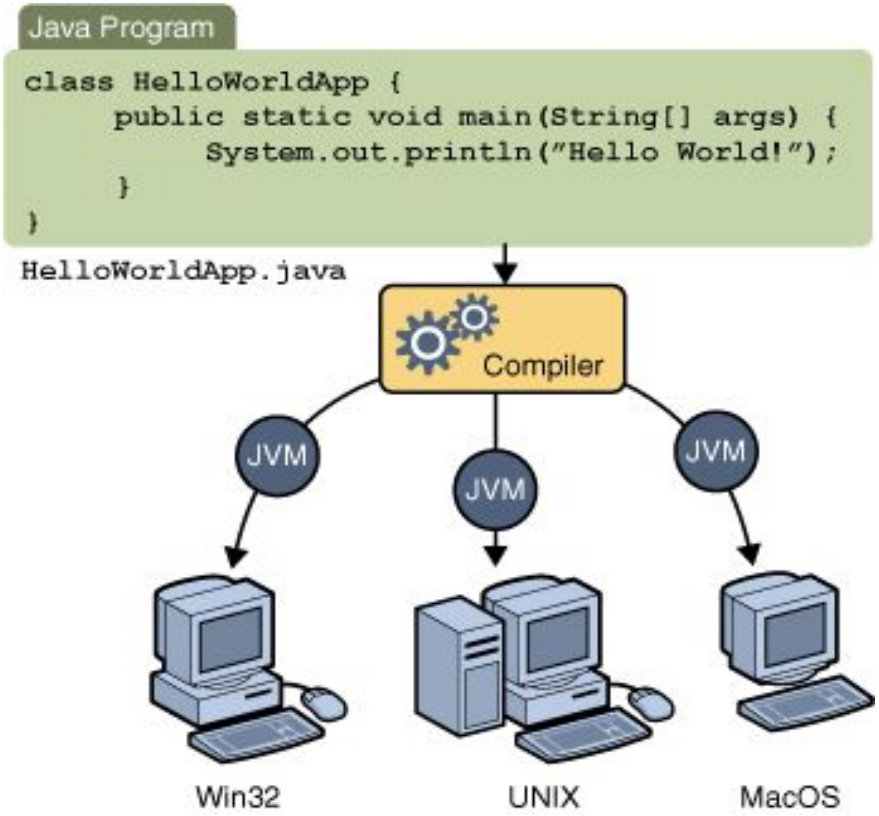
http://download.oracle.com/javase/tutorial/getStarted/intro/definition.html

Создание и выполнение Java-программ:



http://download.oracle.com/javase/tutorial/getStarted/intro/definition.html

Переносимость Java-программ



http://download.oracle.com/javase/tutorial/getStarted/intro/definition.html

Почему Java?

1. Одна программа может быть запущена на различных компьютер различными платформами. Программы на *Java* выполняются внутри специальной программной оболочки, которая называется **виртуальная машина** (JVM).

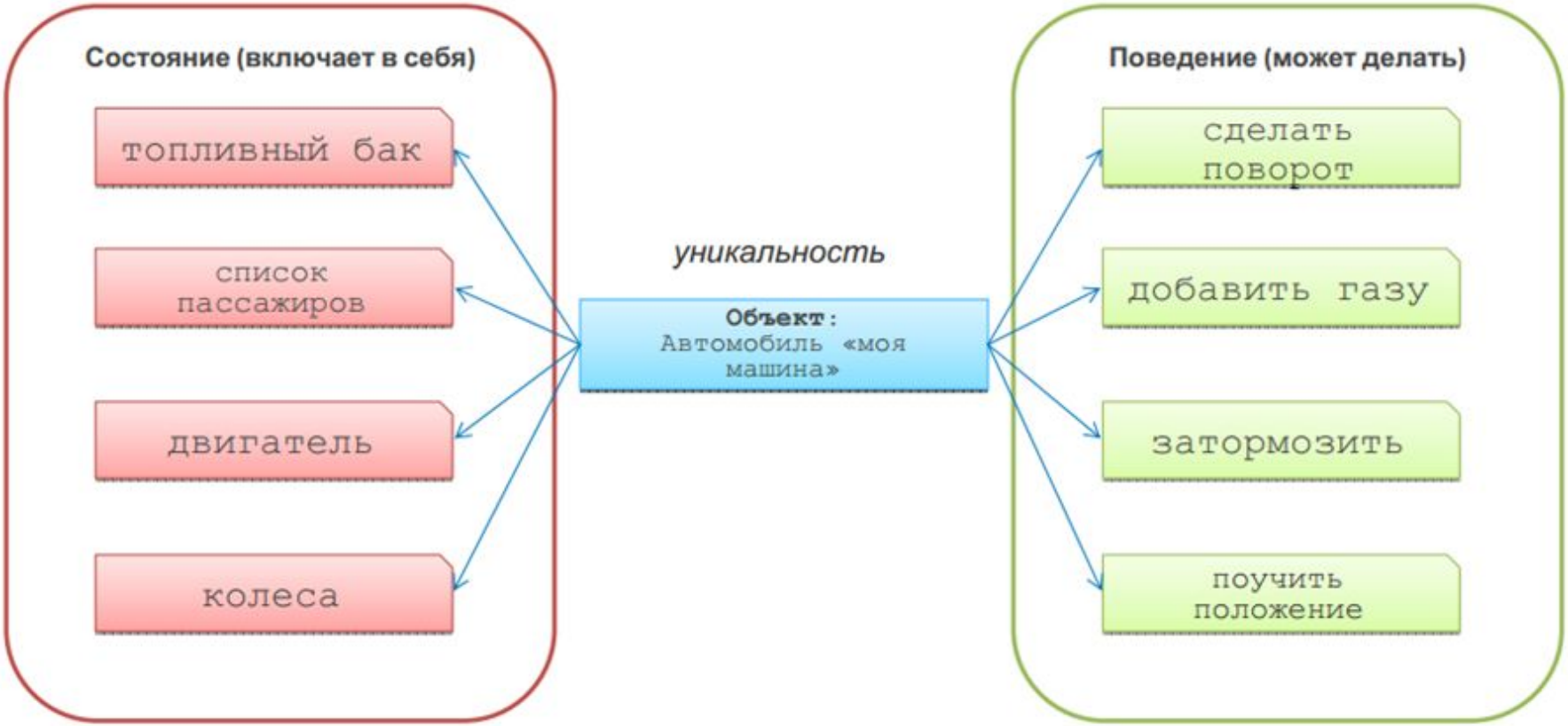
2. *Java* позволяет создавать программные элементы (*классы*), которые представляют объекты из реального мира.

