Статические элементы

Если требуется член класса, которым можно было бы пользоваться независимо от какого-либо объекта этого класса – используется ключевое слово **static.**

К элементам **static** можно обращаться до создания объекта этого класса. Копии **static** переменной не создаются для каждого объекта, все объекты совместно используют одну и ту же **static**-переменную.

Могут быть static переменные и static методы.

Ограничения:

1) static методы могут вызывать только другие static методы;
2) static методы должны обращаться только к static переменным;
3) static элементы не могут ссылаться на this, super.

Статические элементы

При организации вычислений с целью инициализировать статические переменные можно объявить статический блок, который выполняется один раз при загрузке класса.

Внутри статического метода нельзя обращаться к нестатическим переменным объекта.

Для вызова static метода в текущем классе:

Имя_метода (…);

Для вызова static метода в другом классе:

Имя_класса.Имя_метода(n…);
Имя_класса.Имя_переменной;

import java.lang.Math;

Math.sin(x);

Math.PI;

```
class Static {
    static int a=3;
    static int b;

    static void meth (int x){
            System.out.println ("x="+x);
            System.out.println  ("a="+a);
            System.out.println ("b="+b);   }

    static {
        System.out.println("Статический блок
                                        инициализирован");
        b=a*4;}

   public static void main (String args[]) {
                        meth (42);        }              }
```

Статический блок инициализирован
x=42
a=3
b=12

(сначала a=3, статический блок инициализирован, b=12, затем вызов main(…), затем meth(…))

Метод finalize()

Иногда, при уничтожении (сборка мусора) объект обязательно должен выполнить какое-то действие (например, закрыть файл).

Тогда необходимо в классе определить метод

```
protected void finalaze(){
    //действие, которое необходимо
    //выполнить перед  удалением
    // объекта
  }
```

Перед освобождением ресурсов java вызывает **finalaze()** к удаляемому объекту, но программа не должна зависеть от этого метода, так как неизвестно, будет ли он вызван.

protected - спецификатор доступа, который запрещает доступ к методу кодам, определенным вне этого класса.

Вложенные и внутренние классы

Можно определить один класс внутри другого. Такие классы называются вложенными.

Область видимости вложенного класса ограничивается областью видимости включающего класса.

Если класс **B** определен внутри класса **A**, то класс **B** виден внутри **A**, но не вне его.

Вложенный класс имеет доступ ко всем членам класса, в котором он вложен (включая private).

Включающий класс не имеет доступ к членам вложенного класса.

Существует 2 типа вложенных классов:

1) **Статический** (static class…)
   Он должен обращаться к членам включающего класса через объект, напрямую не может. Т.е. требуется создание объекта включающего класса.

2) **Нестатический**
   Внутренний – нестатический вложенный класс, имеющий доступ ко всем переменным и методам включающего класса, и возможность обращаться к ним напрямую.

Внутренние классы

```
class Outer {
   int outer_x=100;

   class Inner{
         int y=10;
         void display (){
         System.out.println ("В методе display: outer_x="+outer_x);  }   }

   void test() {
               Inner inner=new Inner();
                     inner.display();              }

    void showy ( ) {
        System.out.println(y); // ошибка!!!  y не доступен в Outer! }
}

class   InnerDemo {
       public static void main (String args[]){
                     Outer outer = new Outer();
                     outer.test ();                        }   }
```

Внутренние классы можно определять внутри области определения любого блока. Например, вложенный класс можно определить внутри блока, определенного методом, или внутри тела цикла for.

```
class Outer {
  int outer_x=100;

  void test(){
    for(int i=0; i<10;i++){
      class Inner {
        void display(){
          System.out.println(“display: outer_x=“+outer_x);  }  }

    Inner inner = new Inner();
              inner.display();
                                  }
} }
```

Абстракция.Иерархия.

Наследование

Используя наследование, можно создавать общий класс, который определяет характеристики, общие для набора связанных элементов.

