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 С - 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);
    }
}
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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; }
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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 не-статического метода и предоставляет ссылку на объект, для которого был вызван метод.

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

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

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

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. (Заметьте, что в C++ такое действие)

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

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
}
```

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
}
```