Например, можно создать класс *Java* с именем CAR (автомобиль), и создать свойства этого класса, такие, как двери, колеса,… После этого можно создать другой класс, например FORD, который будет иметь все свойства класса CAR, при этом оставаясь в чем-то уникальным.

3. *Java* обладает огромным количеством дополнительных и бесплатных примочек (программных библиотек).

4. *Java* поставляется бесплатно!

Пример


Состояние (включает в себя)
топливный бак
список пассажиров
двигатель
колеса
уникальность
Объект:
Автомобиль «моя машина»
Поведение (может делать)
сделать поворот
добавить газу
затормозить
поучить положение

Пример


```
Автомобиль мояМашина = я.купитьМашину();
Двигатель двигательМоейМашины = мояМашина.двигатель;
двигательМоейМашины.переключитьРежим(форсированный);
```

```
class Автомобиль {
    ТопливныйБак бак;
    СписокПассажиров пассажиры;
    Колеса колеса;
    Двигатель двигатель;

    процедура сделатьПоворот(угол);
    процедура добавитьГазу(уровень);
    процедура затормозить();
    Координаты получитьПоложение();
}
```

```
class Двигатель {
    Свечи свечи;
    Цилиндры цилиндры;
    Карбюратор карбюратор;

    процедура увеличитьОбороты();
    процедура переключитьРежим(режим);
}
```

Пример


```
Автомобиль мой = new Автомобиль();
Автомобиль жены = new Автомобиль();
Топливо топливо = жены.слитьТопливо();
мой.заправить(топливо);
мой.двигаться();
```

```
class Автомобиль {
    Двигатель двигатель;
    процедура двигаться() {
        двигатель.завести();
        двигатель.добавитьГазу();
    }
}
```

История возникновения Java

Создатель Java – Джеймс Гослинг (США)

- Первое применение – бытовая электроника (микроволновые печи, стиральные машины, пульты управления)

Первое название языка – Oak («Дуб»)

- В честь дуба, стоявшего напротив офиса Джеймса Гослинга
- К тому времени уже был ещё один язык Oak

Название Java произошло от сорта кофе

- Это кофе производится на о. Ява (Индонезия)
- Его очень часто употреблял и первые разработчики языка

Duke - талисман языка Java

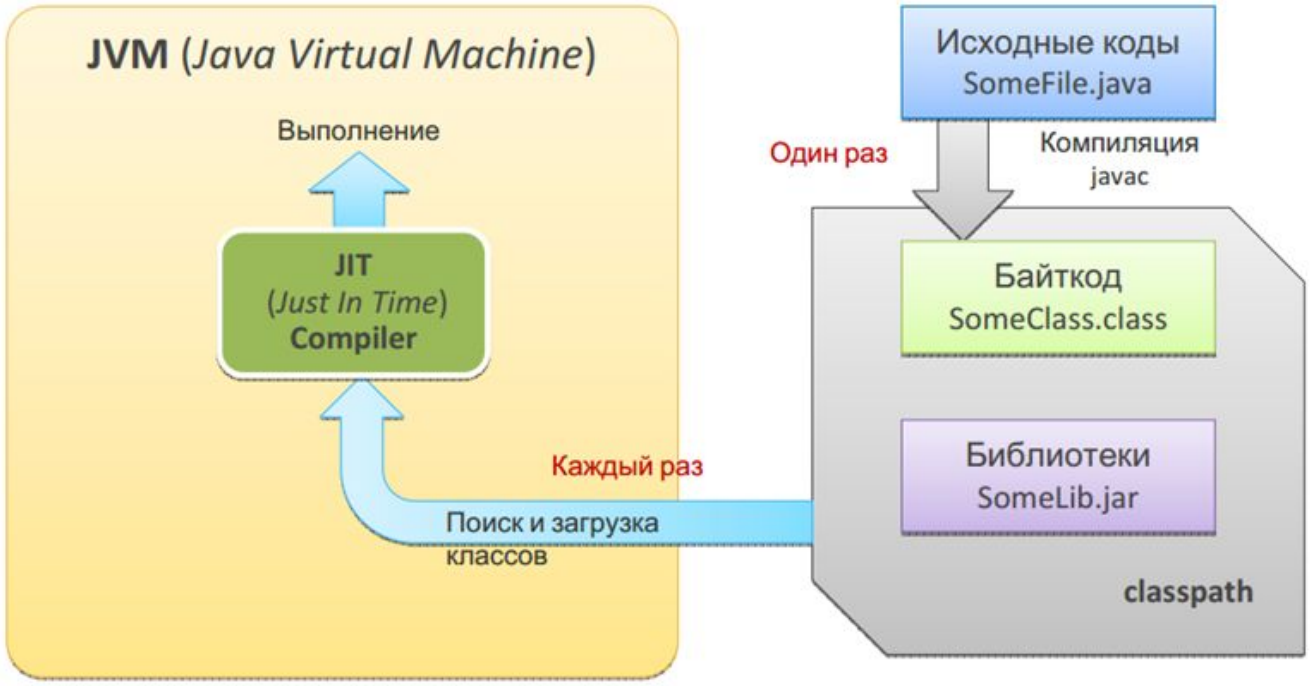
- Ежегодно проводится конкурс Duke Choice Awards
- В 2011 году Duke изменил свой внешний вид

Особенности Java

- кроссплатформенность
- объектная ориентированность
- привычный синтаксис C/C++
- безопасность
- ориентация на Internet
- динамичность
- простота освоения

