Java.SE.02

OBJECT-ORIENTED PROGRAMMING IN JAVA

Author: **Ihar Blinou**
Oracle Certified Java Instructor
ihar_blinou@epam.com

1. Причины возникновение ООП
2. Классы и объекты
3. Три кита ООП
4. Наследование
5. Интерфейсы
6. Введение в Design Patterns

ПРИЧИНЫ ВОЗНИКНОВЕНИЯ ООП

Причины возникновения ООП

Классификация языков (одна из ...)

По одной из классификаций языки программирования делятся на:

- **директивные** (directive), называемые также процедурными (procedural) или императивными (imperative),
- **декларативные** (declarative) языки,
- **объектно-ориентированные** (object-oriented).

Причины возникновения ООП

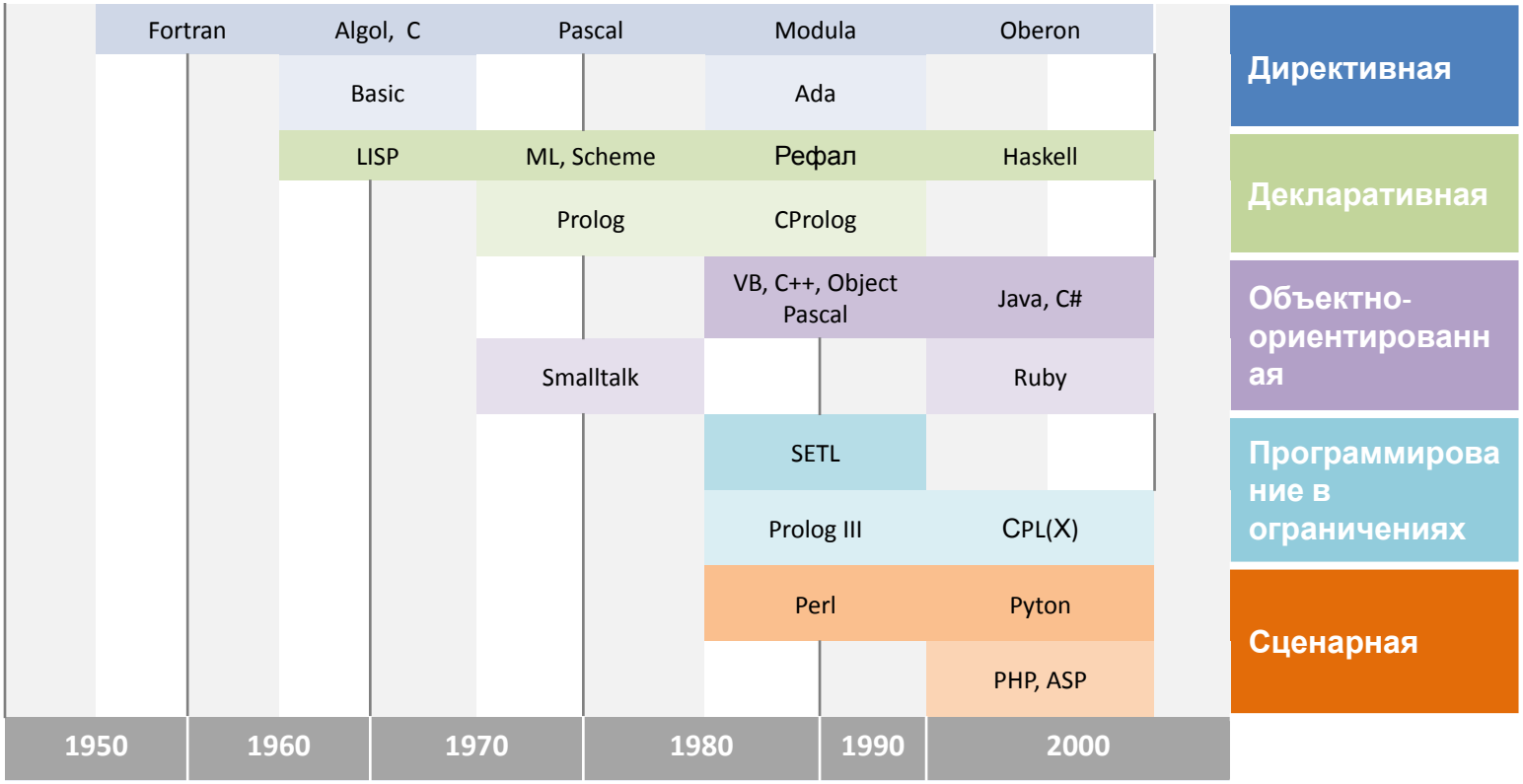
Декларативная и директивная парадигмы

Поговорим о различии между первыми двумя парадигмами.

Главное заключается в следующем: декларативная программа *заявляет* (декларирует), **что** должно быть достигнуто в качестве цели, а директивная *предписывает*, **как** ее достичь.

Причины возникновения ООП

Развитие языков и парадигм программирования

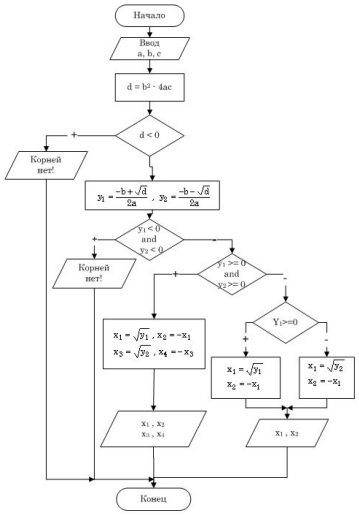


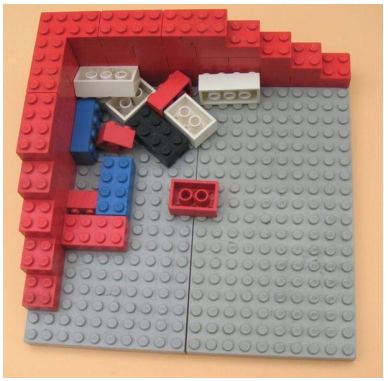
Причины возникновения ООП

Причины возникновения и задачи ООП

```
draw_window:
    mov  eax,12
    mov  ebx,1
    int  0x40
    mov  eax,0
    mov  ebx,100*65536+300
    mov  ecx,100*65536+120
    mov  edx,0x001111cc
    mov  esi,0x8099bbff
    mov  edi,0x00ffffff
    int  0x40
    mov  eax,4
    mov  ebx,8*65536+8
    mov  ecx,0x00ffffff
    mov  edx,labelt
    mov  esi,labellen-labelt
    int  0x40
    mov  eax,8
    mov  ebx,(300-19)*65536+12
    mov  ecx,5*65536+12
    mov  edx,1
    mov  esi,0x5599cc
    int  0x40
    mov  ebx,25*65536+35
    mov  ecx,0xffffff
    mov  edx,text
    mov  esi,40
```

```
function show_category() {
        global $DB, $SITE;
        $num_cuts = 0;
        if(!isset($SITE->input['cmscat'])) {
            // Не указан номер нужной категории
            $SITE->error('Невозможно показать категорию, потому что не указан необходимый параметр');
            return 1;
            }
                $cat_name = $DB->fast_query('SELECT name FROM scl_cms_categories WHERE id='.$SITE->input['cmscat'].';', 'name');
        if($cat_name===0){
            // Категория нет :)
            $SITE->error('Невозможно отобразить категорию, потому что она отстутствует');
            return 1;
            }
    $SITE->begin_output('Категория статей "'.stripslashes($cat_name).'"');

                    $child = $SITE->input['cmscat'];
      $SITE->output .= '<div class="cmsstory" align="right">';
      while($child!=0)
        $SITE->output .= '<a href="index.php?mod=cms&act=show_cat&cmscat='.$child.'">'.$this->get_parent($child).'</a> &bull; ';
      $SITE->output .= '<a href="index.php">Чего-то там</a></div>';

      $cats = array();
      $cats = $this->get_categories($cats, $SITE->input['cmscat']);
      $SITE->output.='<div class="catlinkshead">Подкатегории:</div><div class="catlinks">';

      foreach($cats as $c_id => $c_val) {
        $SITE->output.='<a href="index.php?mod=cms&act=show_cat&cmscat='.$c_id.'">'.$c_val.'</a><br>';
      }
```





Причины возникновения ООП

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. ... ООП использует в качестве базовых элементов объекты, а не алгоритмы.

Хорошо и ... не понятно. Вот и давайте разбираться постепенно.

КЛАССЫ И ОБЪЕКТЫ

Классы и объекты

Классом называется описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы определяют структуру и поведение некоторого набора элементов предметной области, для которой разрабатывается программная модель.

Каждый класс имеет свое имя, отличающее его от других классов, и относится к определенному пакету. Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются программные файлы, содержащие реализацию классов.

Классы позволяют разбить поведение сложных систем на простое взаимодействие взаимосвязанных объектов.

Классы и объекты

Объект. Понятие "объект" не имеет в ООП канонического определения

Объект- *это осязаемая сущность, которая четко проявляет свое поведение*.

Объект состоит из следующих трех частей:

- имя объекта;
- состояние (переменные состояния);
- методы (операции)

Объект ООП *- это совокупность переменных состояния и связанных с ними методов(операций). Эти методы определяют как объект взаимодействует с окружающим миром.*

Классы и объекты

Класс / Экземпляр Класса

Объект совокупность (разнотипных) данных (полей объекта), **физически** находящихся в памяти ЭВМ, и алгоритмов, имеющих доступ к ним.

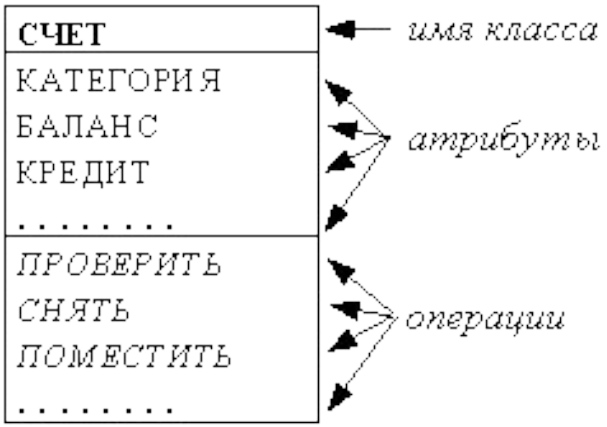
Каждый объект может обладать *именем* (идентификатором), используемым для доступа ко всей совокупности полей, его составляющих. В предельных случаях объект может не содержать полей или методов.

Класс - тип (описание структуры данных и операций над ними), предназначенный для описания множества объектов.

