Java

Введение в классы. Пакеты. Комментарии. Рекомендации

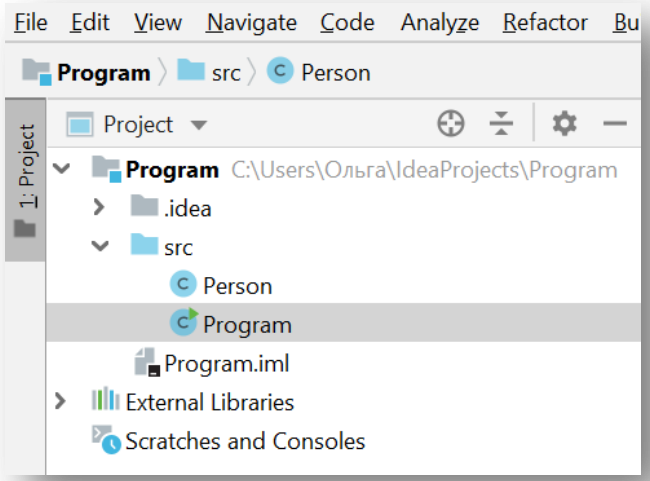
Лекция #10

Пустовалова О.Г.
доцент. каф. мат.мод.
ИММиКН ЮФУ

Содержание

ПРОЕКТ INTELLIJ IDEA

Пример проекта IntelliJ Idea



Program.java Person.java

```java
public class Program{

    public static void main(String[] args) {

        Person kate = new Person( name: "Kate",  age: 32);
        kate.displayInfo();
    }
}
```

Program.java Person.java

```java
class Person{

    String name;
    int age;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \t Age: %d \n", name, age);
    }
}
```

Пример проекта IntelliJ Idea

```
class Person{

    String name;
    int age;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }
    void displayInfo(){
        System.out.printf("Name: %s \t Age: %d \n", name, age);
    }
}
```

Пример проекта IntelliJ Idea

```
public class Program{

    public static void main(String[] args) {

        Person kate = new Person("Kate", 32);
        kate.displayInfo();
    }
}
```

КАК СОЗДАТЬ ПАКЕТ

Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами.*

Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими.

Пакеты служат в основном для обеспечения однозначности имен классов.

Допустим, двух программистов осенила блестящая идея создать класс Employee. Если оба класса будут находиться в разных пакетах, конфликт имен не возникнет.

Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете, записанное в обратном порядке (оно по определению единственное в своем роде).

Пакеты

В составе пакета можно создавать подпакеты и использовать их в разных проектах.