Три основных шага в программировании

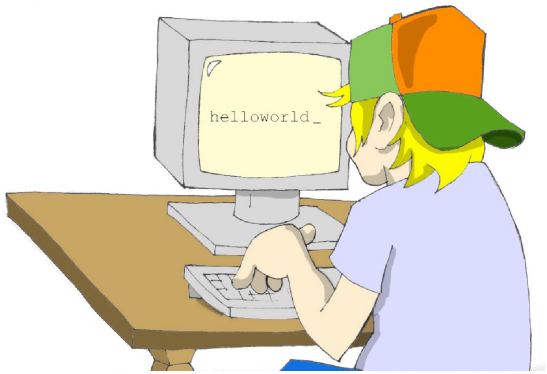
1. Написать программу на Java и сохранить ее на диск.
2. *Выполнить компиляцию* программы, чтобы перевести ее с языка Java в специальный байт-код.
3. Запустить программу



Три основных шага в программировании

1. **Ввод текста программы** – любой текстовый редактор. Сохранить ее на диск с расширением **.java.** Например, ***HelloWorld.java***

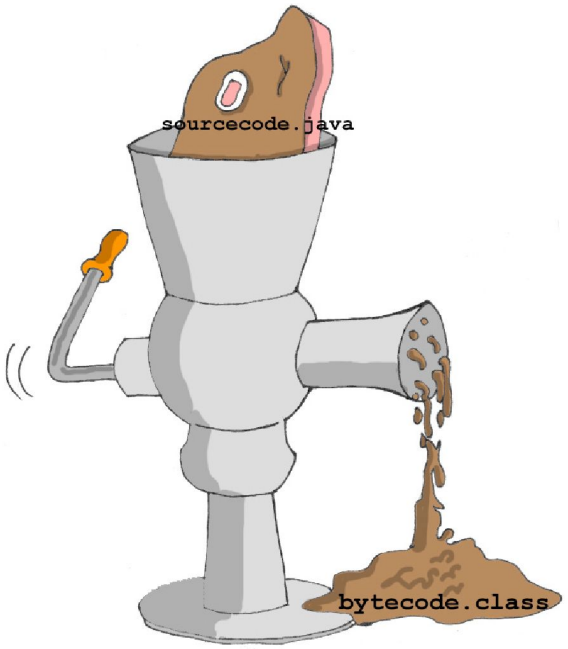
```
public class HelloWorld {

        public static void main(String[] args) {

                System.out.println("Hello World");
        }
}
```



Три основных шага в программировании

2. **Компиляция программы**, чтобы перевести ее с языка Java в специальный байт-код. Компилятор **javac,** который входит в состав пакета **JDK**.

```
javac HelloWorld.java
```



Три основных шага в программировании

3. **Запуск программы**. `java HelloWorld`

Интегрированная среда разработки *Eclipse IDE*

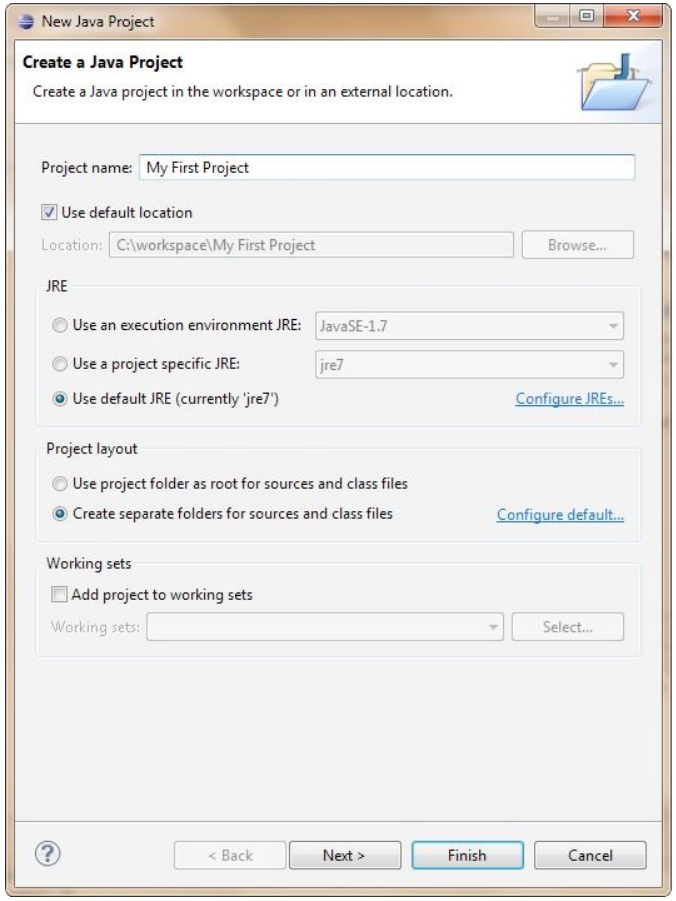
www.eclips.org

1. Чтобы создать новый проект в Eclipse, выберите следующие пункты меню:

- File («Файл»),
- New («Создать»),
- Java Project («Проект Java»).

2. Введите имя нового проекта, например, ***My First Project***.

3. Поле **Location** («Расположение») – место хранение вашего проекта.