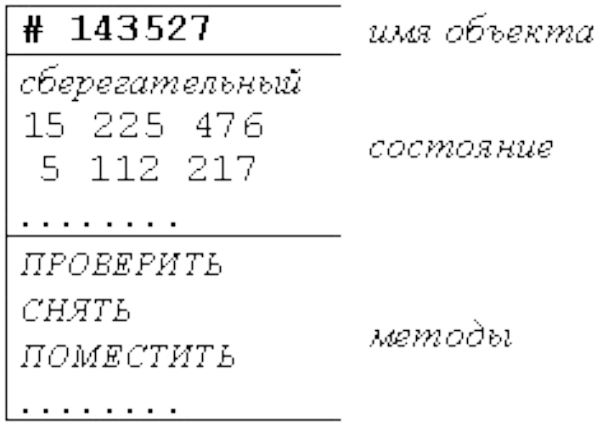
Классы и объекты

Каждый класс может иметь *подклассы* - классы, обладающие всеми или частью его свойств, а так же собственными свойствами. Класс, не имеющий ни одного представителя (объекта) обычно называют *абстрактным*.

Пример класса



Объект класса СЧЕТ



Классы и объекты

Рассмотрим **основные этапы разработки класса**.

Прежде всего, необходимо привести описание разрабатываемого класса. При разработке класса нужно представить определение класса, которое включает в себя:

- определение имени класса (определяет новый тип; абстракция, с которой будем иметь дело);
- определение состояния класса (состав, типы и имена полей в классе, предназначенных для хранения информации, а также уровни их защиты); данные, определяющие состояние класса, получили название членов-данных класса;
- определение методов класса (определение прототипов функций, которые обеспечат необходимую обработку информации). На этом этапе приводится *описание того, **что** мы **хотим** получить от класса, не указывая, **как** мы этого **добьемся***.

Классы и объекты

- Варианты графического изображения класса на диаграмме классов

- Примеры графического изображения конкретных классов

Переменные класса и константы

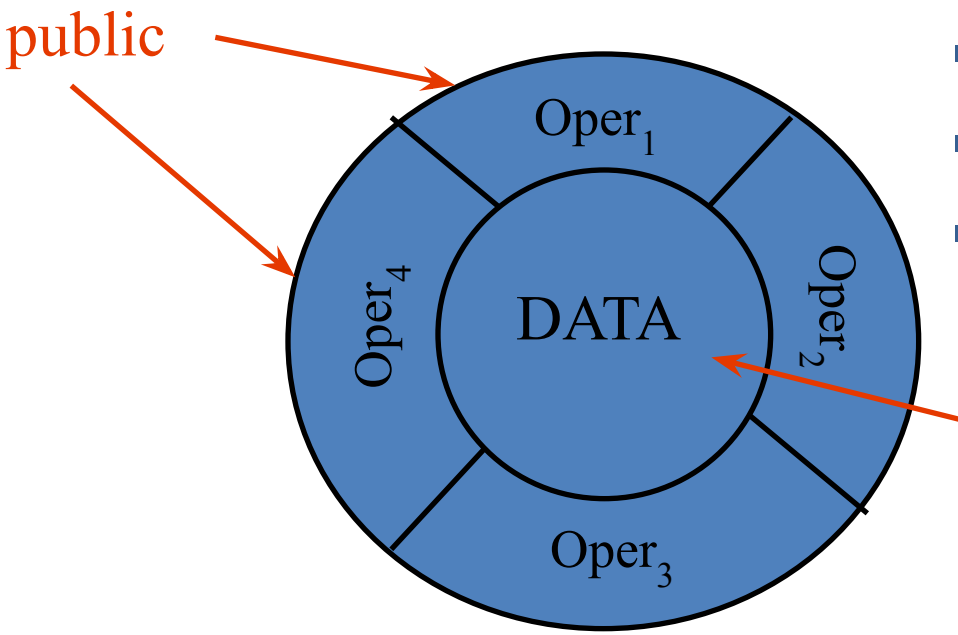
Данные – члены класса, которые называются полями или переменными класса, объявляются в классе следующим образом:

спецификатор тип имя;

***Спецификаторы доступа*:**

```
static      public
final       private
protected
```

Область видимости



- public
- private
- friendly (по умолчанию)
- protected

Классы и объекты. Example 1

```
package _java._se._02.classandobject;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Atributes {
    private int x; // переменная экземпляра класса
    private int y = 71; // переменная экземпляра класса
    public final int CURRENT_YEAR = 2007; // константа
    protected static int bonus; // переменная класса
    static String version = "Java SE 6"; // переменная класса
    protected Calendar now;
    public int method(int z) { // параметр метода
        z++;
        int a; // локальная переменная метода
        // a++; // ошибка компиляции, значение не задано
        a = 4; // инициализация
        a++;
        now = GregorianCalendar.getInstance();//инициализация
        return a + x + y + z;
    }
}
```

Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия ***только*** по *инициализации объекта*;

- Конструктор имеет то же имя, что и класс;
- Вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса;
- Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Классы и объекты. Example 2

Пример использования super и this

```
package _java._se._02.classandobject;
public class Point2D {
  private int x;
  private int y;
  public Point2D(int x, int y) {
    this.x = x;// this используется для присваивания полям
класса
    this.y = y;// x, y, значений параметров конструктора x, y, z
  }
}
```

```
package _java._se._02.classandobject;
public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
  public Point3D() {
    this(-1, -1, -1);// вызов конструктора Point3D с параметрами
  }
}
```

Классы и объекты. Example 3

Конструкторы. Пример перегрузки

```
package _java._se._02.classandobject;

public class TwoConstructors {
  private int id;
  private String text;
  public TwoConstructors() {
    super(); // если класс будет написан без конструкторов, то компилятор
        // предоставит его именно в таком виде
  }
  public TwoConstructors(int idc, String txt) {
    super(); // вызов конструктора суперкласса явным образом необязателен,
        // компилятор вставит его автоматически
    id = idc;
    text = txt;
  }
}
```

Классы и объекты

Объекты создаются динамически с помощью операции **new**, а уничтожаются автоматически.

Когда никаких ссылок на объект не существует (все ссылки на него вышли из области видимости программы) предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена.

“Сборка мусора” происходит нерегулярно во время выполнения программы. Рекомендовано ее выполнить вызовом метода **System.gc()** или **Runtime.getRuntime().gc()**.

Вызов метода **System.runFinalization()** приведет к запуску метода **finalize()** для объектов утративших все ссылки.

Классы и объекты

В Java пересмотрена концепция динамического распределения памяти: отсутствуют способы освобождения динамически выделенной памяти. Вместо этого реализована система автоматического освобождения памяти (сборщик мусора), выделенной с помощью оператора **new**.

Сборщик мусора уничтожает объекты, которым не соответствует ни одна ссылка из активного потока.

Аналогом деструктора можно считать метод **finalize(),** который исполняющая среда языка Java будет вызывать каждый раз, когда сборщик мусора будет уничтожать объекты этого класса.

Классы и объекты. Example 4

```
package _java._se._02.classandobject;

public class FinalizeDemo {
     public static void main(String[] args) {
          Student d1 = new Student(1);
          d1 = null;
          Student d2 = new Student(2);
          Object d3 = d2;
          d2 = d1;
          System.gc();
     }
}
```

Классы и объекты. Example 4

```
class Student {
     private int id;
     public Student(int value) {
          id = value;
     }
     protected void finalize() throws Throwable {
          try {
               System.out.println("объект удален id=" + id);
          } finally {
               super.finalize();
          }
     }
}
```

Результат выполнения:

```
объект удален
id=1
```

Методы классов, передача параметров в методы

Ссылки в методы передаются по значению. Выделяется память под параметры метода, и те переменные, которые являются ссылочными аргументами инициализируются значением своих фактических параметров. Таким образом, минимум две ссылки начинают указывать на один объект.

Классы и объекты. Example 5

```
package _java._se._02.classandobject;
import java.util.Date;

public class MyDate {
  public static void main(String[] args) {
    Date myDate = new Date();
    System.out.println("In main - before call function - myDate=" + myDate);
    changeDate(myDate);
    System.out.println("In main - after call function - myDate=" + myDate);
  }
  public static void changeDate(Date date) {
    System.out.println("In changeDate - before change - date=" + date);
    date.setYear(2999 - 1900);
    System.out.println("In changeDate - after change - date=" + date);
  }
}
```

Результат:

```
In main - before call function - myDate=Mon Sep 12 19:44:25 EEST 2011
In changeDate - before change - date=Mon Sep 12 19:44:25 EEST 2011
In changeDate - after change - date=Thu Sep 12 19:44:25 EEST 2999
In main - after call function - myDate=Thu Sep 12 19:44:25 EEST 2999
```

Классы и объекты

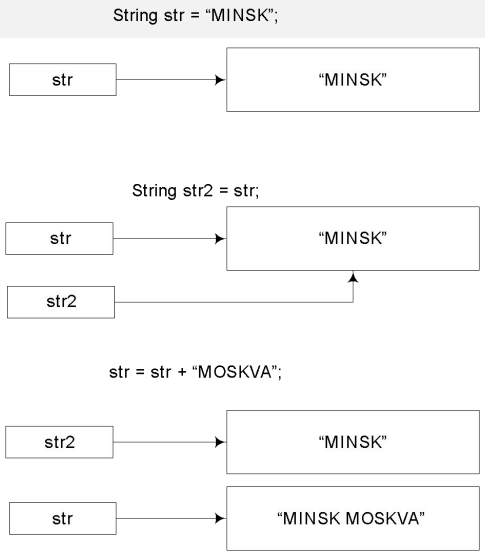
Следовательно, при передаче в метод аргумента-ссылки можно изменить состояние объекта и оно сохранится после возвращения из метода, так как в этом случае нового объекта не создается, а создается лишь новая ссылка, указывающая на старый объект. Из этого правила существует одно исключение – когда передается ссылка, указывающая на константный объект. Константный объект – это такой объект, изменить состояние которого нельзя. При попытке его изменить создается новый модифицированный объект. Примером таких объектов являются объекты класса String.

Классы и объекты. Example 6

```
package _java._se._02.classandobject;
public class WorkWithString {
    public static void main(String[] args) {
        String str = "MINSK";
        String str2 = str;
        System.out.println("str="+str+"   str2="+str2);
        str = str+" MOSKVA";
        System.out.println("str="+str+"   str2="+str2);
    }
}
```

Результат:

```
str=MINSK   str2=MINSK
str=MINSK MOSKVA
str2=MINSK
```



Таким образом, при попытке в методе изменить через переданную ссылку константный объект приведет к созданию нового объекта, на который будет ссылаться параметр метода. При завершении метода связь с этим объектом разрушится.

Классы и объекты. Example 7

```
package _java._se._02.classandobject;
public class WorkWithStringForChange {
  public static void main(String[] args) {
    String str = "MINSK";
    System.out.println("In main - before call function -
str="+str);
    changeString(str);
    System.out.println("In main - after call function -
str="+str);
  }
  public static void changeString(String s){
    System.out.println("In changeString - before change - s=" +s);
    s = s+" end.";
    System.out.println("In changeString - before change - s=" +s);
  }
}
```

Результат:

```
In main - before call function - str=MINSK
In changeString - before change - s=MINSK
In changeString - before change - s=MINSK
end.
In main - after call function - str=MINSK
```

Если необходимо вернуть в вызывающий метод ссылку на новый **константный** объект, созданный в этом методе, следует указать её тип как тип возвращаемого методом значения и использовать **return**.