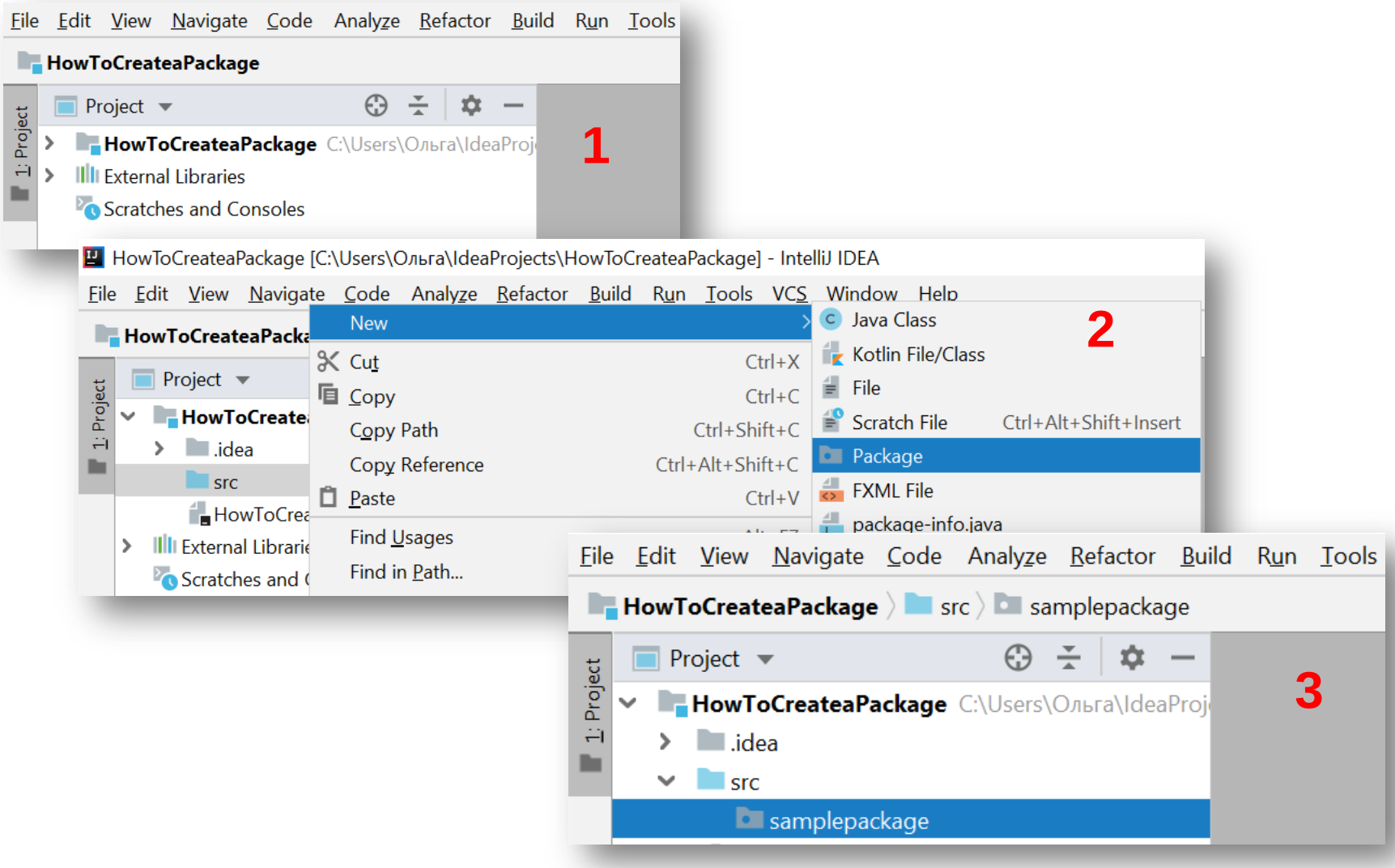
Единственная цель вложенных пакетов — гарантировать однозначность имен.

С точки зрения компилятора между вложенными пакетами отсутствует какая-либо связь.

