Тема 5. Програмни елементи.

доц. д-р инж. Владимир Николов

17.3.2020 г.

Съдържание

- Конструктори ;
- Модификатори за нива на достъп;
- Пакетен спецификатор на достъп;
- Деструктори;
- Предекларация на методи;
- Функции и методи;
- Предефиниране на оператори;
- Автоматични промени на типове;
- Указатели и референции;
- Абстрактни класове;

Конструктори

Java има конструктори, подобни на конструкторите в C++.

Видове:

- Подразбиращ се конструктор

  Може да се дефинира подразбиращ се конструктор и ако не се дефинира такъв, същият се създава автоматично. **Ако не се дефинира друг конструктор, той не се образува автоматично, както в C++;**
- Експлицитен конструктор. Може да се създаде при необходимост от програмиста с един или повече аргумента;
- Няма необходимост от копиращ конструктор, тъй като всички аргументи се предават чрез референция.

Конструктори

Принципи при създаването на обектите:

- Обектите в Java се алокират в системната динамична памет "heap". За разлика от C++ не е необходимо да се освобождава заетата от програмиста динамична памет. Заемането на динамичната памет става с експлицитно извикване на оператора на Java new;
- Освобождаването се прави когато обектът не се реферира от кода с автоматичния garbage collection;
- Обектите се създават с извикването на метода конструктор по същия начин, както в C++;
- Конструкторите могат да са експлицитни и подразбиращ се, защото създаването чрез копиращ конструктор е без смисъл.

Модификатори за нива на достъп

Модификатори за нива на достъп и контрол на достъпа до членовете.Особености и разлики със C++:

- Няма блокове за контрол на достъпа от C++ - спецификаторите за достъп-ключови думи (**public,private и protected**) се поставят пред всяка дефиниция на член на класа;

- **Подразбиращ се модификатор за достъп е пакетен**: **в рамките на разработвания пакет** и недостъпни извън пакета - подобно на приятелската декларация в C++ (**friend**).

Модификатори за нива на достъп

Модификатори за нива на достъп и контрол на достъпа до членовете

- Задължително **всеки клас, поле и метод** на класа трябва експлицитно да се определя като достъпен извън файла. (при много-нишковите се изисква използването на ключова дума **private**).
- Достъпът с атрибут **private** се използва рядко- пакетен достъп е по-често приложим от частния достъп:
  - изключва достъпа от другите класове на същия пакет;
- Комбинацията на двете- **private за клас и protected за членове** се игнорира при компилация).

Модификатори за нива на достъп

В Java трябва да се зададе за всяка декларация поотделно:

```
public class A {
    public int x;
    public int y;
    private float v;
}
```

- По този начин модификаторите за достъп в Java не са етикети, а истински модификатори.
  - Конвертирането на модификаторите от код на C++ се състои в преминаване през декларациите и поставяне на етикет (модификатор на достъпа) за всяка декларация поотделно.

Приятелски (пакетен) спецификатор на достъп

- **Този Java модификатор се приема за модификатор по подразбиране. Може да се разглежда като реферирaн приятелски модификатор за достъп.**
  - Той има ефекта да позволява достъп на всички класове в същия пакет като на текущия клас до съответния член на класа, деклариран с модификатора по подразбиране;
  - За да се даде на определена променлива или метод подобен тип достъп се пропуска спецификатора за достъп. Пример:

```
class A {// пакетен достъп
  public int iPublic;
  private int iPrivete;
  int iPacket; // пакетен достъп
  int jPacket; // пакетен достъп
}
```

Деструктори

В Java няма деструктори.

Следствия:

- Не съществува обхват на променливите, който да определя края на времето на живот на променливите, създадени от програмиста;
- Животът на динамичните променливи се определя от менажера на паметта (garbage collector). Работата с паметта се разглежда в следващите лекции;
- Съществува метод **finalize,** който е член на всеки клас. Той е подобен на деструктора на C++, но се извиква от garbage collector и е предназначен да освобождава други ресурси (файлове, връзки (sockets), канали за връзка (ports), URLs и др.);

(Продължава)

Деструктори

- Ако има нужда от действия при приключване в дадена точка, те се реализират в методи, които се извикват на определеното място в кода на програмата, без да се разчита на finalize( ). Всички обекти в C++ ще бъдат освободени при излизане от обхват, но не всички обекти в Java се освобождават от менажера на паметта в този момент;
- Тъй като Java не поддържа деструктор, при необходимост вместо него може да се създаде метод за имплицитно освобождаване на базовия клас и членовете на класа.

Предекларация на методи

Методите се предекларират в Java по начин, подобен на този, използван за функциите в C++. Разликата е в липсата на подразбиращи се аргументи, които се използват понякога в C++ за предекларация.

Например, C++ конструктор с обща дефиниция, който **нямат еквивалент в Java**:

```
CStudent ( const string& name="", const string& facn="", int points=0){
    m_strName=name; m_strFacNum=facn; m_iMinutes=points;
}
```

Функции и методи- **Особености**

В С кодът е организиран като функции, които могат да са и **глобални функции**, достъпни за цялата програма.