Создание программ в *Eclipse IDE*

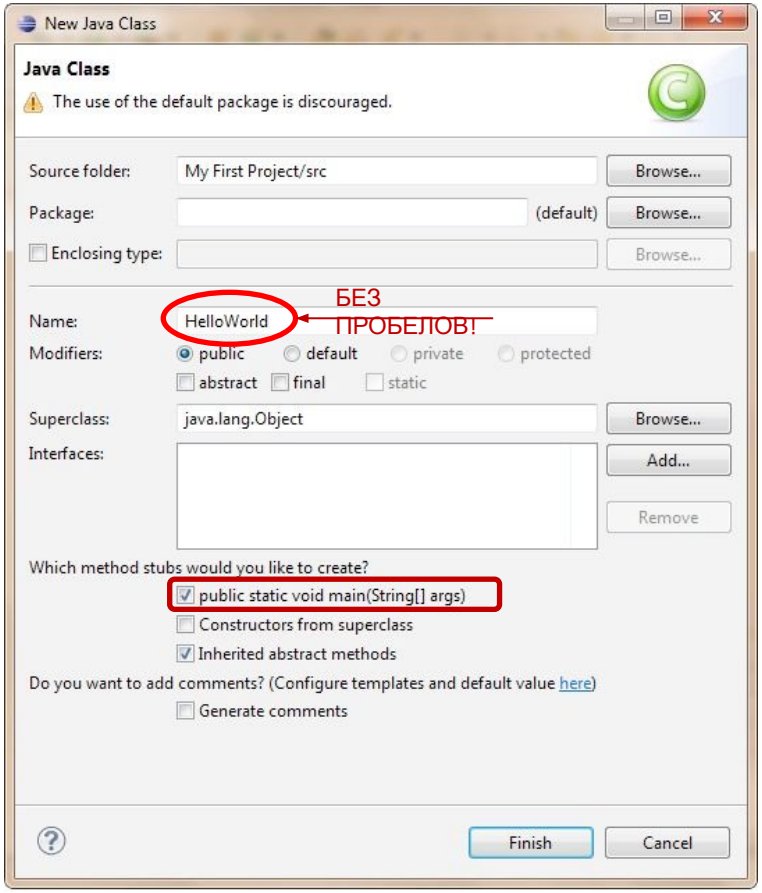
1. Программа в Java – это ***класс***.
В Eclipse, выберите следующие пункты меню:

- File («Файл»),
- New («Создать»),
- *Class*(«Класс»).

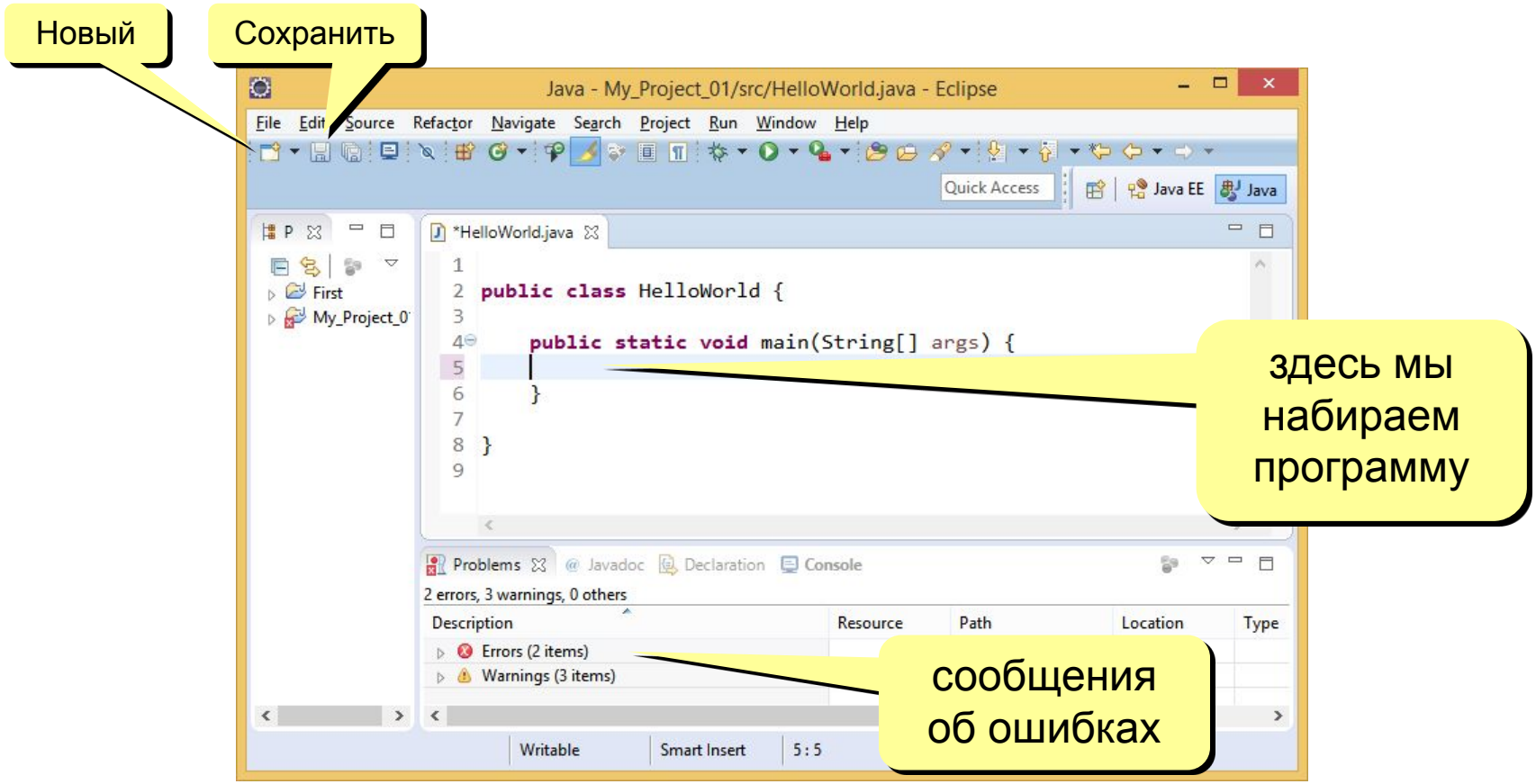
2. Введите имя нового класса, например, ***HelloWorld***.

3. В разделе ***Which methods stubs you would like to create*** («Какие заготовки для методов необходимо создать?») установите флажок для метода **main()**.

4. Нажмите **Finish.**



Как начать работу?



КЛАССЫ И ОБЪЕКТЫ

Классы основные строительные элементы Java-программы.

```
// заголовок класса
class MyClass
{
 //тело класса:
 // поля, конструкторы и методы
}
```

Заголовок класса

1. Модификатор управления доступом (**`public`**, **`protected`**, **`private`**)
2. Ключевое слово **`class`**
3. Название класса с **Б**ольшой**Б**уквы
4. Имя класса предка, предваренное словом **`extends`**
5. После слова **`implements`** через запятую имена интерфейсов, реализуемых классом

Тело класса

Члены класса:

- **Поля** – переменные и константы, характеризующие объект;
- **Методы** – процедуры, описывающие поведение объекта;
- **Вложенные классы**;
- **Вложенные интерфейсы**.