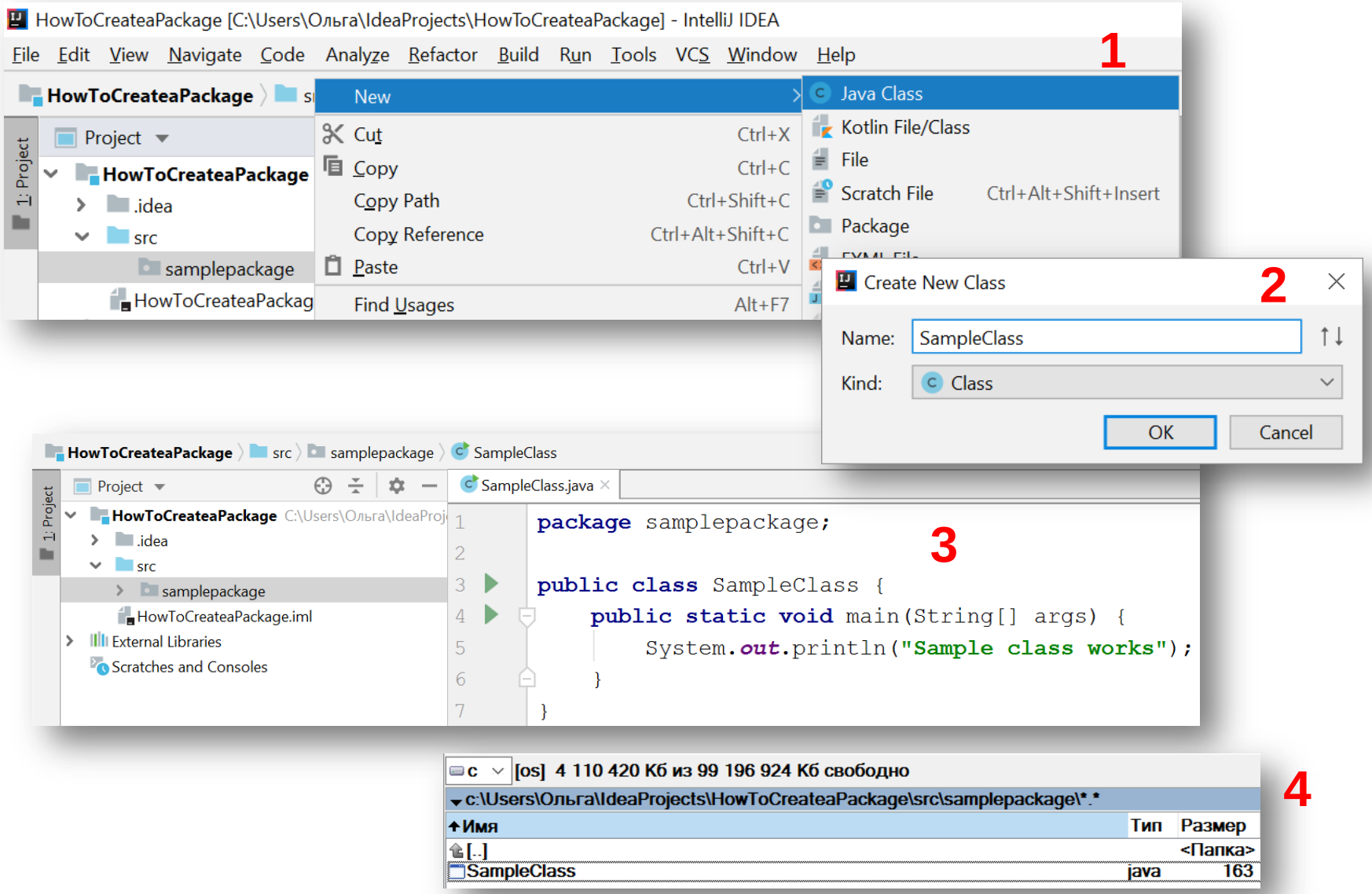
Например, пакеты **java.util** и **java.util.jar** вообще не связаны друг с другом.

Каждый из них представляет собой независимую коллекцию классов.

