8

Инкапсуляция.
Некоторые особенности методов и конструкторов

ОБСУЖДАЕМЫЕ ВОПРОСЫ

1. Инкапсуляция
2. Доступ к переменным экземпляра
3. Модификаторы доступа
4. Область видимости переменных
5. Инициализация массива
6. Перегрузка методов
7. Перегрузка конструкторов
8. Сокрытие деталей реализации

ИНКАПСУЛЯЦИЯ

Википедия определяет инкапсуляцию следующим образом:

«Инкапсуля́ция — свойство языка программирования, позволяющее объединить данные и код в объект и скрыть реализацию объекта от пользователя».

Вспомним, как мы инициализировали состояние объекта класса `Cat`:

```
cat_1.name = "Perl";
cat_1.weight = 12;
cat_1.age = 3;
```

Однако, такой подход может позволить установить следующие значения:

```
cat_1.weight = -12;
cat_1.age = -1;
```

Обеспечив `public` доступ к атрибутам, мы не можем запретить присваивание им неправильных значений. Более правильным подходом является создание `private` переменных и специальных методов `public` доступа get- и set- методы, так называемые **“гетеры”** и **“сетеры”**. В этом случае мы можем обеспечить более полный контроль.

ДОСТУП К ПЕРЕМЕННЫМ ЭКЗЕМПЛЯРА

Cat
- name : String - weight : int - age : int
<<constructors>> + Cat(); + Cat(name : String); + Cat(name : String, age : int); + Cat(name : String, weight : int, age : int); **<<methods>>** + getName(): String + setName(name : String) + getWeight(): int + setWeight(weight : int) + getAge(): int + setAge(weight : int) + printDescription()

Названия методов получения и изменения значений Value принято начинать со слов `get` и `set`, т.е. `getValue` и `setValue`

технологии JavaBean,
понентов графических
инструментов.
В настоящее время эти правила распространены широко, за пределами технологии JavaBean.

РЕАЛИЗАЦИЯ КЛАССА CAT

```
package zoostore.model;

public class Cat {

  private String name;
  private int weight;
  private int age = 1;

  public Cat( String p_name, int p_weight, int p_age ) {
      name = p_name;
      weight = p_weight;
      age = p_age;
   }
  public String getName() {
     return name;
   }
  public void setName( String p_name ) {
     name = p_name;
   }
  . . .
```

РЕАЛИЗАЦИЯ КЛАССА CAT (ПРОДОЛЖЕНИЕ)

```
 ...
 public int getWeight() {
     return weight;
   }
  public void setWeight( int p_weight ) {
     if( p_weight > 1 ) weight = p_weight;
   }
  public int getAge() {
     return age;
   }
  public void setAge( int p_age ) {
     if( p_age > 0 ) age = p_age;
   }
  public void printDescription(){
     System.out.println( "name = " + name
                         + " weight = "+ weight + " age = "+ age);
   }
}
```

РЕАЛИЗАЦИЯ КЛАССА

Некоторые атрибуты, например те, которые мы запрещаем изменять, могут иметь только «геттеры» и не иметь «сеттеров». Некоторые атрибуты могут иметь только «сеттеры». Рассмотрим соглашения об именах методов, которые приняты в JavaBean.

Если переменная называется `var`, а ее тип `T`, то методы должны быть следующими:

```
T getVar()
setVar( T val )
T[] getArr()
setArr( T[] val )
```

Если переменная имеет тип `boolean`, то желательно вместо «геттера» иметь метод:

```
boolean isVar()
```

В названии метода первая буква имени переменной вводится на верхнем регистре. В тех случаях, когда минимум две первых буквы имени переменной введены на верхнем регистре, имя переменной в названии метода никак не трансформируется. Например, если имя переменной `String URL;` то сигнатуры методов будут выглядеть так:

```
String getURL()
setURL( String val )
```

ИНКАПСУЛЯЦИЯ

При создании класса следует скрывать от потребителей действия, которые являются локальными для класса, например, служебные методы. Разработчику, использующему наш класс, нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока этот класс соблюдает свои обязательства- интерфейс класса. Здесь уместно процитировать замечание из одного трактата: *"Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части"*. Создавая типы данных в виде классов, мы предполагаем, что они будут использоваться многократно. И в этой связи становится ясно, что глобально видимыми должны быть только необходимые аспекты типа, тогда изменение внутренней реализации не потребует изменения внешних частей.
Процитируем еще один источник: *«Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации»*. То есть инкапсуляция служит цели сокрытия реализации. Мы делаем глобально видимыми только те части класса, которые ДОЛЖНЫ быть глобально видимыми.