Наследуя данный класс, можно добавлять к новым классам уникальные характеристики.

Наследуемый класс – это **суперкласс**,
наследующий класс – это **подкласс**.

extends – ключевое слово для наследования

Общая форма:

```
class ИмяКласса extends ИмяСуперкласса {
                    //тело класса
                    }
```

В Java для каждого создаваемого класса можно указывать только один суперкласс.

Подкласс наследуют все члены класса.

Подкласс имеет доступ ко всем общедоступным (переменным) членам своего суперкласса.

Подкласс не имеет доступа к членам суперкласса, объявленными private (приватный член класса не доступен любому коду вне класса), но наследует их.

```
class Box {              // класс Коробка
    double width;    // ширина
    double height;   // высота
    double depth;    // глубина

    Box( ) { width=height=depth=0; } // констр-р без парам-ов

    Box (double w, double h, double d){// констр-р с парам-ми
                                width=w;   height=h;   depth=d;   }
    double volume() {   // метод для вычисления объема коробки
        return width*height*depth; }
}

class  BoxWeight   extends   Box {     // класс- наследник
    double weight;                 // добавляет свою переменную ВЕС

    BoxWeight (double w, double h, double d, double m){
            width=w;   height=h;   depth=d;   weight=m;   }
 }
```

Наследование

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

        BoxWeight  M1 = new  BoxWeight(10,20,30,40);
        BoxWeight  M2 = new  BoxWeight(1,2,3,4);

       double vol = M1.volume();
                vol =M2.volume();              }   }
```

Переменная суперкласса может ссылаться на объект подкласса.

```
BoxWeight   M  =  new BoxWeight (3,4,5,6);
          Box  N   =  new Box( );
      double vol  = M.volume( );
                   N = M;
                 vol = N.volume( );
              System.out.println("Вес N равен" + N.weight); // ошибка!!!
```

Доступ может осуществляться только к тем частям объекта, которые определяет суперкласс. Нельзя обратиться **N.weight,** т.к. суперклассу не известно, какие переменные и методы добавляет подкласс.

```
class A {
    int i;              // общедоступная по умолчанию
    private int j;    // приватная для A
    void set_ij (int x, int y) {
                        i=x;
                        j = y; }
}

class  B  extends   A{  // класс B наследник класса A
  int total;
  void sum() {
       total = i + j;    // ОШИБКА, j в этом классе недоступна   }
}
```

Ключевое слово super

Конструктор BoxWeight явно инициализирует поля width, height и depth класса Box – это не эффективно, ведет к дублированию кода и предполагается, что эти члены public, так как BoxWeight к ним обращается.

В ряде случаев необходимо создавать суперкласс, в котором все члены private. В этом случае подкласс не сможет явно инициализировать переменные суперкласса.

Ключевое слово **super** имеет 2 формы:

1) вызов конструктора суперкласса
   super (список_аргументов); // аргументы конструктора
   // суперкласса
2) для обращения к члену суперкласса, скрытого членом подкласса.

Ключевое слово super

Оператор **super()** всегда должен быть первым выполняемым внутри конструктора подкласса.

```
class Box {
        private double width;
        private double height;
        private double depth;

        Box (double w, double h, double d){
                        width=w;   height=h;   depth=d; }

        double volume ( ) { return  width*height*depth;  }
             }

class  BoxWeight  extends  Box {
        double weight;

        BoxWeight (double w, double h, double d, double m) {
                super (w, h, d); // вызов конструктора суперкласса
                weight=m;                                          }  }
```

Ключевое слово super

Так как конструкторы суперкласса могут быть перегруженными, то для вызова соответствующего конструктора нужно указывать аргументы.