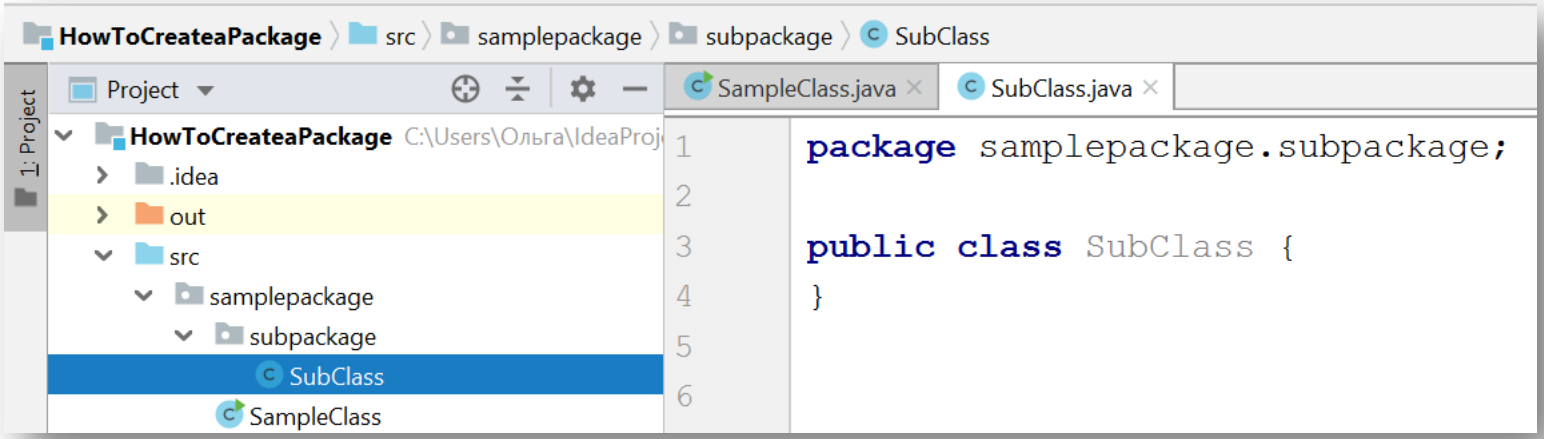
Как создать пакет

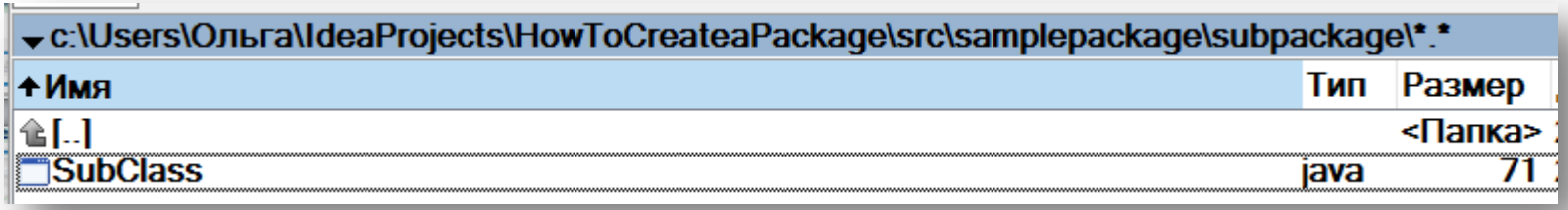


Как создать пакет. В пакете создаем класс



Как создать пакет. В пакете еще один под-пакет





При необходимости можно создать иерархию пакетов

Для чего создавать пакеты

- Для больших проектов в Java классы объединяются в пакеты.
- Пакеты позволяют организовать классы **логические наборы**.
- По умолчанию java уже имеет ряд встроенных пакетов, например, **java.lang**, **java.util**, **java.io** и т.д.
- Пакеты могут иметь вложенные пакеты.

Для чего создавать пакеты

- Организация классов в виде пакетов позволяет избежать **конфликта имен между классами**.
- Бывают ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет **гарантировать однозначность имен**.
- Названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом.

- Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

Пакеты

Пакеты служат в основном для обеспечения **однозначности** имен классов.

Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете, записанное в обратном порядке.

ru.sfedu.mmcs

Пакеты. Импорт классов

В классе могут использоваться все классы из собственного пакета и все *открытые* классы из других пакетов.

Доступ к классам из других пакетов можно получить двумя способами.

1. Перед именем каждого класса можно указать полное имя пакета:

```
java.time.LocalDate today = java.time.LocalDate.now() ;
```

2. Импортировать

```
// все классы из пакета time
import java.time.*;

// импортирование отдельного класса
import java.time.LocalDate;
```

Пакеты. Импорт классов

```
public class SampleClass {

    public static void main(String[] args) {

        java.time.LocalDate today = java.time.LocalDate.now() ;

        System.out.println(" today = "+today);
    }
}
```

today = 2019-04-24

```
package samplepackage;

// все классы из пакета time
import java.time.*;


public class SampleClass {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now() ;

        System.out.println(" today = "+today);
    }}
```

Статический импорт

Имеется форма оператора import, позволяющая импортировать не только классы, но и статические методы и поля.