Модификаторы управления доступом

- **`public`**

определяет, что следующие за ним определения доступны всем;

- **`private`**

означает, что следующие за ним определения может использовать только создатель типа, внутри функций членов этого типа;

- **`protected`**

по действию схож с **`private`**, за одним исключением: унаследованные классы имеют доступ к членам, помеченным **`protected`**, хотя и не имеют доступа к **`private`**-членам.

Модификаторы управления доступом

- Уровень класса - модификатор **`public`** или никакого;
- Уровень члена класса – модификаторы: **`public`**, **`private`**, **`protected`** или никакого.

Модификатор	Класс	Пакет	Класс-потомок	Все классы
`public`	+	+	+	+
`protected`	+	+	+	−
`нет`	+	+	−	−
`private`	+	−	−	−

Создание экземпляра класса

Этапы:

- Объявление объектов

```
MyClass object1, object2;
```

- Выделение памяти под объекты

```
object1 = new MyClass();
```

- Инициализация объектов

`MyClass()` – конструктор

Особенности конструктора:

- ✔ конструктор имеется в любом классе;
- ✔ конструктор выполняется автоматически при создании экземпляра класса;
- ✔ конструктор не возвращает никакого значения;
- ✔ конструктор нельзя наследовать и переопределить в подклассе;
- ✔ конструктор может содержать:
  - вызов конструктора суперкласса (**`super`**);
  - вызов другого конструктора того же класса (**`this`**).

Класс `DynArray`

Класс DynArray

```
int size;     // текущий размер массива
int maxSize; // размер отведенной памяти
int[] array; // сам массив
```

```
    // Конструкторы:
DynArray (int sz);
DynArray (int sz, int maxSz);
DynArray (int sz, int maxSz, int[] iniArray);
    // Методы:
int elementAt (int i); // операция выборки элемента
void resize (int delta);  // изменение текущего размера
           // массива
void enlarge (int delta); // операция расширения массива
void shrink (int delta);  // операция уменьшения массива
void add (int e);       //  добавление одного нового элемента
               // (с возможным расширением массива)
boolean equals();
String toString();
```

Класс **DynArray** - атрибуты

```
public class DynArray {

    // класс имеет три поля
    int size;    // текущий размер массива
    int maxSize; // размер отведенной памяти
    int[] array;   // сам массив
}
```

Класс **DynArray** - конструкторы

```
// аргумент указывает, сколько памяти надо
// отвести под его элементы

    public DynArray (int sz){
        this (sz, sz, null);
    }
```

Класс **DynArray** - конструкторы

```
// аргументы указывают, сколько памяти
// используется под элементы и сколько
// отведено всего

    public DynArray (int sz, int maxSz){
        this (sz, maxSz, null);
    }
```

Класс **DynArray** - конструкторы

```
public DynArray (int sz, int maxSz,int[] iniArray){
   size = sz;
   maxSize = (maxSz < sz ? sz : maxSz);
   array = new int[maxSize];
   if (iniArray != null){
     for (int i=0; i < size && i < iniArray.length; i++)
        array[i] = iniArray[i];
      }
}
```

Класс DynArray

```
// операция выборки элемента

public int elementAt (int i){
    return array[i];
}
```

Класс DynArray

```
// изменение текущего размера массива,
// аргумент delta задает размер изменения

public void resize (int delta){
    if (delta > 0) enlarge(delta);
    else if (delta < 0) shrink(-delta);
}
```

Класс **DynArray**

```
// операция расширения массива
void enlarge (int delta){
     if ((size += delta) > maxSize) {
          maxSize = size;
          int[] newArray = new  int[maxSize];
          for (int i=0; i < size - delta; i++)
               newArray[i] = array[i];
          array = newArray;
     }
}
```

Класс DynArray

```
// операция уменьшения массива

void shrink (int delta){
    size = (delta > size ? 0 : size - delta);
}
```

Класс DynArray

```
//  добавление одного нового элемента
// (с возможным расширением массива)

void add (int e){
    resize(1);
    array[size-1] = e;
}
```

Иерархия классов в Java

- Вершина иерархии классов Java - класс **`Object`**;
- Все классы наследники класса **`Object`**, т.е. ссылочная переменная типа **`Object`** может обращаться к объекту любого класса;
- Запрещено множественное наследование.

Класс `Object`

Методы:

- `equals()` сравнивает данный объект на равенство с объектом, заданным в аргументе, возвращает логическое значение.
- `toString()` пытается содержимое объекта преобразовать в строку символов и возвращает объект класса `String`.

Задание:

1. Внести изменения в класс **`DynArray`** так, чтобы элементы массива могли быть экземплярами произвольного класса.

   (при определении массива используйте тип данных **`Object`**)
2. Переопределите методы **`equals()`** и **`toString()`** (класса **`DynArray`**).
3. Напишите класс **`DynArrayTest`**, тестирующий работу класса **`DynArray`**.

Интерфейсы в Java

Интерфейсы в Java

Интерфейс – это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию.

Интерфейс – ссылочный тип данных, подобный классу, который может содержать только константы, заголовки методов и вложенные типы (классы и интерфейсы).

Определение интерфейса

```
public interface имя extends интерфейс1,
             интерфейс2 {

    тип имя_константы = значение;
    тип_результата имя_метода(параметры_метода);

}
```

Тело интерфейса

Заголовки методов

- не содержат фигурных скобок (не определяют реализацию);
- отделяются точкой с запятой;
- методы являются `public`, поэтому этот модификатор не пишется.

Тело интерфейса

Объявление констант

- Любая переменная, объявленная в интерфейсе является `public`, `static` и `final` поэтому эти модификаторы не пишутся.
- Предполагается обязательное присвоение значения константе в теле интерфейса.

Реализация интерфейсов

```
class имя_класса
    [extends суперкласс]
    [implements интерфейс0 [, интерфейс1...]]

{ тело класса }
```

Реализация интерфейсов

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант. В том случае, когда вы реализуете в классе какой-либо интерфейс, все имена переменных этого интерфейса будут видимы в классе как константы.

Реализация интерфейсов

Если интерфейс не включает в себя методы, то любой класс, объявляемый реализацией этого интерфейса, может вообще ничего не реализовывать. Для импорта констант в пространство имен класса предпочтительнее использовать переменные с модификатором `final`.