```
BoxWeight (BoxWeight  ob) {
        super(ob);
        weight = ob.weight;  }
```

Оператор **super( )** выполняет передачу объекта типа BoxWeight, но срабатывает конструктор Box (Box ob), т.к. переменная суперкласса может ссылаться на объект подкласса.

super всегда ссылается на ближайший суперкласс в иерархии.

Ключевое слово super

Второй способ применения super используется, когда имена членов подкласса скрывают члены суперкласса с такими же именами.

super.член_класса

```
class   A {   int i;  }

class B extends A {
            int i;
            B (int a, int b) {   super.i =a;    i=b;   }
            void show ( ) {
                  System.out.println ( super.i );     // из A
                   System.out.println ( i ) ;          // из B     }  }

class UseSuper {
     public static void main (String args[]) {
                B ob = new B(1, 2);
                   ob.show( );                        }   }
```

Также super может использоваться для вызова метода, скрываемого подклассом.

Порядок вызова конструкторов

Суперкласс ничего не знает о порядке наследования, поэтому любая инициализация, которую ему необходимо выполнить, независима и возможно обязательна для любой инициализации, выполняемой подклассом.

В иерархии классов конструкторы вызываются в порядке подчиненности классов – от суперкласса к подклассу.

Если super( ) не используется, то вызывается конструктор каждого суперкласса, заданного по умолчанию или не содержащий параметров.

```
class A{
      A( ){
         System.out.println("Внутри конструктора A");  } }

class B extends A{
       B( ){
          System.out.println("Внутри конструктора B");  }  }

class C extends B{
        C( ){
           System.out.println("Внутри конструктора C");  } }

class Calling {
       public static void main (String args[ ]) {
                    C  c = new  C( );                    }  }
```

Внутри конструктора A
Внутри конструктора B
Внутри конструктора C

Переопределение методов

Если в иерархии классов имя и сигнатура типов методов подкласса совпадает с именем и сигнатурой метода суперкласса, то говорят, что метод подкласса **переопределяет** метод суперкласса.

Когда переопределенный метод вызывается из подкласса, он всегда будет ссылаться на версию метода, определенную в подклассе (версия в суперклассе будет скрыта).

Переопределение метода выполняется только в том случае, если имена и сигнатуры полностью совпадают. Иначе методы являются перегруженными.

Сигнатура – это тип метода, количество и типы параметров.

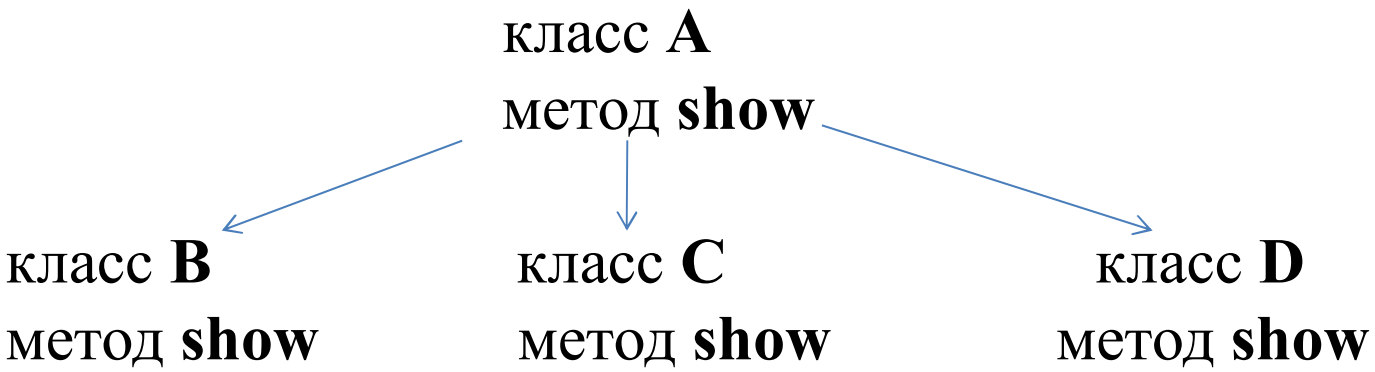
```
class A{
    int  i, j;
    A (int a, int b) { i=a;  j=b; }
    void show( ) {
                System.out.println (“i и j:” +i+”  ”+j);  }   }