Классы и объекты. Example 8

```
package _java._se._02.classandobject;
public class WorkWithStringForChageGood {
  public static void main(String[] args) {
    String str = "MINSK";
    System.out.println("In main - before call function -
str="+str);
    str = changeString(str);
    System.out.println("In main - after call function -
str="+str);
  }
  public static String changeString(String s){
    System.out.println("In changeString - before change - s=" +s);
    s = s+" end.";
    System.out.println("In changeString - before change - s=" +s);
    return s;
  }
}
```

Результат:

```
In main - before call function - str=MINSK
In changeString - before change - s=MINSK
In changeString - before change - s=MINSK
end.
In main - after call function - str=MINSK
end.
```

Классы и объекты. Example 9

Явные и неявные параметры метода. Явные параметры метода определяются списком параметров. Неявный параметр – это this – ссылка на вызвавший метод объект.

```
package _java._se._02.classandobject;
public class Book {
  private String title;
  private String author;
  private final int yearPublished;
  public Book(String title, String author,  int yearPublished)
  {
     this.title = title;
     this.author = author;
     this.yearPublished = yearPublished;
  }
  public String getBook()
  {
     return this.title+" "+this.author+" "+this.yearPublished;
  }
}
```

Статические методы и поля

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются **переменными класса**.

Если один объект изменит значение такого поля, то это изменение увидят все объекты.

Классы и объекты

Для работы со статическими атрибутами используются **статические методы**, объявленные со спецификатором **static**.

- являются методами класса;
- не привязаны ни к какому объекту;
- не содержат указателя this на конкретный объект, вызвавший метод;
- реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции;
- статические поля и методы не могут обращаться к нестатическим полям и методам напрямую (по причине недоступности указателя this), так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

Классы и объекты. Example 10

```
package _java._se._02.classandobject;
public class Mark {
     private int mark = 3;
     public static int coeff = 5;
     public double getResult1() {
          return (double) coeff * mark / 100;
     }
     public static void setCoeffFloat(float c) {
          coeff = (int) (coeff * c);
     }
     public void setMark(int mark) {
          this.mark = mark;
     }
     // из статического метода нельзя обратиться
     // к нестатическим полям и методам
     public static int getResult2() {
          setMark(5);// ошибка
          return coeff * mark / 100;// ошибка
     }
}
```

Статические поля используются довольно редко, а вод поля **static final** наоборот часто. Очень часто используемая статическая константа **System.out.**

```
public class System
{
    …
    public static final PrintStream out = …
    …
}
```

другая часто используемая константа – **Math.PI**. Статические константы нет смысла делать закрытыми, а обращаются к ним через имя класса:

имя_класса.имя_статической_константы

Статические методы не работают с объектами, поэтому их использовать следует в двух случаях:

когда методу *не нужен доступ к состоянию объекта*, а все необходимые параметры задаются явно (например, метод Math.pow(…).

когда методу нужен *доступ только к статическим полям* класса (статический метод не может получить доступ к нестатическим полям класса, так как они принадлежат объектам, а не классам).

Статические методы можно вызывать, даже если ни один объект этого класса не создан. Кроме того, статические методы часто используют в качестве порождающих, т.е. таких методов. которые создают объект своего класса и возвращают ссылку на него.

Классы и объекты

Модификатор final. Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода.

Методы, объявленные как **final**, нельзя замещать в подклассах, для классов – создавать подклассы.

Константа может быть объявлена **как поле класса, но не проинициализирована**. В этом случае она должна быть проинициализирована в логическом блоке класса, заключенном в {}, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса.

Константы могут быть объявлены в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя.

Значение константному полю можно присвоить при объявлении, в логическом блоке инициализации или в конструкторе.

Классы и объекты. Example 11

```
package _java._se._02.classandobject;
public class Rector {
    // инициализированная константа
    final int ID = (int) (Math.random() * 10);
    // неинициализированная константа
    final String NAME_RECTOR;
    public Rector() {
        // инициализация в конструкторе
        NAME_RECTOR = "Старый";
        // только один раз!!!
    }
    // {NAME_RECTOR = "Новый";}
    // только один раз!!!
    public final void jobRector() {
        // реализация
        // ID = 100;
        // ошибка!
    }
```

Классы и объекты. Example 11

```
    public boolean checkRights(final int num) {
        // id = 1; //ошибка!
        final int CODE = 72173394;
        if (CODE == num)
            return true;
        else
            return false;
    }
    public static void main(String[] args) {
        System.out.println(new Rector().ID);
    }
}
```

```
package _java._se._02.classandobject;
public class Prorector extends Rector {
    // public void jobRector(){} //запрещено!
}
```

Модификатор native

Приложение на языке Java может вызывать методы, написанные на языке C++. Такие методы объявляются с ключевым словом **native**, которое сообщает компилятору, что метод реализован в другом месте.

```
public native int loadCripto(int num);
```

Методы, помеченные **native**, можно переопределять обычными методами в подклассах.

Модификатор synchronized .

При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным.

Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении.

Блоки инициализации

При описании класса могут быть использованы логические блоки. **Логическим блоком называется код**, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса.

```
{ /* код */ }
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов как текущего класса, так и не принадлежащих ему.

Для инициализации статических переменных существуют **статические блоки инициализации**. В этом случае фигурные скобки предваряются ключевым словом **static**.

Классы и объекты

При создании объекта блоки инициализации класса вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса.

Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки this, представляющую собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается **только один** раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

Классы и объекты. Example 12

```
package _java._se._02.classandobject;
public class Department {
    {
        System.out.println("logic (1) id=" + this.id);
    }
    static {
        System.out.println("static logic");
    }
    private int id = 7;
    public Department(int d) {
        id = d;
        System.out.println("конструктор id=" + id);
    }
    int getId() {
        return id;
    }
    {
        id = 10;
        System.out.println("logic (2) id=" + id);
    }
```

Классы и объекты. Example 12

```
    public static void main(String[] args) {
        Department obj = new Department(71);
        System.out.println("значение id=" + obj.getId());
    }
}
```

Результат:

```
static logic
logic (1) id=0
logic (2) id=10
конструктор
id=71
значение id=71
```

Инициализация полей класса. Общий порядок инициализации следующий

1. При загрузке классов в память статические поля инициализируются значениями по умолчанию.
2. Статически поля классов инициализируются значением, присвоенным при объявлении.
3. Выполняется статический блок инициализации.
4. При вызове конструктора класса все поля данных инициализируются своими значениями, предусмотренными по умолчанию.
5. Инициализаторы всех полей и блоки инициализации выполняются в порядке их перечисления в объявлении класса.
6. Если в первой строке конструктора вызывается другой конструктор, то выполняется вызванный конструктор.
7. Выполняется тело конструктора.

Классы и объекты

Если значение поля не задано в конструкторе явно, ему автоматически присваивается значение по умолчанию: *числам* — **нули**, *булевским значениям* — **false**,а *ссылкам на объект* — **null**.

Перегрузка методов. Метод называется перегруженным, если существует несколько его версий с одним и тем же именем, но с различным списком параметров.

- Перегрузка реализует «*раннее связывание*». Перегрузка может ограничиваться одним классом.

- Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными.

Перегрузка методов.

- Если в последнем случае списки параметров совпадают, то имеет место другой механизм – переопределение метода.
- Статические методы могут перегружаться нестатическими и наоборот – без ограничений.
- При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод.

Классы и объекты. Example 13

```
package _java._se._02.classandobject;
public class NumberInfo {

    public static void viewNum(Integer i) { // 1
        System.out.printf("Integer=%d%n", i);
    }

    public static void viewNum(int i) {// 2
        System.out.printf("int=%d%n", i);
    }

    public static void viewNum(Float f) { // 3
        System.out.printf("Float=%.4f%n", f);
    }

    public static void viewNum(Number n) { // 4
        System.out.println("Number=" + n);
    }
```

Классы и объекты. Example 13

```
    public static void main(String[] args) {
        Number[] num = { new Integer(7), 71, 3.14f, 7.2 };
        for (Number n : num)
            viewNum(n);
        viewNum(new Integer(8));
        viewNum(81);
        viewNum(4.14f);
        viewNum(8.2);
    }
}
```

Результат:

```
Number=7
Number=71
Number=3.14
Number=7.2
Integer=8
int=81
Float=4.140
0
Number=8.2
```

Классы и объекты

При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

С одной стороны, этот механизм снижает гибкость, с другой – все возможные ошибки при обращении к перегруженным методам отслеживаются на этапе компиляции, в отличие от переопределенных методов, когда их некорректный вызов приводит к возникновению исключений на этапе выполнения.

При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

Классы и объекты

Класс java.lang.Object - родительский для всех классов

Содержит следующие методы:

- **protected Object clone()** – создает и возвращает копию вызывающего объекта;
- **boolean equals(Object ob)** – предназначен для переопределения в подклассах с выполнением общих соглашений о сравнении содержимого двух объектов;
- **Class<? extends Object> getClass()** – возвращает объект типа **Class;**
- **protected void finalize()** – вызывается перед уничтожением объекта автоматическим сборщиком мусора (garbage collection);
- **int hashCode()** – возвращает хэш-код объекта;
- **String toString()** – возвращает представление объекта в виде строки.

Переопределение метода equals() - метод **equals()** при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае.

При переопределении должны выполняться соглашения:

- **рефлексивность** – объект равен самому себе;
- **симметричность** – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;
- **транзитивность** – если метод **equals()** возвращает значение true при сравнении объектов **x** и **y,** а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;
- **непротиворечивость** – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- **ненулевая ссылка** при сравнении с литералом **null** всегда возвращает значение **false**.

Переопределение метода hashCode() - метод **int hashCode()** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

- во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;
- все одинаковые по содержанию объекты одного типа должны иметь одинаковые хэш-коды;
- различные по содержанию объекты одного типа могут иметь различные хэш-коды.

Метод **hashCode()** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта.

Следует переопределять всегда, когда переопределен метод **equals().**

Переопределение метода toString() - метод **toString()** следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе **Object**.

В классе Object возвращает строку с описанием объекта в виде:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Метод вызывается автоматически, когда объект выводится методами **println(), print()** и некоторыми другими.

При реализации всегда следует стремиться к тому, чтобы сообщить максимальную информацию об объекте.

Классы и объекты. Example 14

```
package _java._se._02.classandobject;
public class Student {
    private int id;
    private String name;
    private int age;
    public Student(int id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
```

Классы и объекты. Example 14

```
  public boolean equals(Object obj) {
     if (this == obj)
          return true;
     if (obj == null)
          return false;
     if (getClass() == obj.getClass()) {
          Student temp = (Student) obj;
          return this.id == temp.id && name.equals(temp.name)
                    && this.age == temp.age;
     } else
          return false;
  }
  public int hashCode() {
     return (int) (31 * id + age
          + ((name == null) ? 0 : name.hashCode()));
  }
  public String toString() {
     return getClass().getName() + "@name"
          + name + " id:" + id + " age:” + age;
  }
}
```

Методы с переменным числом параметров

!!! Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[]... args){}
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
void methodName(String...args) {}
```

