Классы, объекты и пакеты в Java

Алексей Владыкин

3 октября 2012

Определение ООП

- Объектно-ориентированное программирование — парадигма программирования, в которой программа строится из взаимодействующих объектов

- Объект — это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области // Гради Буч

Альтернативы ООП

- Структурное программирование
- Функциональное программирование
- Логическое программирование

Свойства объекта

- Объект является экземпляром класса
- Объект имеет внутреннее состояние
- Объект может принимать сообщения
  (в большинстве языков сообщение = вызов метода)

Возможности ООП

- Инкапсуляция
  Сокрытие деталей реализации за набором допустимых сообщений
- Наследование
  Создание производных классов, наследующих свойства базового
- Полиморфизм
  Разная обработка сообщений в разных классах

ООП в Java

- Поддержка ООП заложена в Java изначально (инкапсуляция, наследование, полиморфизм)
- В Java все является объектом, кроме примитивных типов
- Исполняемый код может находиться только в классе
- Стандартная библиотека предоставляет огромное количество классов, но можно свободно создавать свои

```
/*modifiers*/ class Example {

    /* class content: fields and methods */

}
```

Поля

```
class Example {

    /*modifiers*/ int number;
    /*modifiers*/ String text = "hello";

}
```

- Поля инициализируются значениями по умолчанию
- Модификатор `final` — значение должно быть присвоено ровно один раз к моменту завершения инициализации экземпляра

Методы

```
class Example {

    int number;

    /*modifiers*/ int getNumber() {
        return number;
    }

}
```

- Возможна перегрузка методов
  (несколько одноименных методов с разными параметрами)

Конструкторы

```
class Example {

    int number;

    /*modifiers*/ Example(int number) {
        this.number = number;
    }

}
```

- Если не объявлен ни один конструктор, автоматически создается конструктор по умолчанию (без параметров)

Деструктор

- В Java нет деструкторов, сбор мусора автоматический
- Есть метод `void finalize()`, но пользоваться им не рекомендуется
- При необходимости освободить ресурсы заводят обычный метод `void close()` или `void dispose()`

Статические поля и методы

```
class Example {

    /*modifiers*/ static final
            int DEFAULT_NUMBER = 333;

    /*modifiers*/ static int getDefaultNumber() {
        return DEFAULT_NUMBER;
    }

}
```

- Статические поля и методы относятся не к экземпляру класса, а ко всему классу

Класс Example можно использовать как

- Параметр метода
- Возвращаемое значение метода
- Локальная переменная
- Поле класса
  (того же или любого другого)

Создание экземпляра

```
Example e = null;
// e.getNumber() -> NullPointerException

e = new Example(3);
// e.getNumber() -> 3

e.number = 10;
// e.getNumber() -> 10
```

Доступ к статическим членам

```
int defaultNumber = Example.DEFAULT_NUMBER;
// defaultNumber -> 333

defaultNumber = Example.getDefaultNumber();
// defaultNumber -> 333

Example e = new Example(3);
// possible, but discouraged
defaultNumber = e.getDefaultNumber();
// defaultNumber -> 333
```

Объявление класса-наследника

```
class Derived extends Example {

    /*derived class content*/

}
```

- Java не поддерживает множественное наследование, но есть интерфейсы
- Все классы наследуют `java.lang.Object`

Конструктор класса-наследника

```
class Derived extends Example {

    Derived() {
        this(10);
    }

    Derived(int number) {
        super(number);
    }

}
```

Переопределение методов

```
class Derived extends Example {

    @Override
    int getNumber() {
        int number = super.getNumber();
        return Math.max(10, number);
    }

}
```

Полиморфизм в действии

```
Example e = new Example(3);
// e.getNumber() -> 3

e = new Derived(3);
// e.getNumber() -> 10

Derived d = (Derived) e;
// d.getNumber() -> 10
```

Оператор instanceof

- Позволяет проверить тип объекта в момент исполнения программы

```
Example e = new Example(3);
// e instanceof Object  -> true
// e instanceof Example -> true
// e instanceof Derived -> false

e = new Derived(3);
// e instanceof Object  -> true
// e instanceof Example -> true
// e instanceof Derived -> true
```

Интерфейсы

- Интерфейс определяет возможные сообщения, но не их реализацию

```
interface ExampleInterface {
    int getNumber();
}
```

- Класс может реализовывать несколько интерфейсов

```
class Example implements ExampleInterface {
    int getNumber() {
        // implementation
    }
}
```

Модификатор final

- `final class Example {...}`
  нельзя создать класс-наследник
- `final int getNumber() {...}`
  нельзя переопределить метод в дочернем классе