class B  extends  A {
    int k;
    B (int a, int b, int c) {  super (a,b);    k=c;  }
    void show() {   super.show( );
                    System.out.println(“k:”+k);  }  }

class Override {
        public static void main (String args[]){
                B ob = new B(1,2,3);
                   ob.show( );                    }  }
```

Динамическая диспетчеризация методов

- это механизм, посредством которого решение на вызов переопределенного метода принимается во время выполнения, а не во время компиляции.

Переменная суперкласса может ссылаться на объект подкласса, поэтому когда происходит вызов переопределенного метода через ссылку на суперкласс, то JAVA выбирает нужную версию в зависимости от типа объекта (а не ссылки) в момент вызова. Этот выбор осуществляется **во время выполнения**.

При ссылке на различные типы объектов будут вызываться различные версии переопределенного метода.




```
A   a = new  A( );   a.show();  // show из A
A   a = new  B ( );   a.show();  // show из B
A   a = new  C ( );   a.show();  // show из C
```

Динамическая диспетчеризация методов

```
class  A {
   void callme() {
                System.out.println ("Внутри A");  } }
class  B extends A{
    void callme() {
                System.out.println ("Внутри B");  } }
class  C extends A {
      void callme() {
                System.out.println ("Внутри C"); }  }
class  ABC {
   public static void main (String args[]){
     A a=new A( );      B b=new B( );      C c=new C( );
     A  r;                  //ссылочная переменная суперкласса
     r=a;     r.callme();                    // один интерфейс
     r=b;     r.callme();                    // множество методов
     r=c;     r.callme();  }   }             //это полиморфизм
```

Абстрактные классы

Иногда требуется определить суперкласс, определяющий только обобщенную форму, которую будут использовать все подклассы, добавляя свои детали.

Абстрактный класс - структура без реализации методов, которая содержит только объявления (имена) методов. Каждый подкласс будет реализовывать эти методы по своему.

Например, фигура содержит размеры и функцию вычисления площади, но не зная фигуры (треугольник, квадрат, ромб и т.д.) площадь не вычислить.

Абстрактный метод - метод, для которого нет реализации (тела) в суперклассе, но он должен быть обязательно переопределен в подклассе.

Абстрактные классы

Ключевое слово **abstract**

Обозначается **abstract** *тип имя_метода(список параметров)*;

Любой класс, который содержит один или несколько абстрактных методов, должен быть объявлен абстрактным **abstract**.

Нельзя создать объект абстрактного класс (new нельзя),
нельзя объявлять абстрактные конструкторы и статические методы.

Любой подкласс абстрактного класса должен реализовывать все абстрактные методы суперкласса или должен быть сам объявлен абстрактным.

Если методы не нужны , то оставлять пустое тело {}.

```
abstract class A{
    abstract void callme( );
    void callmetoo(){
            System.out.println  (“Метод реализован”);   } }

class  B extends  A {
     void callme ( ){
          System.out.println  (“Реализован в B”);   } }

class AB{
  public static void  main (String args[ ]){
       B b =new B( );
          b.callme( );
          b.callmetoo( ); }  }
```

Абстрактный класс может содержать реализованные методы. Абстрактные классы не могут использоваться для создания объектов, но их можно использовать для создания ссылок на объект подкласса.

Ключевое слово **final**

1) Константа: **final** (окончательная) – после инициализации нельзя менять. Переменная типа **final** должна быть инициализирована во время объявления.
   final int A = 1;

2) Запрет на переопределение метода (методы, объявленные **final** не могут быть переопределены).

```
class A {
   final void meth( ) {
          System.out.println (“Это метод final”);  }  }

class B extends A {
        void meth ( ) {
               System.out.println (“Нельзя!”);  }  }
```

3) Отмена наследования данного класса (если хотим, чтобы нельзя было наследовать класс). **final class AA{….}**
   Класс АА нельзя наследовать.