НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Объектно-ориентированное программирование на языке Java

Часть 4. Использование объектов в Java

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua

Инициализация и очистка

- Многие C - ошибки обусловлены неверной инициализацией переменных .
  - Это особенно часто происходит при работе с библиотеками, когда пользователи не знают, как нужно инициализировать компонент библиотеки, или забывают это сделать .
- Завершение — очень актуальная проблема; слишком легко забыть об элементе, когда вы закончили с ним работу и его дальнейшая судьба вас не волнует.
  - В этом случае ресурсы, занимаемые элементом, не освобождаются, и в программе может возникнуть нехватка ресурсов (прежде всего памяти) .

Инициализация и очистка

- В С ++ введено понятие *конструктора* — специального метода, который вызывается при создании нового объекта.
- Конструкторы используются и в Java; к тому же в Java есть сборщик мусора, который автоматически освобождает ресурсы, когда объект перестает использоваться.

Инициализация с конструктором

- В Java создание и инициализация являются неразделимыми понятиями — одно без другого невозможно.
- Конструктор — не совсем обычный метод, так как у него отсутствует возвращаемое значение.
- Это ощутимо отличается даже от случая с возвратом значения void, когда метод ничего не возвращает, но при этом все же можно заставить его вернуть что-нибудь другое.
- Конструкторы не возвращают никогда и ничего (оператор **new** возвращает ссылку на вновь созданный объект, но сами конструкторы не имеют выходного значения).
- Если бы у них существовало возвращаемое значение и программа могла бы его выбрать, то компилятору пришлось бы как-то объяснять, что же делать с этим значением.

Пример

```
public class Cat {
    Cat(){
        System.out.print("Cat ");
    }
}
public class SimpleConstructor {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            new Cat();
    }
}
```

Перегрузка методов

- Имя конструктора предопределено именем класса, оно может быть только единственным.
- Допустим, вы создаете класс с двумя вариантами инициализации — либо стандартно, либо на основании некоторого файла. В этом случае необходимость двух конструкторов очевидна: конструктор по умолчанию и конструктор, получающий в качестве аргумента строку с именем файла.
- Оба они являются полноценными конструкторами и поэтому должны называться одинаково — именем класса.
- *Перегрузка методов* (overloading) однозначно необходима, чтобы мы могли использовать методы с одинаковыми именами, но с разными аргументами. И хотя перегрузка методов обязательна только для конструкторов, она удобна в принципе и может быть применена к любому методу .

Пример

```
public class Cat {
    String name;
    Cat(){
        name="Vasya";
        System.out.print("Cat "+ name);
    }
    Cat(String name){
        this.name=name;
        System.out.print("Cat "+ name);
    }
}
 public class SimpleConstructor {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            new Cat("Vasya"+i);
    }
}
```

Пример

```
public class Cat {
    String name;
    int age;
    Cat(){
        name="Cat";
        age =0;
          }
    Cat(String name,int i){
        this.name=name;
        this.age=i;
      }

    void print(){
        System.out.print("name:"+name+" age:"+age+"\n");
    }
    void print(String str){
        System.out.print(str+ "name:"+name+"
age:"+age+"\n");
    }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Различение перегруженных методов

- Есть простое правило: ***каждый перегруженный метод должен иметь уникальный список типов аргументов.***
- ***Метод не является перегруженным, если отличаются типы возвращаемых значений***

Ошибка:

```
void f() {     }

int f() { return 1; }
```

Конструкторы по умолчанию

Если созданный вами класс не имеет конструктора, компилятор автоматически добавит конструктор по умолчанию.

```
class Bird {}
```

```
Bird b = new Bird();
```

Конструкторы по умолчанию

Но если вы уже определили некоторый конструктор (или несколько конструкторов, с аргументами или без), компилятор *не будет* генерировать конструктор по умолчанию:

```
class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}
```

```
// Bird2 b = new Bird2(); // <-- Wrong!!!
   Bird2 b2 = new Bird2( i: 1);
   Bird2 b3 = new Bird2( d: 1.5);
```

Ключевое слово this

- Предположим, во время выполнения метода вы хотели бы получить ссылку на текущий объект. Так как эта ссылка передается компилятором *скрытно,* идентификатора для нее не существует.
- Но для решения этой задачи существует ключевое слово —**this**.
- Ключевое слово **this** может использоваться только внутри нестатического метода и предоставляет ссылку на объект, для которого был вызван метод.