Модификатор abstract

- `abstract class Example {...}`
  нельзя создать экземпляр класса
- `abstract int getNumber();`
  метод без реализации (класс должен быть абстрактным)

java.lang.Object

- String toString()

java.lang.Object

- `String toString()`
- `boolean equals(Object obj)`

java.lang.Object

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`

java.lang.Object

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`
- `Class getClass()`

java.lang.Object

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`
- `Class getClass()`
- `void wait()` — три варианта
  `void notify()`
  `void notifyAll()`

java.lang.Object

- `String toString()`
- `boolean equals(Object obj)`
- `int hashCode()`
- `Class getClass()`
- `void wait()` — три варианта
  `void notify()`
  `void notifyAll()`
- `void finalize()`
  `void clone()`

Пример иерархии классов из JDK

- java.lang.Object
  - java.lang.Number
    - java.lang.Integer
    - java.lang.Double
  - java.lang.Boolean
  - java.lang.Character
  - java.lang.String
  - java.lang.AbstractStringBuilder
    - java.lang.StringBuilder
    - java.lang.StringBuffer

Зачем нужны пакеты

- Задание пространства имен,
  предотвращение коллизий имен классов

Зачем нужны пакеты

- Задание пространства имен,
  предотвращение коллизий имен классов
- Логическая группировка связанных классов

Зачем нужны пакеты

- Задание пространства имен,
  предотвращение коллизий имен классов
- Логическая группировка связанных классов
- Сокрытие деталей реализации
  за счет модификаторов доступа

Как работают пакеты

- Задание пакета для класса:

```
package ru.compscicenter.java2012;
```

Как работают пакеты

- Задание пакета для класса:
  ```
  package ru.compscicenter.java2012;
  ```
- Использование класса из пакета:
  - классы текущего пакета и пакета `java.lang` всегда видны
  - классы других пакетов доступны по полному имени с пакетом
  - можно использовать директиву `import`

Как работают пакеты

- Задание пакета для класса:
  `package ru.compscicenter.java2012;`
- Использование класса из пакета:
  - классы текущего пакета и пакета `java.lang` всегда видны
  - классы других пакетов доступны по полному имени с пакетом
  - можно использовать директиву `import`
- Класс, принадлежащий пакету, должен лежать в одноименной директории:
  `ru/compscicenter/java2012/`

Импорт

- Импорт одного класса:

```
import ru.compscicenter.java2012.ExampleClass;
```

Импорт

- Импорт одного класса:
  ```
  import ru.compscicenter.java2012.ExampleClass;
  ```
- Импорт всех классов пакета:
  ```
  import ru.compscicenter.java2012.*;
  ```

Импорт

- Импорт одного класса:
```
import ru.compscicenter.java2012.ExampleClass;
```
- Импорт всех классов пакета:
```
import ru.compscicenter.java2012.*;
```
- Импорт статических полей и методов:
```
import static java.lang.System.out;
import static java.util.Arrays.*;
```

Как работает импорт

- Директивы `import` позволяют компилятору получить полные имена всех используемых классов, полей и методов по их коротким именам

Как работает импорт

- Директивы `import` позволяют компилятору получить полные имена всех используемых классов, полей и методов по их коротким именам
- В `class`-файл попадают полные имена, подстановка содержимого не происходит

Как работает импорт

- Директивы `import` позволяют компилятору получить полные имена всех используемых классов, полей и методов по их коротким именам
- В `class`-файл попадают полные имена, подстановка содержимого не происходит
- При запуске программы все используемые классы должны присутствовать в `classpath`

- `public`
  доступ для всех
- `protected`
  доступ в пределах пакета и дочерних классов
- `private`
  доступ в пределах класса
- по умолчанию (нет ключевого слова)
  доступ в пределах пакета

- Можно объявить несколько классов в одном файле `.java`
- Только один класс может быть `public`,
  остальные должны быть с пакетным доступом
- Эффект ничем не отличается от создания отдельного `java`-файла для каждого класса

- Можно объявить класс внутри другого класса
- Такие классы имеют доступ к `private`-членам друг друга
- Экземпляр вложенного класса связан с экземпляром внешнего класса
- Если связь не нужна, вложенный класс объявляют с модификатором `static`

```
class Example {

    private int number;

    int getNumber() {
        return new Inner().getNumber();
    }

    class Inner {
        int getNumber() {
            return number;
        }
    }
}
```

Что сегодня узнали

- Что такое ООП
- Как в Java объявить класс, создать его экземпляры и работать с ними
- Как в Java реализуется инкапсуляция, наследование и полиморфизм
- Что такое пакеты и как с ними работать