Классы и объекты. Example 15

```
package _java._se._02.classandobject;
public class DemoVarargs {
    public static int getArgCount(Integer... args) {
        if (args.length == 0)
            System.out.print("No arg=");
        for (int i : args)
            System.out.print("arg:" + i + "  ");
        return args.length;
    }
    public static void main(String args[]) {
        System.out.println("N=" + getArgCount(7, 71, 555));
        Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
        System.out.println("N=" + getArgCount(i));
        System.out.println(getArgCount());
    }
}
```

Результат:

```
arg:7  arg:71  arg:555  N=3
arg:1  arg:2  arg:3  arg:4  arg:5  arg:6  arg:7
N=7
No arg=0
```

Классы и объекты. Example 16

```
package _java._se._02.classandobject;
public class DemoOverload {

  public static void printArgCount(Object... args) {  // 1
    System.out.println("Object args: " + args.length);
  }

  public static void printArgCount(Integer[]... args) {  // 2
    System.out.println("Integer[] args: " + args.length);
  }

  public static void printArgCount(int... args) { // 3
    System.out.print("int args: " + +args.length);
  }
```

Классы и объекты. Example 16

```
    public static void main(String[] args) {
        Integer[] i = { 1, 2, 3, 4, 5 };
        printArgCount(7, "No", true, null);
        printArgCount(i, i, i);
        printArgCount(i, 4, 71);
        printArgCount(i); // будет вызван метод 1
        printArgCount(5, 7);
        // printArgCount();//неопределенность!
    }
}
```

Результат:

```
Object args: 4
Integer[] args:
3
Object args: 3
Object args: 5
int args: 2
```

ТРИ КИТА ООП

Три кита ООП

Объектно-ориентированное программирование основано на трех принципах:

- **Инкапсуляции;**
- **Наследовании;**
- **Полиморфизме.**

и одном механизме:

- **Позднее связывание**

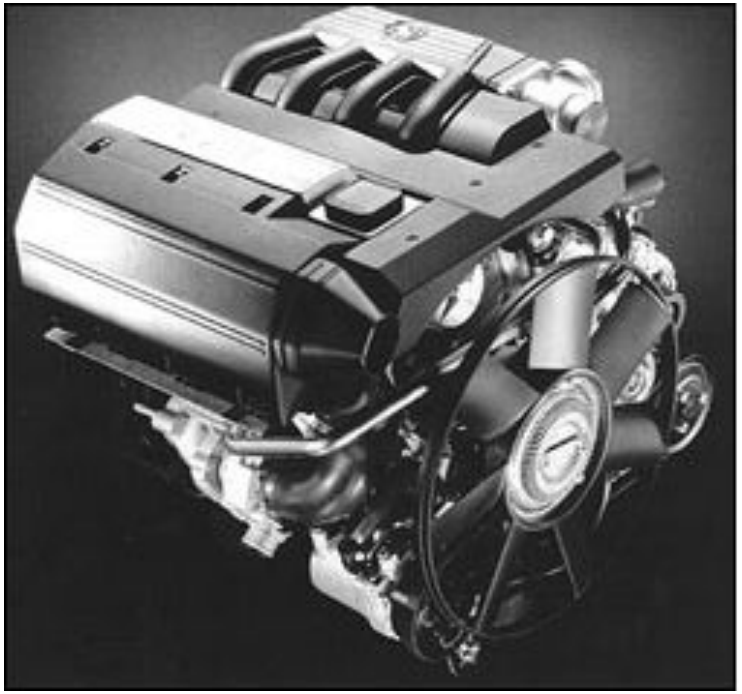
  Включение сюда механизма вопрос крайне спорный.

Три кита ООП

Инкапсуляция (encapsulation) - это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создаётся так называемый "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object).

Три кита ООП

Инкапсуляция



Три кита ООП

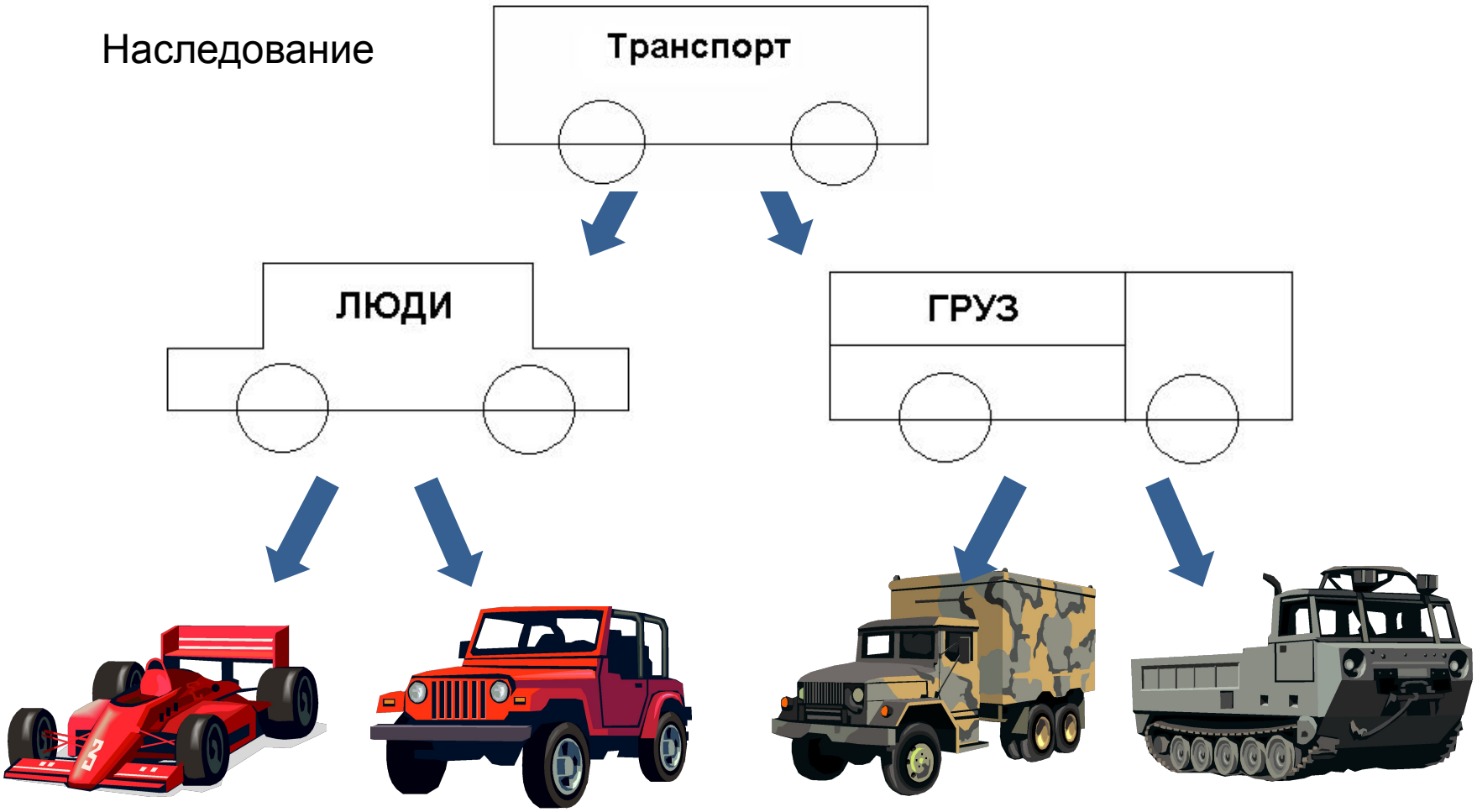
Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование бывает двух видов:

- *одиночное* - когда каждый класс имеет одного и только одного предка;
- *множественное* - когда каждый класс может иметь любое количество предков.

Три кита ООП

Наследование



Три кита ООП

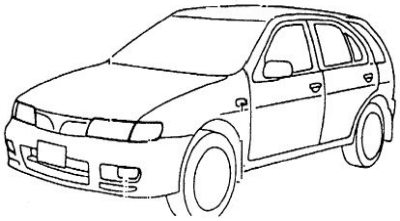
Полиморфизм (polymorphism) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

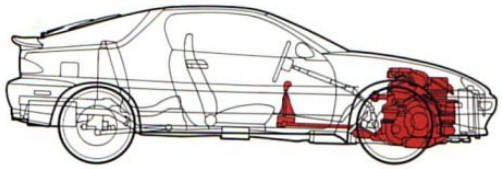
Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий.

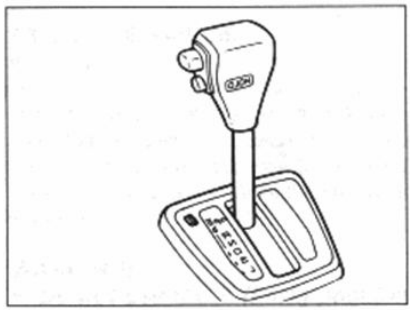
В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов".

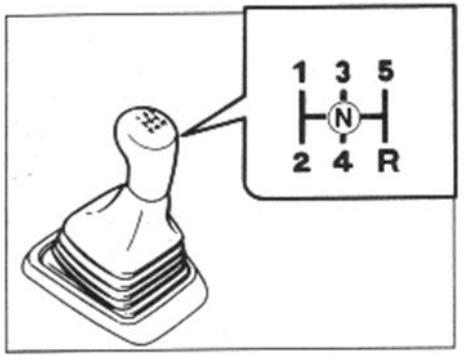
Три кита ООП

Полиморфизм



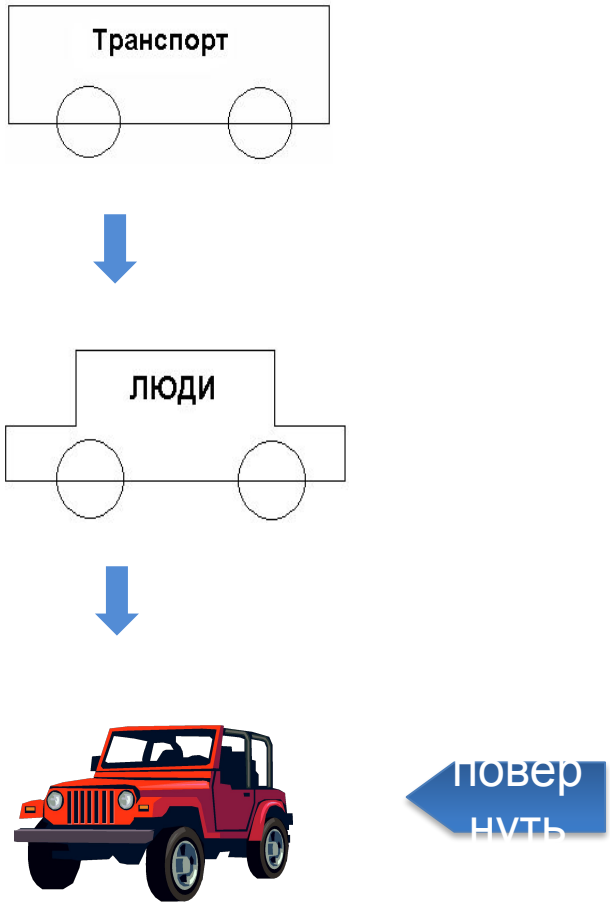






Три кита ООП

Позднее связывание. При вызове того или иного метода класса сначала ищется метод у самого класса. Если метод найден, то он выполняется и поиск этого метода на этом завершается. Если же метод не найден, то обращаемся к родительскому классу и ищем вызванный метод у него. Если найден - поступаем как при нахождении метода в самом классе. А если нет - продолжаем дальнейший поиск вверх по иерархическому дереву. Вплоть до корня(верхнего класса) иерархии.