Разширението (C++) добавя и класове като по този начин предоставя **методи-функции, свързани с класове.** Методите на класове в C++ са подобни на тези в Java. Поради съвместимостта към С, не може да ограничат програмистите на C++ да използват глобалните функции. Резултатът е смесица от използване на функции и методи, което прави програмите на C++ понякога неясни (кога коя от функциите ще се изпълни).

Функции и методи- **Особености**

Особености на Java**:**

- Няма **глобални** (извън класовете) функции. Като по-чист обектно-ориентиран език от C++, Java принуждава програмистите да свързват всички под-програмни обръщения с класове:
  - Не се ограничават възможностите, а само се подобрява организацията на кода;
  - Използването на глобалните функции не е грешно, но те не се вписват добре в концепцията за обектно-ориентирано програмиране, която е залегнала в ядрото на Java.

Функции и методи- **Особености**

Преобразуването на глобалните C++ функции налага някои организационни промени:

Пример:

```
char EncryptChar(char c, int key);
char DecryptChar(char c, int key);
char* EncryptString(const char* s, int key);
char* DecryptString(const char* s, int key);
```

Функции и методи- **Особености**

Включване в клас:

```
public class Crypt {
    public static char encryptChar(char c, int key) {
      // код за криптиране на символ
    }

    public static char decryptChar(char c, int key) {
      // код за декриптиране на символ
    }

    public static String encryptString(String s, int key) {
      // код за криптиране на низ
    }

    public static String decryptString(String s, int key) {
      // код за декриптиране на низ
    }
}
```

Промяна на типовете, които не се поддържат (char *)

Функции и методи- **Особености**

Основни разлики:

- Като се декларират като public static, те се правят видими от цялото приложение на Java. Основната разлика на Java е, че имплементациите на функциите са дефинирани в класове на Java, защото не се поддържа организацията на хедерни (*.h) и сорс (*.cpp) файлове. Цялата информация за класа се включва директно в неговата дефиниция;
- Стандартната **конвенция за именуване** в Java е методите да започват с малки букви. За да се използват методите, те трябва да се реферират заедно с класа си:

```
char c = Crypt.encryptChar('a', 7);
```

Функции и методи- **Особености**

Основни разлики:

- Промяна в кода на С функциите при преминаване на Java е използването на класовете String вместо указателите към символни масиви:
  - Java не поддържа указатели, а само референции;
- Java не поддържа подразбиращи се аргументи на функциите;

Функции и методи- **Особености**

- В Java не се използва ключовата дума **virtual** за функциите, защото всички не статични методи се свързват динамично. Затова не е нужно да се специфицира този метод на свързване изрично от програмиста:
  - Предназначение на **virtual** в C++ :
    - Да се управлява (при необходимост) ефективността на програмата като при използването и се получава определено намаляване;
    - В кода, където не се използва, управлението на ефективността се осигурява със статични методи;

Функции и методи- **Особености**

- В Java може да се използва ключовата дума **final**, която указва на компилатора, че методът **не може да се предекларира в наследника**, поради което същият трябва да се свърже статично и да се използва по подобен начин като inline методите на C++ и не-виртуални обръщения. Тези оптимизации са задължителни за компилатора.

Функции и методи- **Особености**

- **Конвертиране от процедурно към ООП**:

```
public class Crypt {
      private int key;// добавяне на поле

      public Crypt(int k) { // добавяне на конструктор
        key = k;
      }

     public  void setKey(int k) {
       key = k;
      }

      public int getKey() {
       return key;
      }

      public char encryptChar(char c) {
        // код за криптиране на символ
      }

      public char decryptChar(char c) {
        // код за декриптиране на символ
      }

      public String encryptString(String s) {
        // код за криптиране на низ
      }

      public String decryptString(String s) {
        // код за декриптиране на низ
      }
}
```

Предефиниране на оператори

Не се поддържа от Java.

- Предимства на предефинирането:

  синтактичното удобство, пренасяне на семантиката операторите на езика върху обектите;
- Недостатък: понякога може да направи неясни програмите, които ги използват.

Решение в Java- **функционалността на операторите да се имплементира като методи в класовете на Java.**

- Причина: Разработчиците на Java са решили да не се поддържа, за да се запази езика колкото може по-прост.

Предефиниране на оператори

Степен на използване в C++:

- Най-вече в STL и алгоритмите. **По-базовите C++ класове като string, използват много често сравнения с оператори, което затруднява прехвърлянето**.
- Преработване на C++ код в Java много зависи от това до каква степен кодът зависи от тази особеност на езика.

В Java възниква необходимостта от заместител, който да осигури алгоритмите с оператори.

Предефиниране на оператори

- **Заместител на C++ операторите:**

Единствения начин за превод на предефинирани C++ оператори на Java е да се създадат методи със същата функционалност.