Реализация интерфейсов

Если интерфейс включает в себя заголовки методов, то любой класс, объявляемый реализацией этого интерфейса, должен содержать реализацию всех методов, описанных в интерфейсе.

Пример: Классы и интерфейсы

```
public class IntList {
    //внутренний класс
     static class ListItem {
           int item;
           ListItem next;

           public ListItem(int i, ListItem n){
                item=i;
                next=n;
           }
      };
```

Пример: Классы и интерфейсы

```
//поля класса
 int count = 0;
 ListItem first = null;
 ListItem last = null;
```

Пример: Классы и интерфейсы

```
//создание пустого списка
public IntList(){}

//создание копии уже имеющегося списка
public IntList(final IntList src){
   addLast(src);
//добавляет список src в конец списка this
}
```

Пример: Классы и интерфейсы

```
//добавление элементов в конец списка
public void addLast(final IntList src) {
  for(ListItem cur = src.first; cur != null;
              cur = cur.next)
        addLast(cur.item);
}
```

Пример: Классы и интерфейсы

```
//добавление элемента в конец списка
public void addLast(int item){
      ListItem newItem = new ListItem(item,
  null);
      if (last == null){
            first = newItem;
      } else {
            last.next = newItem;
      }
      last = newItem;
      count++;
}
```

Пример: Классы и интерфейсы

```
//добавление элемента в начало списка
public void addFirst(int item){
    ListItem newItem = new
  ListItem(item,first);
    if (first == null){
        last = newItem;
    }
    first = newItem;
    count++;
}
```

Пример: Классы и интерфейсы

```
//удаление первого элемента из списка
public int remove(){
    int res = first.item;
    first = first.next;
    count--;
    return res;
}
```

Пример: Классы и интерфейсы

```
public interface Visitor {
    void visit (int item);
}

//В класс IntList добавим метод-итератор:

public void iterator (Visitor visitor){
     for (ListItem cur = first; cur!=null;
           cur = cur.next){
          visitor.visit(cur.item);
     }
}
```

Интерфейс – тип данных

Если тип переменной определен как интерфейс, то объект, присвоенный этой переменной, должен быть экземпляром класса, реализующего этот интерфейс.

Продолжаем пример:

```
public class Summator implements Visitor {
    int sum = 0;
    String s ="";
    public void visit(int item){
          s+=(item+" ");
          sum+=item;
      }
    public int getSum(){return sum;}
    public String getList(){return s;}
}
```

Пример. Классы и интерфейсы

```
public class IntListTest {
     public static void main(String[] args) {
          IntList lst = new IntList();
          for (int i=0; i<10; i++){
               lst.addFirst(2*i);
               lst.addLast(20-2*i);
          }
          System.out.println(lst.getCount());
          Summator summator = new Summator();
          lst.iterator(summator);
          System.out.println(summator.getList());
          System.out.println(summator.getSum());
     }
}
```

Итак:

Интерфейс определяет протокол взаимодействия двух объектов.

Объявление интерфейса содержит сигнатуры методов, но не их реализации

Объявление интерфейса может содержать определение констант.

Класс, реализующий интерфейс должен реализовывать все методы объявленные в интерфейсе.

Имя интерфейса может использоваться везде, где может использоваться тип данных.

Ключевое слово `abstract`:

- Иногда бывает удобным описать только заголовок метода, без его тела, и таким образом объявить, что такой метод будет существовать в этом классе.
- Реализацию этого метода, то есть его тело, можно описать позже.

Пример

Необходимо создать набор графических элементов

- геометрические фигуры - круг, квадрат, звезда и т.д.;
- элементы пользовательского интерфейса - кнопки, поля ввода и т. д.

Есть специальный контейнер, который занимается их отрисовкой.

- Внешний вид каждой компоненты уникален, а значит, соответствующий метод (назовем его `paint()`) будет реализован в разных элементах совсем по-разному.
- У компонент может быть много общего: занимаемая область, цвет, видимость,…

Удобно объявить **абстрактный метод в родительском классе**: у него нет внешнего вида, но известно, что он есть у каждого наследника!

Пример. Абстрактный класс

```
public class AbstructFigure {
   protected int x;
   protected int y;
   protected Color c;

   public abstract void paint();
   ...
}
```

Пример:
Ссылки на метод как параметры методов

Начиная с JDK 8 в Java можно в качестве параметра в метод передавать ссылку на другой метод.

Ссылка на метод передается в виде:

- Статический метод

```
имя_класса::имя_статического_метода
```

- Нестатический метод

```
объект_класса::имя_метода
```

Пример: класс **Pet**

```
public class Pet {
    protected String name;
    protected int countLegs;
    protected int countEaes;

    public Pet(String name){
        this.name = name;
        countLegs = 4;
        countEaes = 2;
    }
    public Pet(String name, int countLegs, int countEaes){
        this.name = name;
        this.countLegs = countLegs;
        this.countEaes = countEaes;
    }
```

Пример: класс **Pet** (продолжение)

```
    public int getCountLegs(){
        return countLegs;
    }
    public int getCountEaes(){
        return countEaes;
    }
    public String toString(){
        return this.getClass().getSimpleName() + "\n  Name = " + name
+ "\n  countLegs = " + countLegs + "\n  countEaes = " + countEaes +
"\n";
    }
}
```

Потомки класса **Pet**: **Cat** и **Spider**

```
public class Cat extends Pet{
    public Cat(String name) {
        super(name);
    }
}

public class Spider extends Pet{
    public Spider(String name) {
        super(name, 8, 2);
    }
}
```

Функциональный интерфейс **Expression**

Здесь определен функциональный интерфейс **Expression**, который имеет один метод.

```
int getCount(Pet pet);
```

Аргумент имеет тип объекта (мы хотим применять к экземпляру Pet), к которому применен метод тестирующего класса (см [5]), результат – тип возвращаемого значения (мы получаем **int** – количество лап или глаз).

```
public interface Expression {          int
getCount(Pet pet);
}
```

Тестирующий класс

```
public class test {
   public static void main(String[] args) {
    int n = 4;
    Pet[] p = new Pet[n];
    p[0] = new Cat("Maya");
    p[1] = new Spider("Bum");
    p[2] = new Cat("Puma");
    p[3] = new Cat("Sun");
    System.out.println("nLegs = " + nCount(p, Pet::getCountLegs));
System.out.println("nEaes = " + nCount(p, Pet::getCountEaes));
   }
   public static int nCount(Pet[] p, Expression s) {
    int result = 0;
    for (Pet i : p) {
        result += s.getCount(i);                    }
    return result;
   }
}
```

Пакеты в Java

Пакет – объединение классов и интерфейсов

- объединение родственных классов и интерфейсов;
- упрощение поиска классов и интерфейсов, выполняющих определенные функции;
- исключение конфликта имен, каждый пакет имеет свое пространство имен;
- неограниченный доступ классов и интерфейсов, объединенных в пакет, друг к другу.