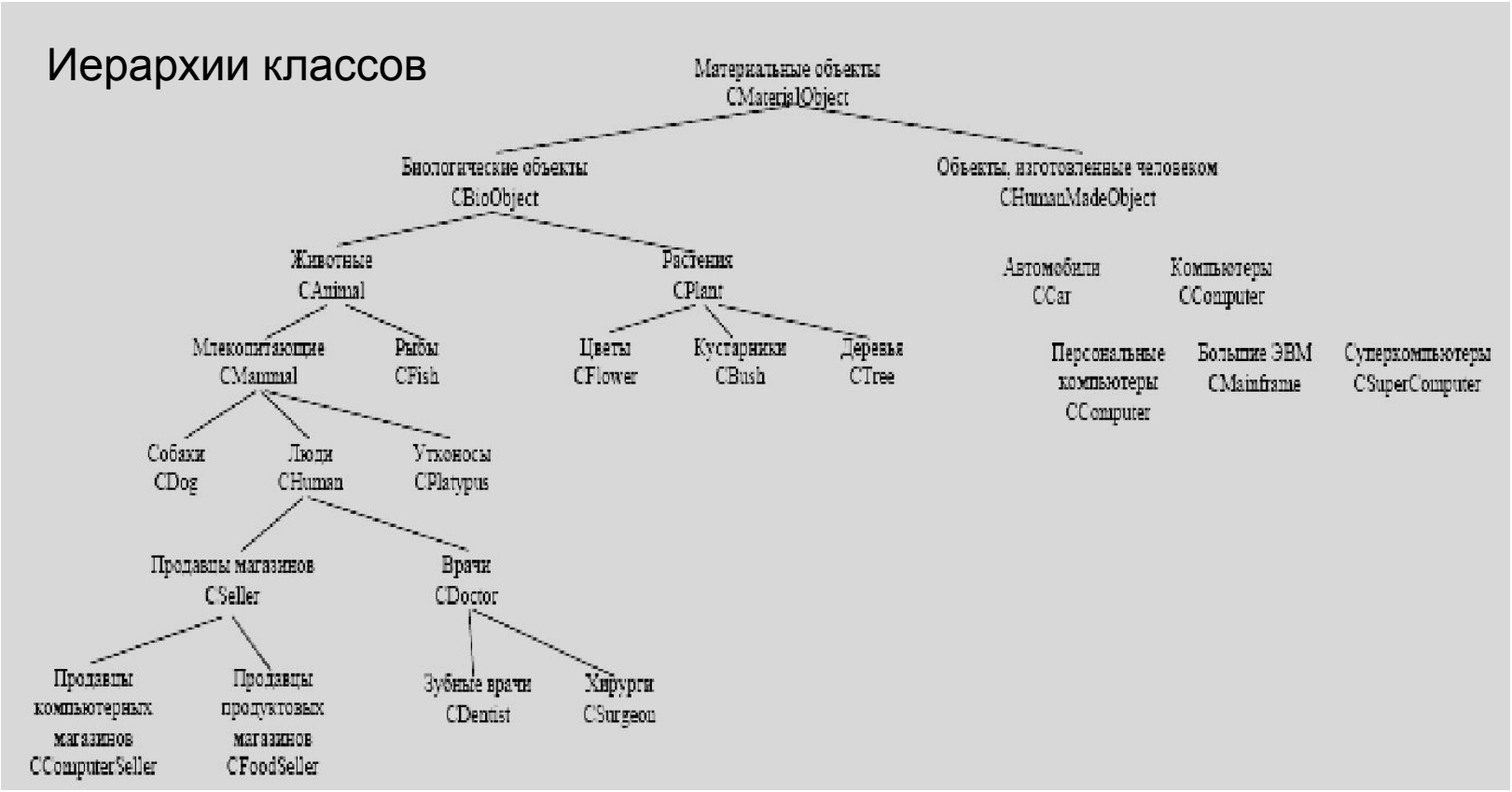


НАСЛЕДОВАНИЕ

Наследование

Понятие наследования. Один класс может наследовать или расширять поля и методы другого класса с помощью ключевого слова extends. Класс, который выступает базой для расширения, называют суперклассом, класс, который непосредственно проводит расширение, - подклассом. Подкласс имеет доступ **ко всем открытым полям** и методам суперкласса, так, словно они описаны в подклассе: производный класс не имеет доступа к закрытым полям и методам класса. Также подкласс может добавлять методы и переопределять методы.

Наследование



Наследование

Переопределение методов. Переопределенным методом называют метод, описанный в производном классе, сигнатура этого метода совпадает с сигнатурой метода, описанного в суперклассе.

Наследование. Example 17

```
public class Book {
    private String title;
    private int yearPublished;
    private int price;
    public Book() {}
    public Book(String title, int yearPublished, int price)    {
        this.title = title;
        this.yearPublished = yearPublished;
        this.price = price;
    }
    public String getTitle()    {
        return title;
    }
    public int getYearPublished()    {
        return yearPublished;
    }

    public int getPrice()    {
        return price;
    }
    public void printReport()    {
        System.out.println("Название: "+title+" год издания: "+yearPublished+"
цена: "+price);
    }
}
```

Наследование. Example 17

```
public class ProgrammerBook extends Book{
    public ProgrammerBook(String title, int yearPublished, int price,
String level)    {
        super(title,yearPublished,price);
        this.level = level;
    }
    public String getLevel()    {
        return level;
    }
    public void printReport()    {
        System.out.println("Название: "+getTitle()+" год издания:
"+getYearPublished()+" цена: "+getPrice()+" уровень: "+level);
    }
    private String level;
}
public class BookInspector {
    public static void main(String[] args)    {
        Book mybook = new Book("Золушка",2000, 19000);
        ProgrammerBook myprogrbook =new
ProgrammerBook("Java",2006,46000,"hight");
        mybook. printReport();
        myprogrbook. printReport();
    }
}
```

Наследование. Example 18

Методы подставки. С пятой версии языка появилась возможность при переопределении методов указывать другой тип возвращаемого значения, в качестве которого можно использовать только типы, находящиеся ниже в иерархии наследования, чем исходный тип.

```
public class CourseHelper {
     public Course getCourse(){
          System.out.println("Course");
         return new Course();
     }
}

public class BaseCourseHelper extends CourseHelper {
      public BaseCourse getCourse(){
          System.out.println("BaseCourse");
         return new BaseCourse();
     }
}
```

Наследование. Example 18

```
public class RunnerCourse {
    public static void main(String[] args) {
        CourseHelper bch = new BaseCourseHelper();
        Course course = bch.getCourse();
        //BaseCourse course = bch.getCourse();//ошибка
компиляции
        System.out.println(bch.getCourse().id);
    }
}
```

В данной ситуации при компиляции в подклассе **BaseCourseHelper** создаются два метода. При обращении к методу **getCourse()** версия метода определяется «ранним связыванием» без использования полиморфизма, но при выполнении вызывается метод-подставка. Обращение к полю производится по типу ссылки, возвращаемой методом **getCourse()**, то есть к полю класса **Course**.

Наследование

Вызов конструкторов при наследовании. При создании объектов производного класса, конструктор производного класса вызывает соответствующий конструктор базового класса с помощью ключевого слова super(параметры). Вызов конструктора базового класса из конструктора производного должен быть произведен в первой строке конструктора производного класса. Если конструктор производного класса явно не вызывает конструктор базового, то происходит вызов конструктора по умолчанию базового класса, в этом случае в базовом классе должен быть определен конструктор по умолчанию.

Если метод переопределен в производном классе, то одноименный метод базового класса можно вызвать из производного с помощью конструкции.

```
super.имя_метода(параметры);
```

Наследование. Example 19

```
public class ProgrammerBook extends Book{
…
    public void printReport(){
        super.printReport();
        System.out.println(" уровень: "+level);
    }
…
}
```

Следует помнить, что при вызове show() обращение производится к ближайшему суперклассу.

Наследование. Example 20

Ссылки на суперкласс и их свойства. Объектная переменная базового класса может ссылаться на объекты как базового, так и производного классов. Такая возможность называется полиморфизмов. Автоматический выбор нужного метода во время выполнения программы называется динамическим связыванием (dynamic binding).

```
public class BookInspector {
    public static void main(String[] args) {
        Book mybook = new Book("Золушка",2000, 19000);
        Book myprogrbook =
         new ProgrammerBook("Java",2006,46000,"hight");
        mybook.printReport();
        myprogrbook.printReport();
    }
}
```

Наследование

Когда вызывается метод, принадлежащий объекту, происходит следующее.

1. Компилятор проверяет объявленный тип объекта и имя метода. Допустим, происходит вызов метода x.f(args), прочем неявный параметр объявлен как объект класса C. Заметим, что могут существовать несколько методов с именем f, имеющих разные типы параметров (например, метод f(int) и метод f(String)). Компилятор пронумерует все методы с именем f в классе C и все открытые методы с именем f в суперклассах класса C.

Наследование

2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов с именем f есть только одни метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется разрешением перегрузки (overloading resolution). Например, при вызове x.f(“Hello”) компилятор выберет метод f(String), а не метод f(int). Ситуация может осложниться вследствие преобразования типов (int в double). Если компилятор не находит ни одного метода с подходящим набором параметров, или в результате преобразования типов возникает несколько методом, соответствующих данному вызову, выдается сообщение об ошибке

Наследование

3. Если метод является закрытым (private), статическим (static), терминальным (final) или конструктором, компилятор точно знает, какой метод вызвать. Такой процесс называется статическим связыванием (static binding). В противном случае метод, подлежащий вызову, определяется по фактическому типу неявного параметра, и во время выполнения программы должно использоваться динамическое связывание. В нашем примере компилятор сгенерировал бы команду метода f(String) с помощью динамического связывания.

Наследование

4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую фактическому типу объекта, на который ссылается переменная x. Допустим, что объект имеет фактический тип D, являющийся суперклассом класса C. Если в классе D определен метод f(string), то вызывается именно он. Если нет, то поиск метода f(String), подлежащего вызову, выполняется в суперклассе и т.д.