Допустим, в начале исходного файла введена следующая строка кода:

```
import static java.lang.System.*;
```

Это позволит использовать статические методы и поля, определенные в классе System, не указывая имени класса:

```
out.println("Goodbye, World!"); // вместо System.out
exit(O); // вместо System.exit
```

Статические методы или поля можно также импортировать явным образом:

```
import static java.lang.System.out;
```

Но в практике программирования на Java такие выражения, как System.out или System.exit, обычно не сокращаются.

Ввод классов в пакеты

Чтобы ввести класс в пакет, следует указать имя пакета в начале исходного файла *перед* определением класса:

```
package com.horstmann.corejava;
public class Employee
        {
         ...
        }
```

Если оператор package в исходном файле не указан, то классы, описанные в этом файле, вводятся в *пакет по умолчанию.*

У пакета по умолчанию нет имени. Все рассмотренные до сих пор классы принадлежали пакету по умолчанию.

Пакеты следует размещать в подкаталоге, путь к которому соответствует полному имени пакета.

ДОКУМЕНТИРУЮЩИЕ КОММЕНТАРИИ

Документирующие комментарии

В состав JDK входит полезное инструментальное средство — утилита javadoc, составляющая документацию в формате HTML из исходных файлов.

Добавив в исходный код комментарии, начинающиеся с последовательности знаков / * * , нетрудно составить документацию, имеющую профессиональный вид.

Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему.

Утилита javadoc извлекает сведения о следующих компонентах программы.

- Пакеты.
- Классы и интерфейсы, объявленные как public.
- Методы, объявленные как public или protected.
- Поля, объявленные как public или protected.

Документирующие комментарии

Комментарии вида / * * . . . * / содержат произвольный текст, после которого следует дескриптор.

Дескриптор начинается со знака @, например @author или @paгam.

Первое предложение в тексте комментариев должно быть кратким описанием.

В самом тексте можно использовать элементы HTML-разметки, например,
 .. . — для выделения текста курсивом,
<code>. . .</code> — для форматирования текста моноширинного шрифта,
.. . — для выделения текста полужирным
и даже <img . . . > — для вставки рисунков.

Следует, однако, избегать применения заголовков (<h1> - <h6>) и горизонтальных линий (<hr>), поскольку они могут помешать нормальному форматированию документа.

Комментарии к классам

Комментарии к классу должны размещаться после операторов import, непосредственно перед определением класса.

```
/**
 *Объект класса {@code Card} имитирует игральную карту,
 *например даму червей. Карта имеет масть и ранг
 *(1=туз, 2...10, 11=валет, 12=дама, 13=король).
 */
public class Card
{
    ...
}
```

Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают.

Кроме дескрипторов общего назначения, можно использовать перечисленные ниже специальные дескрипторы:

@param - *описание переменной*

@return – *описание возвращаемого значения*

@throws - указывает на то, что метод способен генерировать исключение

Комментарии к методам. Пример

```
/**
 * Увеличивает зарплату работников
 * @param Переменная byPercent содержит величину
 * в процентах, на которую повышается зарплата
 * (например, 10 = 10%) .
 * @return Величина, на которую повышается зарплата
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

Комментарии к полям

Документировать нужно лишь открытые поля.

Они, как правило, являются статическими константами.

```
/**
 * Масть черви
 */
public static final int HEARTS = 1;
```

Комментарии общего характера

@author *имя* - может быть несколько таких дескрипторов — по одному на каждого автора

@version *текст* - раздел версии программы

@deprecated *текст* - класс, метод или переменная не рекомендуется к применению, например:

@deprecated Use <code>setVisible(true)</code> instead

@sience *текст* - *текст* означает описание версии программы, в которой впервые был внедрен данный компонент, например,

@sience version 1.7.1

@see *ссылка* - добавляет ссылку в раздел "См. также« (стр. 188)

ПЕРЕДАЧА ПАРАМЕТРОВ

Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, то ему автоматически присваивается значение по умолчанию:

- ✓ числам — нули;
- ✓ логическим переменным — логическое значение false;
- ✓ ссылкам на объект — пустое значение null.

Если поля инициализируются неявно, программа становится менее понятной.

Конструктор без аргументов