```
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Вызов конструкторов из конструкторов

- Если вы пишете для класса несколько конструкторов, иногда бывает удобно вызвать один конструктор из другого, чтобы избежать дублирования кода.
- Такая операция проводится с использованием ключевого слова **this**.
- Вдобавок вызов другого конструктора должен быть первой выполняемой операцией, иначе компилятор выдаст сообщение об ошибке.

```
class Bird2 {
    Bird2(int i) { this(d:1.0*i); }
    Bird2(double d) {}
}
```

Значение ключевого слова static

- Ключевое слово **this** поможет лучше понять, что же фактически означает объявление статического (**static**) метода. У таких методов не существует ссылки **this**.
- Вы не в состоянии вызывать нестатические методы из статических (хотя обратное позволено), и статические методы можно вызывать для имени класса, без каких-либо объектов.

Очистка: финализация и уборка мусора

Важные отличия между C++ и Java:

- в C++ *объекты уничтожаются всегда* (в правильно написанной программе) ,

- В **Java** объекты удаляются уборщиком мусора не во всех случаях.

1. ***Ваши объекты могут быть и не переданы уборщику мусора.***
2. ***Уборка мусора не является уничтожением.***
3. ***Процесс уборки мусора относится только к памяти .***

- Java не позволяет создавать локальные объекты — все объекты должны быть результатом действия оператора **new**.
- Но в **Java** отсутствует аналог оператора delete, вызываемого для разрушения объекта, так как уборщик мусора и без того выполнит освобождение памяти.
- Значит, деструктор в **Java** отсутствует из-за присутствия уборщика мусора

Инициализация членов класса

Java иногда нарушает гарантии инициализации переменных перед их использованием. В случае с переменными, определенными локально, в методе эта гарантия предоставляется в форме сообщения об ошибке. Например:

```
void f() {
    int i;
    i++;  // Error - not initialized
}
```

Если примитивный тип является полем класса, то и способ обращения с ним несколько иной. Каждому примитивному полю класса гарантированно присваивается значение по умолчанию.

Явная инициализация

- Что делать, если вам понадобится придать переменной начальное значение?
- Проще все сделать это прямым присваиванием этой переменной значения в точке ее объявления в классе. (в C++ такое действие можно?)

```
public class Primitives {
    int x = 6;
    double t = 5.5;
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Порядок инициализации

- Внутри класса очередность инициализации определяется порядком следования переменных, объявленных в этом классе.
- Определения переменных могут быть разбросаны по разным определениям методов, но в любом случае переменные инициализируются перед вызовом любого метода — даже конструктора.

Порядок инициализации

```
class Window {
    Window(int marker) {
        System.out.println("Window(" + marker + ")");
    }
}
class House {
    Window w1 = new Window(1); // Before constructor
    House() {
        // Show that we're in the constructor:
        System.out.println("House()");
        w3 = new Window(33); // Reinitialize w3
    }
    Window w2 = new Window(2); // After constructor
    void f() {
        System.out.println("f()");
    }
    Window w3 = new Window(3); // At end
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Порядок инициализации

- В классе **House** определения объектов **Window** намеренно разбросаны, чтобы доказать, что все они инициализируются перед выполнением конструктора или каким-то другим действием.
- Вдобавок ссылка **w3** заново проходит инициализацию в конструкторе.
- Ссылка **w3** инициализируется дважды: перед вызовом конструктора и во время него. (Первый объект теряется, и со временем его уничтожит уборщик мусора.) Поначалу это может показаться неэффективным, но такой подход гарантирует верную инициализацию — что бы произошло, если бы в классе был определен перегруженный конструктор, который *не инициализировал* бы ссылку **w3,** а она при этом не получала бы значения по умолчанию?

```
public static void main(String[] args) {
House h = new House();
h.f(); // Показывает, что объект сконструирован
}
```

Абстрактные классы

- Абстрактный класс – это некое обобщение. Например, не существует конкретного объекта, напрямую созданного от класса Млекопитающие. Класс Млекопитающие – обобщение, абстракция. От этого класса создаются дочерние классы – отряды, и только от них уже создаются объекты.
- Абстрактный класс отвечает на вопрос "что чем является". Например, парнокопытные являются млекопитающими.

Абстрактные классы

- В Java, чтобы объявить класс абстрактным, надо в заголовке прописать слово abstract. Также обычно должен быть хотя бы один абстрактный метод. Рассмотрим пример:

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Абстрактные классы

```
package com.company;
public class Main {
    public static void main(String[] args) {
        Animal[] house = {new Cat(), new Dog(), new Dog()};
        for (int i = 0; i <house.length ; i++) {
            house[i].name="pat"+i;
        }
        for (Animal animal: house ) {
            animal.eat();      animal.voice(); }
} }
```

```
abstract class Animal {
    String name;
    void eat(){
        System.out.println(this.name+"  eats three times a day");}
    abstract void voice();
}
class Cat extends Animal {
    void voice() {
        System.out.println(this.name+"  Meow");
    } }
class Dog extends Animal {
    void voice() {
        System.out.println(this.name+"  Woof");
    } }
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Абстрактные классы

В данном случае

- Нельзя создавать объекты от класса Animal , так как в его заголовке есть слово abstract.
- Нельзя опустить реализацию метода voice() в классах Cat и Dog , поскольку они наследники абстрактного класса, в котором указанный метод объявлен абстрактным.
- Абстрактные методы не имеют тел.
- Если бы класс Animal не содержал абстрактный voice(), или метод был бы не абстрактным, то в дочерних классах можно было бы не переопределять данный метод. Таким образом, объявляя абстрактные методы, мы заставляем дочерние классы придерживаться определенного стандарта.
- Абстрактный класс может не иметь абстрактных методов. Отличие такого класса от обычного родительского только в том, что от него нельзя создавать объекты.

Абстрактные классы

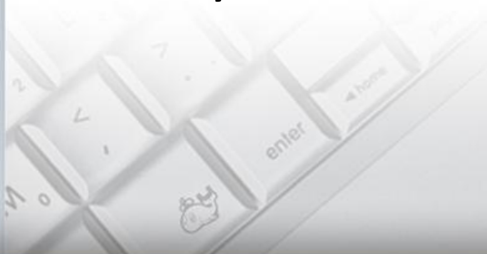
- Зачем нужны абстрактные методы, будь они в абстрактных классах, если вся их реализация ложится на плечи дочерних классов? Если вы создаете группу порожденных от сестринских классов объектов, то можете присваивать их переменным типа абстрактного класса или интерфейса и обрабатывать всю группу, например, в одном цикле. У всей группы будут одни и те же методы, хотя реализация будет зависеть от типа объекта.

Абстрактные классы

- Абстрактный класс не может использоваться непосредственно для порождения объектов. Для этого необходимо, используя этот класс как базовый, породить другой класс, в котором нужно определить все абстрактные методы. Тогда можно будет создавать объекты.
- С другой стороны не запрещено описывать переменные абстрактного класса. Просто им нужно присваивать ссылки на объекты неабстрактных классов.

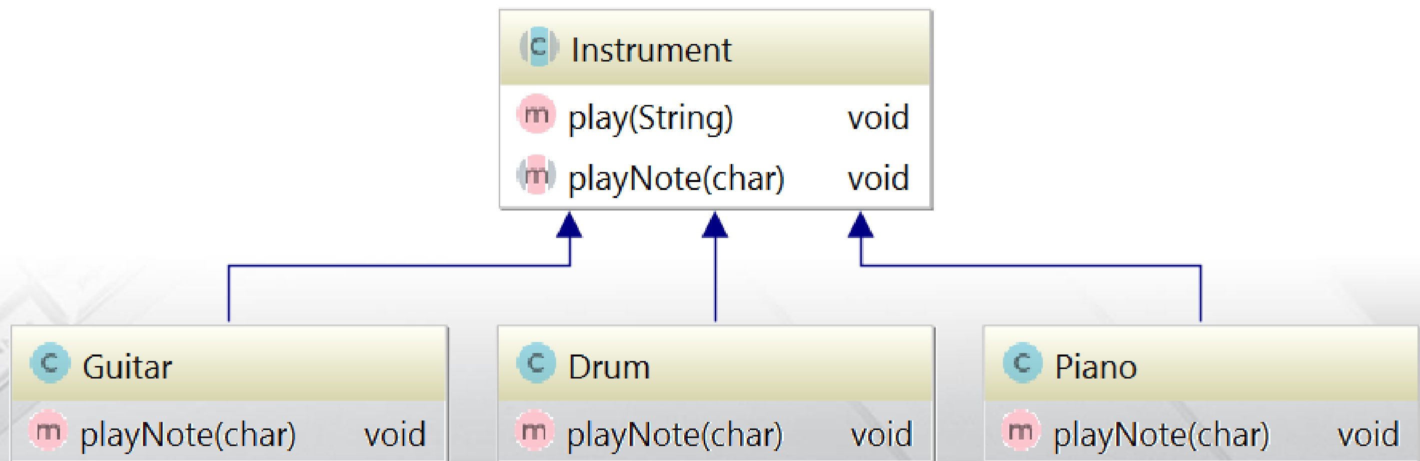
Interfaces

- Интерфейс – это в большинстве случаев определенная функциональность. Например, способность летать, распаковывать архив, парсить страницу. Интерфейс может наследоваться любым классом. Интерфейс отвечает на вопрос "у кого что есть". Например, у самолетов и птиц есть способность к полету. Несвязанные между собой ближайшим общим предком классы могут наследовать один и тот же интерфейс.
- Один класс может использовать несколько интерфейсов. Этим объясняется их популярность в Java, так как здесь отсутствием множественное наследование классов.



Interfaces

- Интерфейсы и абстрактные классы предоставляют более структурированный способ отделить интерфейс от реализации
- Если у вас есть абстрактный класс, такой как Instrument, объекты этого класса почти всегда не имеют значения. Вы создаете абстрактный класс, когда хотите манипулировать набором классов через его общий интерфейс.
- Инструмент предназначен для выражения только интерфейса, а не конкретной реализации, поэтому создание объекта Instrument не имеет смысла, и вы, вероятно, захотите помешать пользователю сделать это.



Interfaces

- Ключевое слово interface создает полностью абстрактный класс, который вообще не обеспечивает реализацию *.
- Он позволяет создателю определять имена методов, списки аргументов и типы возвращаемых данных, но без реализации тел методов *.
- Интерфейс предоставляет только форму, но не реализацию*.

* Изменения интерфейса Java 8 включают статические методы и методы по умолчанию в интерфейсах. До Java 8 у нас могли быть только объявления методов в интерфейсах. Но с Java 8 теперь могут быть стандартные методы и статические методы в интерфейсах.

Interfaces