Наследование. Example 21

Предотвращение переопределения методов. Чтобы предотвратить переопределение некоторых их необходимо объявить терминальными с помощью ключевого слова final. Если в классе Book объявить метод getPrice() терминальным, то в производном классе ProgrammerBook переопределить его будет нельзя.

```
public class Book {
…
     public final int getPrice()
     {
         return price;
     }…
}
public class ProgrammerBook extends Book{
…
     public final int getPrice() // так делать нельзя
     {
         return price;
     } …
}
```

Наследование

Классы, объявленные как терминальными, нельзя расширить. Объявить терминальный класс можно следующим образом.

```
final public class Book
{}
```

Если класс объявлен терминальным, то это не значит, что его поля стали константными

Наследование

Приведение типов при наследовании. Как известно, в языке Java каждая объектная переменная имеет тип, описывающий разновидность объекта, на который ссылается переменная, и все, что он может делать.

На основе описания классов компилятор проверяет, сужает или расширяет возможности класса программист, объявляющий переменную. Если переменной суперкласса присваивается объект подкласса, возможности класса сужаются, и компилятор без проблем позволяет программисту сделать это. Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса (subclass).

Наследование. Example 22

```
public class BookInspector {
    public static void main(String[] args){
        Book[] mybook = new Book[4];
        mybook[0] =  new Book("Золушка", 2000,19000);
        mybook[1] =  new
ProgrammerBook("Java",2006,46000,"hight");
        mybook[2] =  new Book("Дневной дозор",2002,25000);
        mybook[3] =  new
ProgrammerBook("C++",2009,32000,"medium");
        Book book;
        book = mybook[0];
        book.printReport();
        book = mybook[1];
        book.printReport();
        ProgrammerBook prbook;
        // prbook = (ProgrammerBook)mybook[0]; // ОШИБКА
        prbook = (ProgrammerBook)mybook[1];  // ВСЕ ХОРОШО
        report(mybook);
    }
    static void report(Book[] book) {
        for(int i=0; i<book.length; i++)
            book[i].printReport();
    }
}
```

Наследование

При недопустимом преобразовании типов при выполнении программы система обнаружит несоответствие и возбудит исключительную ситуацию. Если её не перехватить, то работа программы будет остановлена. Следовательно, перед приведением типов следует проверить его на корректность. Делается это с помощью оператора **instanceof**.

Наследование. Example 23

```
public class BookInspector {
    public static void main(String[] args) {
        Book[] mybook = new Book[4];
        mybook[0] = new Book("Золушка", 2000,19000);
        mybook[1] = new ProgrammerBook("Java",2006,46000,"hight");
        mybook[2] = new Book("Дневной дозор",2002,25000);
        mybook[3] = new ProgrammerBook("C++",2009,32000,"medium");
        report(mybook);
    }
    static void report(Book[] book){
        ProgrammerBook prbook;
        Book bk;
        for(int i=0; i<book.length; i++){
            if(book[i] instanceof ProgrammerBook){
                prbook = (ProgrammerBook)book[i];
                System.out.println("Programmer level = " +
prbook.getLevel());
            } else {
                bk = (Book)book[i];
                System.out.println("Book price = " + bk.getPrice());
            }
        }
    }
}
```

Наследование

Компилятор не позволить выполнить некорректное приведение типов. Например, приведение типов

```
Date dt = (Date)book[1];
```

приведет к ошибке на стадии компиляции, поскольку класс **Date** не является подклассом класса **Book**.

Наследование

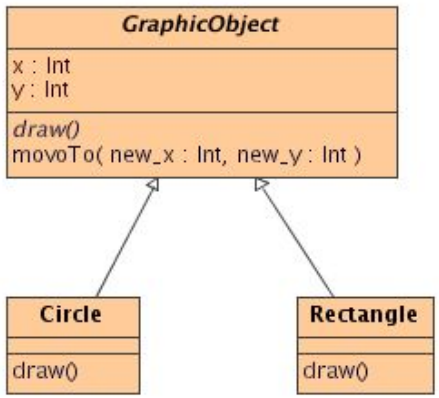
Абстрактные методы и классы. Часто при проектировании иерархии классов верхние классы иерархии становятся все более и более абстрактными, так что реализовывать некоторые методы в них не имеет никакого смысла. Однако удалить их из класса нельзя, так как при дальнейшем использовании базовых объектных ссылок на объекты производных классов необходим доступ к переопределенным методам, а он возможен только при наличии в них метода с такой же сигнатурой как в базовом классе. В таком случае метод следует объявлять абстрактным. В классе, где метод объявляется абстрактным, его реализация не требуется. Если в классе есть абстрактные методы, тои класс можно объявить абстрактным.

Наследование. Example 24

- Абстрактные классы объявляются с ключевым словом abstract и могут содержать объявления абстрактных методов, которые не реализованы в этих классах.
- Объекты таких классов создать нельзя, можно создать объекты подклассов, которые реализуют эти методы.
- Абстрактные методы помещаются в абстрактных классах или интерфейсах, тела таких методов отсутствуют и реализуются в подклассах.

```
public abstract class AbstractCourse {
    private String name;
    public AbstractCourse() {
    }
    public abstract void changeTeacher(int id);
    // определение метода отсутствует
    public setName(String n) {
        name = n;
    }
```

Наследование. Example 24




```
public abstract class GraphicObject {
   public abstract void draw();
//абстрактный метод

   public void moveTo(int x, int y) {
     //движение центра фигуры
   }
}

class Circle extends GraphicObject {
   public void draw(){
     //рисуем круг
   }
}

class Runner {
   public static void main(String[] args) {
     GraphicObject mng; //  можно объявить ссылку
     //mng = new GraphicObject();
     //нельзя создать объект!
     mng = new Circle();
     mng.draw();
   }
}
```

Наследование. Example 25

```
public abstract class Book {
    private String title;
    private int yearPublished;
    private int price;

    public Book() {}
    public Book(String title, int yearPublished, int price){
        this.title = title;
        this.yearPublished = yearPublished;
        this.price = price;
    }
    public abstract void printReport();

    public String getTitle(){
        return title;
    }
    public int getYearPublished(){
        return yearPublished;
    }
    public int getPrice(){
        return price;
    }
}
```

Наследование. Example 25

```
public class ProgrammerBook extends Book{
    public ProgrammerBook(String title, int year_published, int price, String
level){
        super(title,year_published,price);
        this.level = level;
    }
    public String getLevel(){
        return level;
    }
    public void show(){
        System.out.println("Название: "+getTitle()+" - год издания:
"+getYearPublished()+" - цена: "+getPrice()+" уровень: "+level);
    }
    private String level;
}
public class ChildrenBook extends Book{
    public ChildrenBook(String title, int year_published, int price, int yearFrom){
        super(title,year_published,price);
        this.yearFrom = yearFrom;
    }
    public void printReport() {
                System.out.println("Название: "+getTitle()+" - год издания:
"+getYearPublished()+
" - цена: "+getPrice()+" возраст: "+yearFrom);
    }
    private int yearFrom;
}
```

Наследование

При расширении абстрактного класса все его абстрактные методы необходимо определить или подкласс также объявить абстрактным. Нельзя создавать объекты абстрактных классов, однако можно объявлять объектные переменные.

Наследование. Example 26

Статические методы при наследовании. Для статических методов в Java полиморфизм неприменим

```
public class Book {
    public static void printReport() {
        System.out.println("Метод show() из класса Book");
    }
}
public class ProgrammerBook extends Book{
    public static void printReport() {
        System.out.println("Метод show() из класса ProgrammerBook");
    }
}

public class BookInspector {
    public static void main(String[] args) {
        Book[] mybook = new Book[2];
        mybook[0] = new Book();
        mybook[1] = new ProgrammerBook();
        mybook[0]. printReport() ();// Метод show() из класса Book
        mybook[1]. printReport() ();// Метод show() из класса Book
    }
}
```

Наследование. Example 27

Наследование от стандартных классов. Кроме собственных Java позволяет расширять и стандартные классы.

```
import java.sql.Time;
public class MyTime extends Time {
    public MyTime(long i) {
        super(i);
    }
    public String current(){
        long hours = getHours();
        if(hours >= 4 && hours < 12) return "утро";
        else if ((hours >12 && hours < 17)) return "день";
        else if (hours >= 17 && hours < 23) return "вечер";
        else return "ночь";
    }
    public static void main(String[] args){
        MyTime mytime = new MyTime(300000000);
        System.out.println(mytime.current());
    }
}
```

ИНТЕРФЕЙСЫ

Интерфейсы

Интерфейсы в Java применяются для добавления к классам новых возможностей, которых нет и не может быть в базовых классах. Интерфейсы говорят о том, что класс может делать, но не говорят, как он должен это делать. Предположим, что существует интерфейс ПЛАТИТЬ_ЗАРПЛАТУ. Возможность “платить зарплату” должна быть у многих классов. Например, класс ДИРЕКТОР может реализовывать интерфейс ПЛАТИТЬ_ЗАРПЛАТУ, класс КАССИР также может иметь возможность ПЛАТИТЬ_ЗАРПЛАТУ. Однако, в интерфейсе не будет сказано, как классы будут эту возможность (платить долгожданную зарплату) осуществлять. Интерфейс только гарантирует, что класс выполняет какие-то функции, а как он их выполняет – дело неинтерфейсное. Так, диплом подтверждает, что человек может делать какую-то работу, однако то, как он её будет выполнять по диплому непонятно. Следовательно, диплом здесь выступает в роли интерфейса.

Во многих источниках ещё можно прочитать, что интерфейсы являются заменой множественному наследованию. Только интерфейсы более удобны, более логичны и менее громоздки.

Интерфейсы

Определение интерфейса. Синтаксис определения интерфейса следующий.

```
доступ interface имя_интерфейса
{
// пля интерфейса
// методы интерфейса
}
```

Поля интерфейса по умолчанию являются **final static**. Все методы по умолчанию открыты (**public**).

```
public interface Square {
    double square();

    double PI = 3.1415926;
}
```

Интерфейсы. Example 28

Реализация интерфейса происходит в классе с помощью ключевого слова implements.

Если реализуемых интерфейсов несколько, то они перечисляются через запятую. Интерфейс считается реализованным, когда в классе и/или в его суперклассе реализованы все методы интерфейса.

```
public class Box implements Square{
    private int a;
    public Box(int a) { this.a = a; }
    public double square() { return a*a; }
    public void print() {
     System.out.println("Square box: "+square());}
}
public class Rectangle implements Square{
    private int a, b;
    public Rectangle(int a, int b) {this.a=a; this.b=b; }
    public double square() { return a*b; }
    public void print() {System.out.println("Square
rectangle: "+square());}
}
```

Интерфейсы. Example 28

```
public class Circle implements Square{
    private int r;
    public Circle(int r) { this.r = r;}
    public double square() { return r*r*Square.PI;}
    public void print() {System.out.println("Square circle: «
                    +square());}
}
public class Test {
    public static void main(String[] args){
        Box box = new Box(4);
        Rectangle rectangle = new Rectangle(2,3);
        Circle circle = new Circle(3);
        box.print();
        rectangle.print();
        circle.print();
        System.out.println("Box: "+box.square());
        System.out.println("Rectangle: "+rectangle.square());
        System.out.println("Circle: "+circle.square());
    }
}
```

Интерфейсы

Свойства интерфейсов.