```
public Employee()
{
    name ="";
    salary = 0;
}
```

Если в классе совсем не определены конструкторы, то автоматически создается конструктор без аргументов.

В этом конструкторе всем полям экземпляра присваиваются их значения, предусмотренные по умолчанию.

Передача параметров в метод. Общий подход

Во многих языках программирования (в частности, C++ и Pascal) предусмотрены два способа передачи параметров:

- ✓ по значению
- ✓ по ссылке.

Передача параметров в метод по значению

Для того, чтобы можно было изменить переданное значение в качестве параметра нужно использовать объект.

Передача объекта в метод. Пример. Часть 1. Employee

```
class Employee {
   // к данным полям имеют доступ только методы самого класса
   private String name;
   private double salary;
   // конструктор
   public Employee(String n, double s) {
      name = n;
      salary = s;     }
   // метод
   public String getNameO() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public void raiseSalary(double byPercent) {
      double raise = salary * byPercent / 100;
      salary += raise;
   }
    public static void tripleSalary(Employee x) {
        x.raiseSalary(200);
    }
}
```

Передача объекта в метод. Пример. Часть 2. Employee

```
public class EmployeeTest {

    public static void main(String[] args) {
// заполнить массив staff тремя объектами типа Employee
        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Carl ", 10000);
        staff[1] = new Employee("Harry ", 500);
        staff[2] = new Employee("Tony ", 400);

// поднять всем работникам зарплату на 5%
        for (Employee e : staff)
            e.raiseSalary(5);

        staff[0].tripleSalary(staff[0]);
// вывести данные обо всех объектах типа Employee
        for (Employee e : staff)
                System.out.println(e.getNameO() + " - " + e.getSalary());
    }
}
```

```
Carl    - 31500.0
Harry   - 525.0
Tony    - 420.0
```

РЕКОМЕНДАЦИИ ПО РАЗРАБОТКЕ КЛАССОВ

Рекомендации по разработке классов

1. Всегда храните данные в переменных, объявленных как private.

- Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции.
- Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует

Рекомендации по разработке классов

2. Всегда инициализируйте данные.

- В языке Java локальные переменные не инициализируются, но поля в объектах инициализируются.
- Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

Рекомендации по разработке классов

3. Не употребляйте в классе слишком много простых типов.

- Несколько связанных между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, а кроме того, их легче видоизменить.

Четыре поля из класса Customer нужно объединить в новый класс Address:

```
private String street;
private String city;
private String state;
private int zip;
```

Рекомендации по разработке классов

4. Не для всех полей нужно создавать методы доступа и модификации.

- существуют поля, которые после создания объекта совсем не изменяются. К их числу относится, в частности, массив сокращенных названий штатов США в классе Address.

Рекомендации по разработке классов

5. Разбивайте на части слишком крупные классы.

- Если есть очевидная возможность разделить один сложный класс на два класса попроще, то воспользуйтесь ею.

Рекомендации по разработке классов

6. Выбирайте для классов и методов осмысленные имена, ясно указывающие на их назначение.

- Удобно принять следующие условные обозначения: имя класса должно быть именем существительным (**Order**) или именем существительным, которому предшествует имя прилагательное (**RushOrder**) или деепричастие (**BillingAddress**).

- Как правило, методы доступа должны начинаться словом **get**, представленным строчными буквами (**getSalary**), а модифицирующие методы — словом **set**, также представленным строчными буквами (**setSalary**).

Рекомендации по разработке классов

7. Отдавайте предпочтение неизменяемым классам.

- Трудность модификации состоит в том, что она может происходить параллельно, когда в нескольких потоках исполнения предпринимается одновременная попытка обновить объект. Результаты такого обновления непредсказуемы.

- Если же классы являются неизменяемыми, то их объекты можно благополучно разделять среди нескольких потоков исполнения.

Спасибо за внимание!