```
interface nameOfInterface {
    //создание интерфейса
    // Поля final и static
    // абстрактные или Default методы
 }
```

Interfaces

- **Интерфейс описывает только поведение. У него нет состояния.**
- **А у абстрактного класса состояние есть: он описывает и то, и другое.**

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Interfaces

- Один класс может реализовать несколько интерфейсов:

```
class NewPower implements Printable,
                          Comparable, Serializable {
    // Тут обязаны быть реализации всех
//методов всех интерфейсов ...
    }
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Константы в интерфейсах

Кроме методов в интерфейсах могут быть определены статические константы:

```
interface Stateable{
    int OPEN = 1;
    int CLOSED = 0;
    void printState(int n);
}
```

Хотя в примере модификаторы констант не указаны, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.

Default methods

В Java 8 метод можно реализовать прямо в интерфейсе.

Метод, реализованный в интерфейсе, называется методом по умолчанию и обозначается ключевым словом **default**. Если класс реализует интерфейс, он может, но не обязан, реализовать методы, реализованные в интерфейсе. Класс наследует реализацию по умолчанию. Вот почему не обязательно модифицировать классы при изменении интерфейса, который они реализуют.

Default methods

```
interface Animal {

    void voice();

    default void drink() {
        System.out.println("I drink!");
    }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Default methods

- **Множественное наследование?**

Все усложняется, если некий класс реализует более одного (скажем, два) интерфейса, а они реализуют один и тот же самый метод по умолчанию. Какой из методов унаследует класс? Ответ — никакой. В таком случае класс должен реализовать метод самостоятельно (напрямую, либо унаследовав его от другого класса).

Ситуация аналогична, если только один интерфейс имеет метод по умолчанию, а в другом этот же метод является абстрактным. Java 8 старается быть дисциплинированной и избегать неоднозначных ситуаций. Если методы объявлены более чем в одном интерфейсе, то никакой реализации по умолчанию классом не наследуется - вы получите ошибку компиляции

Static methods

Если хотите иметь реализацию статического метода, то пишете ключевое слово **static**.

Таким образом, если вы поставили **default** или **static**, то вы обязаны предоставить реализацию метода. Статический метод интерфейса можно вызвать так же, как и статический метод класса, указав впереди имя интерфейса. Оба новых модификатора всегда предполагают модификатор **public**. Его можно указывать, а можно и не указывать. Но он всегда подразумевается. Другими словами, эти интерфейсные методы, содержащие реализацию, видны всем.

Java 9

В Java 9 можно создавать **private** и **private static** методы. И писать их реализацию. Эти методы используются другими методами интерфейса, позволяя выносить туда дублирующийся код.

Методы **private** и **default** могут обращаться к любым другим методам интерфейса. Методы **private static** и **static** только к статическим. Концептуально это ничего не меняет, но хороший стиль поддерживать проще.

Java 9

```
interface Calculatable{
    default int sum(int a, int b){
        return sumAll(a, b);
}
    default int sum(int a, int b, int c){
        return sumAll(a, b, c);
}
    private int sumAll(int... values){
        int result = 0;
        for(int n : values){ result += n; }
        return result;
    }}
```

Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable{

   void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Вложенные интерфейсы

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах. Например

```
class Printer{
    interface Printable {

        void print();
    }
}
```

Вложенные интерфейсы

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

        Calculation.multiple m= new Calt();
        System.out.println(m.mult(3,5));

    }
}
class Calt  implements Calculation.multiple{}
class Calculation {
    interface multiple{
        default int mult(int a,int b){
            return a*b;
        }
    }
}
```