- С помощью оператора **new** нельзя создать экземпляр интерфейса.
- Можно объявлять интерфейсные ссылки.
- Интерфейсные ссылки должны ссылать на объекты классов, реализующих данный интерфейс.
- Через интерфейсную ссылку можно вызвать только методы определенные с интерфейсе.
- С помощью оператора **instanseof** можно проверять, реализует ли объект определенный интерфейс.
- Если класс не полностью реализует интерфейс, то он должен быть объявлен как **abstract**.
- Интерфейс может быть расширен при помощи наследования от другого интерфейса, синтаксис в этом случае аналогичен синтаксисом наследования классов .

Интерфейсы. Example 29

```
public class Test {
    public static void main(String[] args)    {
        Box box = new Box(4);
        //box = new Square();  // ERROR
        Square square;
        square = box;
        box.print();
        System.out.println("Box: "+square.square());
        // square.print() // ERROR
        if (box instanceof Square)     {
            System.out.println("box implements square");
        }
    }
}
```

Интерфейсы. Example 30

Вложенные интерфейсы. Интерфейсы можно вложить (объявить членом) другого класса или интерфейса. В этом случае значение доступа может принимать значения **public, private, protected**. Когда вложенный интерфейс использует вне области вложения, то он используется вместе с именем класса или интерфейса.

```
public interface Square {
    double PI = 3.1415926;
    double square();

    public interface InnerSquare{
        double getInnerSquare();
    }
}
public class Test {
    public static void main(String[] args){
        Square.InnerSquare innerSquare;
    }
}
```

Интерфейсы. Example 31

Интерфейсы и обратные вызовы. Обратным вызовом (callback) называется набор инструкций, который выполняется всякий раз, когда происходит какое-либо событие, например, действие, выполняемое при нажатии кнопки. Рассмотрим пример Кея Хорстманна.

```
import   java.awt.* ;
import   java.awt.event.*;
import   java.util.*;
import   javax.swing.*;
import   javax.swing.Timer;
public class TimerTest{
    public static void main(String[] args) {
        ActionListener   listener   = new TimePrinter();
        // Создает таймер, вызывающий блок прослушивания
каждые   10  секунд.
        Timer t   = new Timer(10000,    listener);
        t. start() ;
        JOptionPane.showMessageDialog(null,"Выход?");
        System.exit(0);
    }
}
```

Интерфейсы. Example 31

```
class TimePrinter implements ActionListener {
    public void actionPerformed(ActionEvent event){
        Date now = new Date();
        System.out.println("Текущее время ... " + now);
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Интерфейсы

Существуют ИСТОЧНИК события, СЛУШАТЕЛЬ события и непосредственно само СОБЫТИЕ. С точки зрения Java все эти существа – объекты каких-то классов. На источник не налагается никаких ограничений, главное, чтобы он мог генерировать хоть какое-нибудь событие. Самими событиями являются объекты классов, расширяющих класс Event или его потомков. А вот слушатель события обязательно должен быть объектом класса, реализующего интерфейс Listener или его потомка. В примере класс реализует интерфейс ActionListener, где содержится всего один метод **public void actionPerformed(ActionEvent event)**.

Объект-слушатель события должен зарегистрироваться в объекте источнике-событии. При наступлении самого события источник вызывает метод actionPerformed() зарегистрированного слушателя, передавая ему ссылку на объект-событие.

Интерфейсы

Клонирование объектов. Интерфейс Cloneable. Рассмотрим ситуацию, когда в метод передается ссылка на объект и метод сам возвращает ссылку на объект.

Интерфейсы. Example 32

```
import java.util.Date;
public class MyDate {
    private Date date = new Date();
    public static void main(String[] args){
        MyDate mydate = new MyDate();
        Date date2;
        date2 = mydate.returnDate();
        mydate.printDate();
        System.out.println("Year: "+(date2.getYear()+1900)+" mounth:
"+date2.getMonth()+" day: « +date2.getDate());
        date2.setMonth(3);
        mydate.printDate();
        System.out.println("Year: "+(date2.getYear()+1900)+" mounth:
"+date2.getMonth()+" day: « +date2.getDate());
    }
    public void printDate(){
        System.out.println("Year: "+(date.getYear()+1900)+" mounth:
"+date.getMonth()+" day: « +date.getDate());
    }
    public Date returnDate(){
        return date;
    }
}
```

Результат программы следующий:

```
Year: 2008 mounth: 7 day: 3
Year: 2008 mounth: 7 day: 3
Year: 2008 mounth: 3 day: 3
Year: 2008 mounth: 3 day: 3
```

Результат не совсем тот. который ожидался. Через внешнюю ссылку **date2** изменилось внутреннее состояние объекта **mydate**. В такой ситуации возвращаемый объект необходимо клонировать, т.е. использовать метод **clone**().

Интерфейсы. Example 33

```
import java.util.Date;
public class MyDate {
…
     public Date returnDate(){
          return (Date) date.clone();
     }
…
}
```

Результат выполнения программы:

Year: 2008 mounth: 7 day: 3
Year: 2008 mounth: 7 day: 3
Year: 2008 mounth: 7 day: 3
Year: 2008 mounth: 3 day: 3

Интерфейсы

Метод **clone**() существует практически во всех библиотечных классах. Однако, в классах, разрабатываемых самим программистом, метод **clone**() наследуется из класса **Object**, который умеет копировать лишь поля. Следовательно, если вызывать унаследованный метод **clone**() опять получиться ситуация, когда разные ссылки указывают на один и тот же объект (такое клонирование называют поверхностным). Часто объекты содержат подобъекты (ссылки на другие объекты), и, чтобы получить их копию, необходимо переопределять метод **clone**() (выполнять глубокое копирование).

Интерфейсы

Для того, чтобы переопределить метод **clone**() необходимо реализовать интерфейс **Cloneable** и описать метод **clone**() с модификатором **public**.

Интерфейс **Cloneable** не содержит методов относится к помеченным (**tagged**) интерфейсам, а его реализация гарантирует, что метод **clone**() класса **Object** возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение **CloneNotSupportedException**.

Интерфейсы

Класс **Object** содержит protected-метод **clone**(), осуществляющий побитовое копирование объекта производного класса.

- Сначала необходимо переопределить метод **clone**() как **public** для обеспечения возможности вызова из другого пакета.
- В переопределенном методе следует вызвать базовую версию метода **super.clone**(), которая и выполняет собственно клонирование.
- Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс **Cloneable.**

Интерфейсы. Example 34

```
package _java._se._02.inheritance;
public class Student implements Cloneable {
     private int id = 71;
     public int getId() {
          return id;
     }
     public void setId(int value) {
          id = value;
     }
     public Object clone() { // переопределение метода
          try {
            return super.clone(); // вызов базового метода
          } catch (CloneNotSupportedException e) {
            throw new AssertionError("невозможно!");
          }
     }
}
```

Интерфейсы. Example 35

```
package _java._se._02.inheritance;
import java.util.ArrayList;
public class StudentDeepClone implements Cloneable {
    private int id = 71;
    private ArrayList<Mark> lm = new ArrayList<Mark>();
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public ArrayList<Mark> getMark() {
        return lm;
    }
    public void setMark(ArrayList<Mark> lm) {
        this.lm = lm;
    }
```

Интерфейсы. Example 35

```
  public Object clone() {
    try {
     StudentDeepClone copy = (StudentDeepClone)  super.clone();
     copy.lm = (ArrayList<Mark>)  lm.clone();
     return copy;
     } catch (CloneNotSupportedException e) {
     throw new AssertionError("отсутствует Cloneable!");
     }
  }
}

class Mark {
     private int mark = 3;
     public static int coeff = 5;
     public double getResult() {
          return (double) coeff * mark / 100;
     }
     public static void setCoeffFloat(float c) {
          coeff = (int) (coeff * c);
     }
     public void setMark(int mark) {
          this.mark = mark;
     }
}
```

Интерфейсы. Example 36

Сравнение объектов. Интерфейс Comparable. Метод sort(…) класса Arrays позволяет упорядочивать массив, переданный ему в качестве параметра. Для элементарных типов правила определения больше/меньше известны.

```
import java.util.Arrays;
public class SortArray {
    public static void main(String[] args){
        int[] mas = {3,6,5,1,2,9,8};
        printArray(mas);
        Arrays.sort(mas);
        printArray(mas);
    }
    public static void printArray(int[] ar){
        for(int i : ar) System.out.print(i+" ");
        System.out.println();
    }
}
```

Интерфейсы

Также этот метод упорядочивает и массив объектов при одном условии: объекты массива должны принадлежать классу, реализующему интерфейс **Comparable**. Класс, реализующий интерфейс Comparable должен иметь метод **compareTo()**. Естественно, вызов **x.compareTo()** должен действительно уметь сравнивать два объекта.

```
public interface Comparable {
     int  compareTo (Object other);
}
```

Интерфейсы. Example 37

```
import java.util.Date;
public class Book implements Comparable, Cloneable{

    private String title;
    private int yearPublished;
    private int price;
    private Date date;

    public Book() {}
    public Book(String title, int yearPublished, int price, Date date){
        this.title = title;
        this.yearPublished = yearPublished;
        this.price = price;
        this.date = date; // внимательно изучите эту строку
              // определите в ней потенциальную опасность
    }
    public String getTitle() { return title;     }
    public int getYearPublished() { return yearPublished; }
    public int getPrice(){ return price;}
    public void printReport(){System.out.println("Название: "+title+
         " год издания: "+year_published+" цена: "+price); }
```

Интерфейсы. Example 37

```
public int compareTo(Object object) {
        Book book=null;
        if(object instanceof Book)
            book = (Book)object;
        else {/*возбуждаем исключение*/}
        if ( price < book.price )
            return -1;
        else if ( price > book.price )
                return 1;
             else return 0;
    }
    public Object clone() {
        Book book = new Book();
        book.title = title;
        book.year_published = year_published;
        book.price = price;
        book.date = (Date)date.clone();
        return book;
    }
}
```

ВВЕДЕНИЕ В DESIGN PATTERNS

Введение в Design Patterns

Шаблон

- это идея, метод решения, общий подход к целому классу задач, постоянно встречающихся на практике
- систематизация приемов программирования и принципов организации классов

Основные принципы объектно-ориентированного проектирования, применяемого при создании диаграммы классов и распределения обязанностей между ними, систематизированы в шаблонах **GRASP**

(**General Responsibility Assignment Software Patterns**)

GRASP. CREATOR

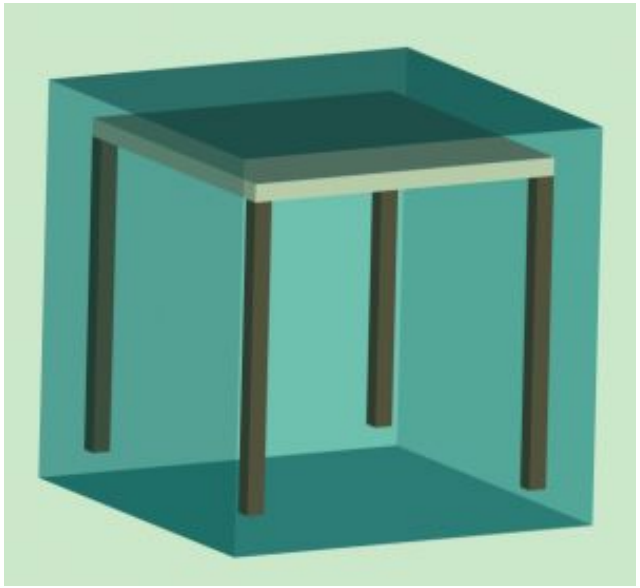
Наиболее частой проблемой ОО-дизайна является проблема “Кто должен создавать объект X?” Объект A должен создавать объект B если:

- A содержит или агрегирует B
- A записывает B
- A широко использует B
- A содержит данные инициализации, которые должны быть переданы при создании объекту B

Пусть есть стол. Самый обычный стол состоит из столешницы и четырех ножек.

Используя объектную декомпозицию мы получим три класса:

- Table :
- Desk
- Leg



Введение в Design Patterns. Example 38

Есть два пути запрограммировать такое решение. Сначала попробует не использовать шаблон Creator