МОДИФИКАТОРЫ ДОСТУПА

Следующая таблица, описывает области видимости для разных компонентов классов с разными модификаторами доступа.

Модификатор доступа	Тот же самый класс	Тот же самый пакет	Субкласс (класс потомок, возможно из другого пакета)	Внешние классы из других пакетов
`private`	Да	Нет	Нет	Нет
`опущен (default)`	Да	Да	Нет	Нет
`protected`	Да	Да	Да	
`public`	Да	Да	Да	Да

МОДИФИКАТОРЫ ДОСТУПА (ПРОДОЛЖЕНИЕ)

А что, если вы вообще не определяете спецификатор доступа? Доступ по умолчанию не имеет ключевого слова, но обычно называется дружественным - “**friendly**”. Это значит, что все другие классы в том же пакете имеют доступ к дружественным членам, но для классов за пределами этого пакета, члены являются приватными (`private`). Говорят, что дружественные элементы имеют доступ на уровне пакета.

Однако, старайтесь не опускать модификатор доступа, так как это стимулирует прямое обращение к внутренним конструкциям класса из других классов этого же пакета. Такой подход допустим только в редких случаях библиотек специального назначения, так как тесно привязывает классы пакета друг к другу.

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ

Параметры методов подобны локальным переменным в том, что создаются в стеке при вызове методов. Инициализация параметров выполняется передачей фактических значений при вызове метода. Локальные параметры создаются в стеке при работе метода, в котором они определены, и должны быть инициализированы явно. Жизнь параметров и локальных переменных продолжается до возврата из метода. Жизнь переменных экземпляра совпадает с жизненным циклом объекта.

- Локальные переменные и параметры, имена которых совпадают с именами переменных экземпляра и класса перекрывают последние.
- Для того, чтобы в методе обратиться к перекрытой переменной экземпляра используется ссылка **this** на данный экземпляр.

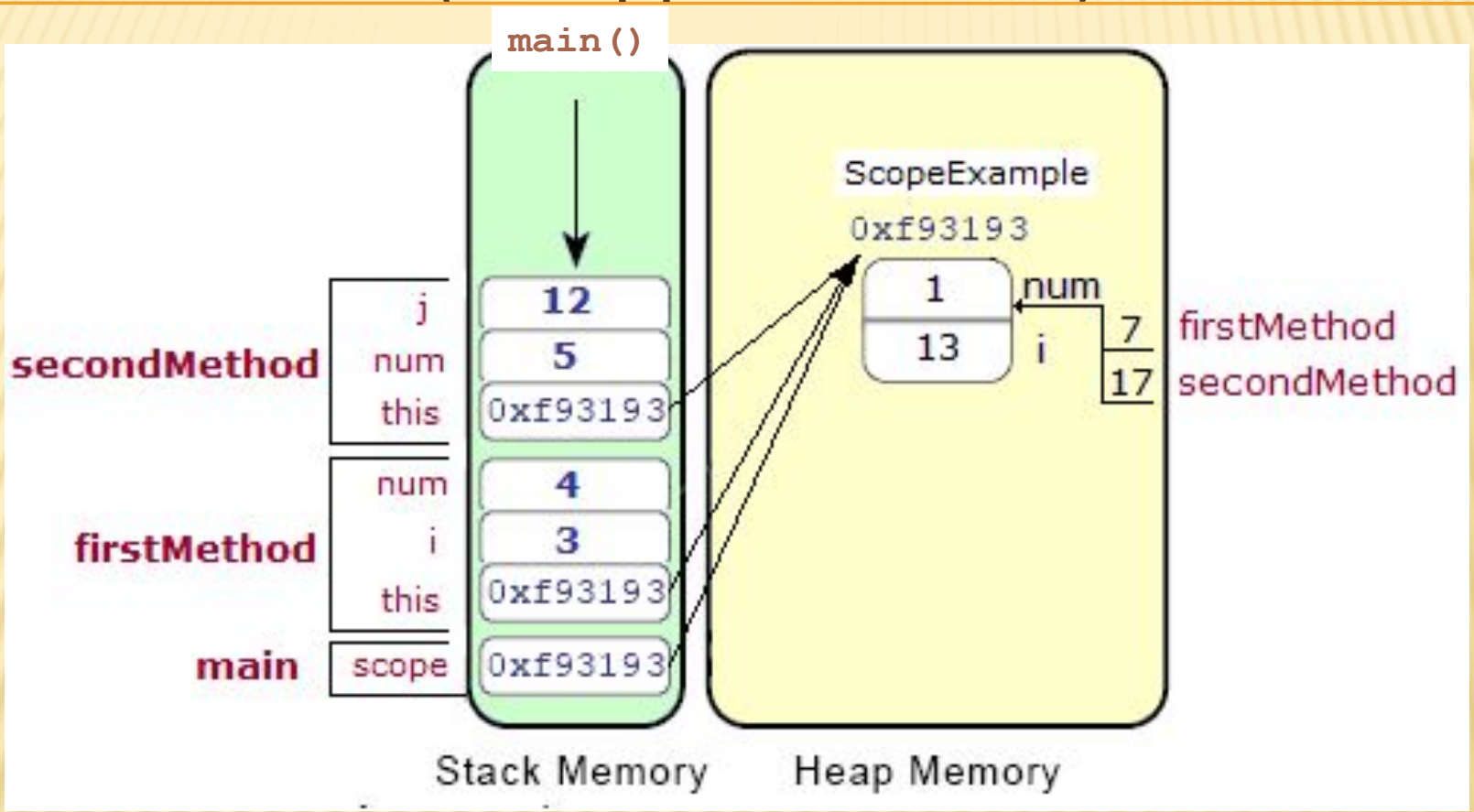
ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ (ПРОДОЛЖЕНИЕ)

```
public class ScopeExample {
 private int num = 1;
 private int i = 13;
 private String name;

 pulic String getName() { return name; }
 public void setName(String name) {
    this.name = name; // здесь this обязательно
 }

 public void firstMethod() {
   int i = 3, num = 4;
   this.num = i + num;
   secondMethod( 5 );
 }

 public void secondMethod( int num ) {
   int j = 12;
   this.num = num + j;
 }
}
```