Использование пакета

Создание пакета

оператор:

```
package имя_пакета;
```

Обращение к членам пакета из другого пакета

- `имя_пакета.имя_класса;` (полное имя класса)
- `import имя-пакета.имя_класса;` (импорт класса)
- `import имя_пакета.*;` (импорт целого пакета)

Особенности импорта пакетов

- импорт по умолчанию - **java.lang**, текущий пакет, безымянный пакет;
- импорт только классов и интерфейсов данного пакета, но не подпакетов:

```
import имя_пакета.*;
```

- если два импортированных пакета имеют классы с одинаковыми именами, то необходимо использовать полное имя класса:

```
import  имя_пакета.имя_подпакета.*;
```

- импорт статических полей и методов.

Иерархия пакетов

Древовидная структура (иерархия) пакетов и подпакетов в точности отображается на структуру файловой системы.

Все файлы с расширением **`class`** (содержащие байт-коды), образующие пакет, хранятся в одном каталоге файловой системы.

Подпакеты собраны в подкаталоги этого каталога.

Лирика

Структура исходного файла с текстом программы на языке **Java**:

- Необязательный оператор **package**.
- Необязательные операторы **import**.
- Описания классов и интерфейсов.

Правила:

- В файле только один открытый **public-класс**.
- Имя файла совпадает с именем открытого класса.

Лирика

Cod Conventions

Pascal naming convention – все слова в имени начинаются с заглавной буквы, используется для именования классов;

Camel naming convention – все слова в имени кроме первого начинаются с заглавной буквы, используется для именования полей и методов.

public-класс записывается первым в файле.

Обработка исключений

Обработка исключений

Исключительные ситуации (**`exceptions`**) могут возникнуть во время выполнения (**`runtime`**) программы, прервав ее обычный ход. К ним относится деление на нуль, отсутствие загружаемого файла, отрицательный или вышедший за верхний предел индекс массива, переполнение выделенной памяти и масса других неприятностей, которые могут случиться в самый неподходящий момент.

Обработка исключений

Исключение в языке **Java** — это объект, который описывает исключительную (т. е. ошибочную) ситуацию, произошедшую в некоторой части кода. Когда исключительная ситуация возникает, создается объект, представляющий это исключение, и "вбрасывается" в метод, вызвавший ошибку.

Обработка исключений

Если в программе не описан обработчик исключения, то исключение захватывается обработчиком, заданным исполнительной системой **`Java`** по умолчанию.

Любое исключение, которое не захвачено программой, будет в конечном счете выполнено обработчиком по умолчанию.

Обработка исключений

Преимущества самостоятельной обработки исключений:

1) позволяет фиксировать ошибку,
2) предохраняет программу от автоматического завершения.

Обработка исключений

в **Java** управляется с помощью пяти ключевых слов:

- `try`
- `catch`
- `finally`
- `throw`
- `throws`

```
try {
```
Программные операторы, которые нужно контролировать относительно исключений
```
} catch (ExceptionType1 exOb) {
```
Операторы, обрабатывающие исключение **ExceptionType1**, если оно возникло в блоке **try**
```
} catch (ExceptionType2 exOb) {
```
Операторы, обрабатывающие исключение **ExceptionType2**, если оно возникло в блоке **try**
```
}  finally {
```
Операторы, которые должны быть выполнены перед возвратом из **try**-блока
```
}
```

Обработка исключений

Когда вы используете множественные **`catch`-операторы**, важно помнить, что в последовательности **`catch`**-предложений подклассы исключений должны следовать перед любым из их суперклассов.

Кроме того, в **`Java`** недостижимый код — ошибка.

Иерархия классов исключений

- **`Throwable`**
  - **`Exception`**
    - **`RuntimeException`**
      - …
  - **`Error`**
    - …

Обработка исключений

`Error` определяет исключения, перехват которых вашей программой при нормальных обстоятельствах не ожидается.

Исключения типа **`Error`** применяются исполнительной системой **`Java`** для указания ошибок, имеющих отношение непосредственно к среде времени выполнения.

Исключения типа **`Error`** обычно создаются в ответ на катастрофические отказы, которые, как правило, не могут обрабатываться вашей программой.

Обработка исключений

`Exception` - используется для исключительных состояний, которые должны перехватывать программы пользователя.

Собственные заказные типы исключений, создаваемые программистами, являются подклассами **`Exception`**

`RuntimeException`, исключения этого типа определены автоматически и включают такие события, как деление на нуль, недопустимая индексация массива и т. п.

Некоторые из классов непроверяемых исключений:

- **`ArithmeticException`** – арифметические вычисления
- **`IndexOutOfBoundsException`** – индекс вне границ массива
- **`ArrayStoreException`** – попытка сохранения в массиве объекта недопустимого типа
- **`IllegalArgumentException`** – использование неверного аргумента при вызове метода
- **`NullPointerException`** – использование пустой ссылки
- **`ClassCastException`** – неверная операция преобразования типов (ошибка приведения типов)
- **`NumberFormatException`** – ошибка преобразования из строки в число

Некоторые из классов проверяемых исключений:

- **`CloneNotSupportedException`** – класс, для объекта которого вызывается клонирование, не реализует интерфейс **`Clonable`**
- **`InterruptedException`** – поток прерван другим потоком
- **`ClassNotFoundException`** – невозможно найти класс

Иерархия классов исключений

- **`Throwable`**
  - **`Exception`**
    - **`RuntimeException`**
      - Непроверяемые исключения
    - Проверяемые исключения
  - **`Error`**

Некоторые наследуемые методы:

Метод **`getMessage()`** возвращает сообщение об исключении

Метод **`getStackTrace()`** возвращает массив, содержащий трассировку стека исключения

Метод **`printStackTrace()`** отображает трассировку стека

```
try{
     int x = 6/0;
}
catch (Exception ex){

     ex.printStackTrace();
}
```

Обработка исключений

Программа может сама явно выбрасывать исключения, используя оператор **`throw`**.