```
package _java._se._01.pattern.creator.nocreator;
public class Table {
     private Desk desk;// desk object
     private Leg[] legs;// array of legs
     public Table(Desk d, Leg[] l) {
          desk = d;
          legs = l;
     }
}
```

Введение в Design Patterns. Example 38

```
package _java._se._01.pattern.creator.nocreator;
public class Desk {
    private int width;
    private int length;
    private int height;
    public Desk(int w, int l, int h){
            width = w;
            length = l;
            height = h;
    }
}
```

```
package _java._se._01.pattern.creator.nocreator;
public class Leg {
    private int width;
    private int length;
    private int height;
    public Leg(int w, int l, int h) {
        width = w;
        length = l;
        height = h;
    }
}
```

Введение в Design Patterns. Example 39

Так как столешница и ножка стола – оба параллелепипеды, то мы можем создать класс параллелепипед и унаследовать их от него, однако сейчас поговорим об создании объектов. Вышеупомянутые классы могут использоваться следующим образом.

```
package _java._se._01.pattern.creator.nocreator;
public class Main {
  public static void main(String[] args) {
    Desk desk = new Desk(900, 900, 20);
    Leg[] legs = { new Leg(40, 40, 880),
    new Leg(40, 40, 880), new Leg(40, 40, 880),
         new Leg(40, 40, 880) };
    Table table = new Table(desk, legs);
  }
}
```

Да, объект создается, однако значительная часть логики создания объекта остается за его пределами объекта. Программист, пользователь такого класса, должен знать внутреннюю структуру объекта, чтобы его создать. Давайте изучим, как решить такую задачу правильно

Конструктор класса **Table** должен содержать следующие параметры

- Width – ширина стола
- Length – длина стола
- Height – высота стола
- DeskHeight – высота столешницы
- LegSection – число ножек

Введение в Design Patterns. Example 40

```
package _java._se._01.pattern.creator.withcreator;
import _java._se._01.pattern.creator.nocreator.Desk;
import _java._se._01.pattern.creator.nocreator.Leg;

public class Table {
  private Desk desk;
  private Leg[] legs;
  public Table(int width, int length, int height, int deskHeight,
         int legSection) {
    desk = new Desk(width, length, deskHeight);
    for (int i = 0; i < 4; i++) {
      legs[i]=newLeg(legSection, legSection, height -
deskHeight);
    }
  }
}
```

GRASP. LOW COUPLING.

Пусть нам надо напечатать следующую таблицу параметров. В таблице существует два типа строк: square и circle. Square имеет параметр “side”, circle– “radius”.

Type	parameter	value
Circle	radius	5.00
Square	side	2.00
Square	side	8.00

Введение в Design Patterns. Example 41

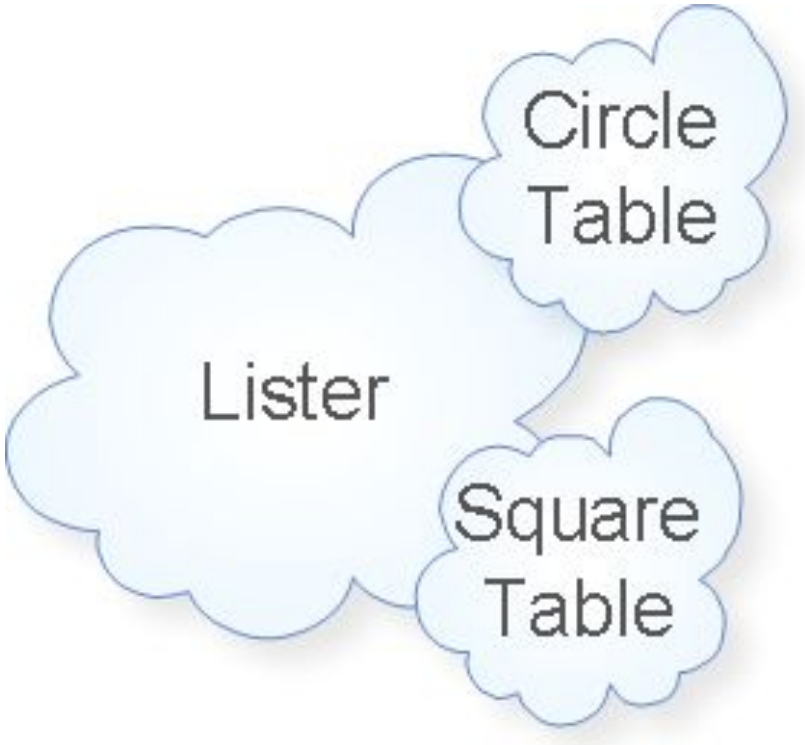
Рассмотрим решение такой задачи без применения шаблона LowCoupling

```
package _java._se._02.pattern.lowcoupling.nolowcoupling;
public class Lister {
 private Table[] tables;
 public void out() {
  System.out
     .println("+----+----+---+\n| Type   | parameter |  value  |\n+----+----+---+\n");
  for (int i = 0; i < tables.length; i++) {
     if (tables[i] instanceof CircleTable) {
       System.out.printf("|  Circle  |  radius   |  %5.2f  |\n",
       ((CircleTable)tables[i]).getRadius());
     } else {
       System.out.printf("|  Square  |  side     |  %5.2f  |\n",
       ((SquareTable) tables[i]).getSide());
     }
     System.out.println("+----+----+---+\n");
  }
 }
}
```

Введение в Design Patterns. Example 41

```
class Table{}

class CircleTable extends Table{
    private double radius;
    public CircleTable(int r) { radius = r; }
    public double getRadius() { return radius; }
}

class SquareTable extends Table{
    private double side;
    public SquareTable(int s) { side = s; }
    public double getSide() { return side; }
}
```

Classes coupling



Введение в Design Patterns. Example 42

```
package _java._se._02.pattern.lowcoupling.withlowcoupling;

public class CircleTable extends Table {
     private int radius;
     public CircleTable(int r, int h) {
          radius = r;// setting the radius
     }
     public double getSquare() {
          return radius * radius * Math.PI;
     }
     public void out() {
          System.out.printf("|  Circle
               |  radius   |  %5.2f  |\n", radius);
     }
}
```

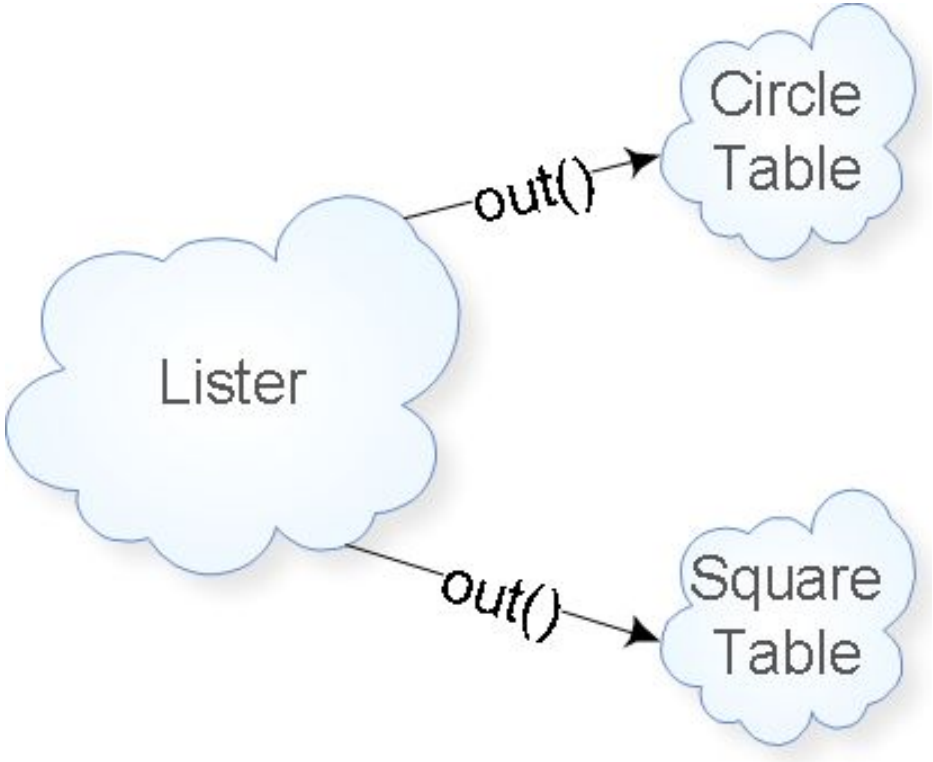
Введение в Design Patterns. Example 42

```
package _java._se._02.pattern.lowcoupling.withlowcoupling;

public class SquareTable {
     private int side;
     public SquareTable(int s, int h) {
          side = s;// setting the side
     }
     public double getSquare() {
          return side * side;
     }
     public void out() {
          System.out.printf("|  Square
               |  side     |  %5.2f  |\n", side);
     }
}
```

Введение в Design Patterns. Example 42

```
package _java._se._02.pattern.lowcoupling.withlowcoupling;
public class Lister {
    private Table[] tables;

    public void out(){
    System.out.println("+————-+————+———+\n|  Type
        | parameter |  value  |\n+ ————-+————+———+\n");
    for (int i = 0; i < tables.length; i++){
    tables[i].out();
    System.out.println("+————-+————+———+\n");
        }
    }
}
```

Low coupling illustration



Введение в Design Patterns

Преимущества использования шаблонов:

- Нет необходимости решать каждую задачу с нуля
- Использование проверенных решений
- Можно заранее представить последствия выбора того или иного варианта
- Проектирование с учетом будущих изменений
- Компактный код, который легко можно будет использовать повторно

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Java.SE.02

Object-oriented programming in Java

Author: **Ihar Blinou, PhD**
Oracle Certified Java Instructor
Ihar_blinou@epam.com