Перекрыты переменные `i` и `num`.

```
// использование:
ScopeExample scope = new ScopeExample();
scope.firstMethod();
```

ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ (ПРОДОЛЖЕНИЕ)



Ссылка `this` не только позволяет обратиться к перекрытым переменным экземпляра, что, кстати, делается не слишком часто, так как давать одинаковые имена локальным переменным и переменным экземпляра настоятельно не рекомендуется, но также позволяет одному объекту передать ссылку на себя другому объекту.

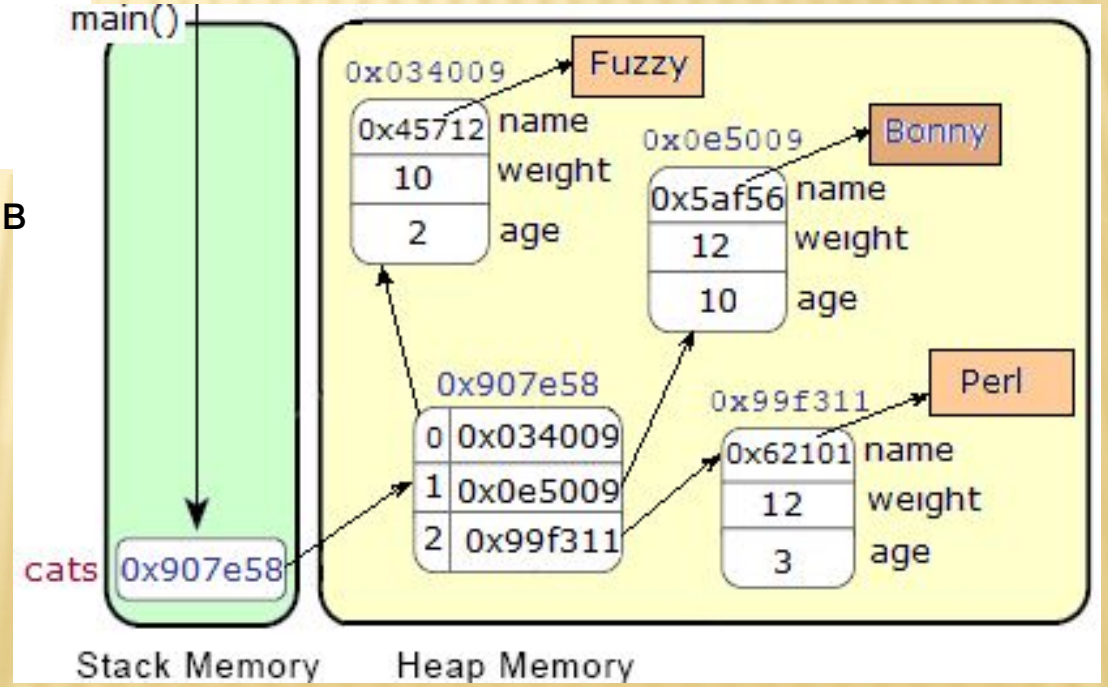
ИНИЦИАЛИЗАЦИЯ МАССИВА

Теперь, когда у нас имеется конструктор для инициализации кошек, создадим массив объектов и посмотрим, как его можно проинициализировать, и чем массив ссылок отличается от массива примитивов.

```
Cat[] cats = new Cat[3];
cats[0] = new Cat( "Perl", 12, 3 );
cats[1] = new Cat( "Bonny", 12, 10 );
cats[2] = new Cat( "Fuzzy", 10, 2 );
for( Cat c : cats ) {
            c.printDescription();
 }
```