Общая форма оператора **`throw`** такова:

t**`hrow ThrowableInstance`**;

Здесь **`ThrowableInstance`** должен быть объектом типа **`Throwable`** или подкласса **`Throwable`**.

Обработка исключений

Два способа получения **`Throwable`**-объекта:

- использование параметра в предложении **`catch`**;
- создание объекта с помощью операции **`new`**.

После оператора **`throw`** поток выполнения немедленно останавливается, и любые последующие операторы не выполняются.

Затем просматривается ближайший включающий блок **`try`** с целью поиска оператора **`catch`**, который соответствует типу исключения. Если соответствие отыскивается, то управление передается этому оператору. Если нет, то просматривается следующее включение оператора **`try`** и т.д.

Если соответствующий **`catch`** не найден, то программу останавливает обработчик исключений, заданный по умолчанию, и затем выводится трасса стека.

Обработка исключений

Если метод способен к порождению исключения, которое он не обрабатывает, он должен определить свое поведение так, чтобы вызывающие методы могли сами предохранять себя от данного исключения.

```
type имя_метода(список параметров) throws список исключений {
    // тело метода
}
```

Все исключения (кроме **`Error`**, **`RuntimeException`** или любых их подклассов), которые метод может выбрасывать, должны быть объявлены в предложении **`throws`**.

Если данное условие не соблюдено, то произойдет ошибка времени компиляции.

Ссылки:

https://metanit.com/java/tutorial/ Руководство по языку программирования Java (русский)

http://download.oracle.com/javase/7/docs/api/ документация по классам Java (English)

http://www.javaportal.ru (русский)

http://www.javabeginner.com (English)

Перечисления `enum`

Перечисления **enum**

- набор логически связанных констант;
- может быть объявлено вне класса.

```
enum <Название> {
   // Список элементов
}
```

Перечисления: пример

```
enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

Перечисления: пример

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
    }
}
```

Перечисления: пример

- могут использоваться в классах для хранения данных;
- в конструкторе можем присвоить, как и обычные поля класса.

```
class Schedule{
    String way;
    Day day;
    Schedule (String w, Day d){
        way = w;
        day = d;
    }
}
```

Перечисления: пример

Перечисления могут использоваться в классах для хранения данных:

```
public class Program{
  public static void main(String[] args) {
    Schedule s1 = new Schedule("Moscow", Day.MONDAY);
    Schedule s2 = new Schedule("Piter", Day.SUNDAY);
    Schedule s3 = new Schedule("Ufa", Day.MONDAY);
    ...
  }
}
```

Перечисления: пример

Можно использовать в операторе выбора:

```
switch (s1.day){
   case MONDAY:
      System.out.println ("MONDAY");
      break;
   case TUESDAY:
      System.out.println ("TUESDAY");
      break;
   ...
}
```

Методы перечислений: `values()`

Статический метод `values()` - возвращает массив всех констант перечисления:

```
public class Program{
  public static void main(String[] args) {
     Day[] days = Day.values() ;
     for (Day d : days) {
     System.out.println(d);
   }
  }
}
```

Методы перечислений: `ordinal()`

Метод `ordinal()` - порядковый номер определенной константы (нумерация начинается с 0) :

```
System.out.println(Day.FRIDAY.ordinal);  //
4
```

Конструкторы, поля и методы перечисления

```
public class Program{
  public static void main(String[] args) {
    System.out.println(Color.RED.getCode());  // #FF0000
    System.out.println(Color.GREEN.getCode());// #00FF00
}}

enum Color {
   RED("#FF0000"), BLUE("#0000FF"), GREEN("#00FF00");
   private String code;

   Color (String code){ this.code = code; }
   public String getCode() {return code;}
}
```

Обобщения (Generics)

Обобщения (Generics)

- обобщенные типы и методы
- позволяют нам уйти от жесткого определения используемых типов.

Обобщения: пример

Статический метод **values()** - возвращает массив всех констант перечисления:

```
class Account <T> {  //T – универсальный параметр
  private T id;
  private int sum;
  Account(T id, int sum) {
     this.id = id;
     this.sum = sum;
  }
  public T getId() {return id;}
  public int getSum() {return sum;}
  public void setSum(int sum) {this.sum = sum;}
}
```

Обобщения: пример

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

  Account<String> acc1 = new Account<String>("2345", 5000);
  String id1 = acc1.getId();
  System.out.println (id1);

  Account<int> acc2 = new Account<int>(2345, 5000);
  int id2 = acc1.getId();
  System.out.println (id2);
}
```

Обобщенные конструкторы

```
public class Program{
  public static void main(String[] args) {
    Account acc1 = new Account ("cid2373", 5000);
    Account acc2 = new Account (53757, 4000);
    System.out.println (acc1.getId());
    System.out.println (acc2.getId());
}}

class Account {
    private String id;
    private int sum;
    <T>Account (T id, int sum) {
        this.id = id.toString;
        this.sum = sum;
    }
    public String getId() {return id;}
    public int getSum() {return sum;}
    public void setSum(int sum) {this.sum = sum;}
}
```

Обобщенные методы

```
public class Program{
  public static void main(String[] args) {
    Printer printer = new Printer();
    String[] people = {"Tom", "Alice", "Sam", "Kate"};
    Integer[] numbers = {23, 4, 5, 2, 13, 456, 4};
    printer.<String>print(people);
    printer.<Integer>print(numbers);
}}

class Printer{
   public <T> void print(T[] items){
   for (T item: items){
      System.out.println(item);
   }
   }
}
```