Также как и массив примитивов, массив ссылок можно создать и сразу проинициализировать:

```
Cat[] cats = {
      new Cat( "Perl", 12, 3 ),
      new Cat( "Bonny", 12, 10 ),
      new Cat( "Fuzzy", 10, 2 );
  }
```



ПЕРЕГРУЗКА МЕТОДОВ

Название метода подсказывает нам, что этот метод делает. Однако бывают ситуации, когда одинаковые действия надо выполнять для параметров различного типа или для разного количества параметров. В этом случае мы можем написать методы, имеющие одинаковые названия, но разное типы формальных параметров или/и разное их количество.

```
public int avg( int a, int b ) {
   return (a + b)/2;
 }
public float avg( float a, float b ) {
   return (a + b)/2;
 }
```

Перегрузка может осуществляться и по разному количеству параметров.

```
public float avg( float a, float b, float c ) {
   return (a + b + c)/3;
 }
```

ПЕРЕГРУЗКА МЕТОДОВ (ПРОДОЛЖЕНИЕ)

То, какой метод нужно на самом деле вызывать, система определяет по типам и количеству фактических параметров. При этом тип возвращаемого значения может отличаться, а может и совпадать у разных методов. *Перегрузка по типу возвращаемого значения не производится.*

Если перегруженных методов может быть много, и только по количеству однотипных параметров, можно воспользоваться появившейся в версии 1.5 возможностью использования методов с переменным количеством параметров . . . (троеточие). Например, наш гипотетический метод рассчитывающий среднее значение может иметь четыре, пять и большее количество параметров. Воспользовавшись в таком случае возможностью передачи переменного количества параметров, мы можем написать один метод решающий эту задачу.

ПЕРЕГРУЗКА МЕТОДОВ (ПРОДОЛЖЕНИЕ)

```
public float avg( float ... numbers ) {
   float sum = 0;
   for( float n : numbers ) {
          sum += n;
     }
   return( sum / numbers.length );
 }
```

Проанализировав текст, вы уже очевидно поняли, что в этом случае мы принимаем на входе массив, который в дальнейшем и обрабатываем.

Вызывать этот метод можно разными способами:

```
float a = avg( 2.0f, 5.45f );
float a = avg( 2.0f, 5.45f, 4.0f );
float a = avg( 2.0f, 5.45f, 4.0f, 3.4f );
```

ПЕРЕГРУЗКА КОНСТРУКТОРОВ

Конструкторы также могут быть перегруженными.

```
public Cat() {
      this( "Anonim", 10, 1 );
   }
  public Cat( String p_name, int p_weight ) {
      name = p_name;
      weight = p_weight;
   }
  public Cat( String p_name, int p_weight, int p_age ) {
      name = p_name;
      weight = p_weight;
      age = p_age;
   }
```

В данном примере ключевое слово `this` позволяет вызвать другой конструктор из данного. Подобный вызов может быть только первым оператором в конструкторе, то есть выше него не может быть других операторов. После такого вызова другие операторы допустимы.

СОКРЫТИЕ ДЕТАЛЕЙ РЕАЛИЗАЦИИ

Хорошо спроектированный класс, компонент или модуль скрывает детали реализации, четко разделив открытый API от реализации. Компоненты взаимодействуют друг с другом только через свои открытые программные интерфейсы (API), они не знают, не хотят знать и не должны знать, – какая работа выполняется внутри других компонентов. *Эта концепция называется Encapsulation - сокрытием реализации* и представляет собой один из фундаментальных принципов разработки программного обеспечения.

Инкапсуляция важна по многим причинам, одна из которых в том, что этот механизм эффективно изолирует компоненты, составляющие систему, обеспечивая их *слабое связывание* и позволяет разрабатывать, тестировать и обновлять их независимо друг от друга.

Главный принцип здесь в том, чтобы сделать видимыми только те атрибуты и методы, которые должны быть видны, *минимизируя API*.

ИТОГИ

В теме рассмотрены:

- Инкапсуляция
- Доступ к переменным экземпляра
- Модификаторы доступа
- Область видимости переменных
- Инициализация массива
- Перегрузка методов
- Перегрузка конструкторов
- Сокрытие деталей реализации