- Методите на Java получават различни имена от предефинираните оператори;
- Това означава, че ще е нужно внимателно да се прегледа кода, да се определи на кои точно места се използват тези оператори и да се преобразуват в извикване на методи.

Предефиниране на оператори (C++)

Пример : Клас за представяне на комплексни числа с предефинирани оператори на C++.

```
class Complex { //C++ код на класа
      float real;
      float imag;
public:
      Complex(float r, float i);

      Complex operator+(const Complex& c) const {
        return Complex(real + c.real, imag + c.imag);
      }

      Complex operator-(const Complex& c) const {
        return Complex(real - c.real, imag - c.imag);
      }
};
```

Предефиниране на оператори (C++)

Изваждането на два обекта от тип Complex е синтактично еднакво с изваждането на две променливи от прости типове (C++ код) :

```
Complex c1(3.0, 4.0);
Complex c2(5.0, 2.5);
Complex c3 = c2 - c1;
```

Тази възможност на езика C++ предизвиква сравнително голямо усложняване на езика Java, която разработчиците искат да избегнат. Разликата е в това, че не може да се предостави същия синтактичен еквивалент на C++ кода в Java. Същата функционалност може да се имплементира с методи.

Предефиниране на оператори в Java

Пример за решение на Java: предефинираните оператори са заменени с методи:

```
public class Complex {
      private float real;
      private float imag;

      public Complex(float r, float i) {
        real = r;
        imag = i;
      }

       public Complex add(Complex c) {
        return (new Complex(real + c.real, imag + c.imag));
      }

       public Complex subtract(Complex c) {
        return (new Complex(real - c.real, imag - c.imag));
      }
}
```

Предефиниране на оператори в Java

Промяната във версията на Java е в дефинирането на методите за операторите - add и substract. Класът Complex се използва по следния начин:

```
Complex c1 = new Complex(3.0, 4.0);
Complex c2 = new Complex(5.0, 2.5);
Complex c3 = c2.subtract(c1);
```

Вижда се, че операцията изваждане не е толкова интуитивна когато се ползва подхода на Java, в сравнение със C++. Въпреки това тя е подобна.

Автоматични промени на типове

Същност: Автоматичните подмени на типове представлява подмяната на типове (type casting), което се получава понякога и автоматично в C/C++. Като пример, в C++ е позволено да се присвои реално число на променлива, декларирана като цяло число, което би довело до загуба на информация.

Java не поддържа подобна автоматична промяна на типовете. В Java, ако вследствие от типово преобразуване се получи загуба на информация, се изисква експлицитно сменяне на типа към новия тип.

Автоматични промени на типове

Пример за автоматична подмяна на типа в C++:

```
float f = 3.1412;
int i = f;
```

Някои компилатори на C++ биха генерирали предупреждение в този случай, но това не се счита за грешка. В Java, се генерира грешка при компилация. Тя се поправя с експлицитно подменяне на типа:

```
float fVar = 3.1412;
int i = (int)fVar;
```

Указатели и референции

Указателите в Java:

- **Java няма указатели, като обектите се предават индиректно, с използване на референции, вместо да се предават указатели към тях;**
- **Достъп до елементите на масивите се прави само с помощта на индекси:**
  - **Не могат да се подменят с изчислим достъп с указатели!**

Указатели и референции

Степен на използване на указателите в C++:

- Допускат разработка на системи, в които се използват указатели към функции или указатели към член-функции, например за динамично пренасочване от функция с общ основен тип (void*) и множество от методи с една и съща спецификация към различни тела за изпълнение;
- Често програмистите на C++ използват математически изрази (аритметика) с указателите, за да поддържат сложни типове данни;
- Липсата на указатели лишава програмиста от възможността да създава собствени структури от данни, като например динамично разширяващи се масиви.

Указатели и референции

Недостатъци при използване на указателите при програмиране на C++:

- Повечето от грешките в C/C++ програмите се причиняват от неправилно използване на указатели.
  - Практиката на програмирането на C++ показва, че когато се използват указатели, съществува възможността да се прави невалиден достъп до паметта;
  - В резултат на това, могат да се допуснат трудни за откриване грешки, причинени от използването на **аритметика** с указатели.

Указатели и референции

- **Заместител на указателите в Java:**
  - Има подобна функционалност чрез използване на референции;
  - Java предава всички обекти и масиви чрез техните референции. Този подход елиминира грешки при достъпа чрез указатели. Референциите също така улесняват програмирането, защото правилното използване на указатели е много по-трудно, освен за най-опитните програмисти;
  - Получава се ефекта от сигурността, предоставена от системата за изпълнение в реално време на Java. Например, тя извършва проверка за валиден размер на всички операции за индексиране на масиви.

Указатели и референции

Указателите и преобразуването на код от C/C++ на Java.

Най-трудната част за превеждане на код от C/C++ на Java са указателите. Основни стъпки:

- Първо се подменят всички масиви от символи в Java низове (String). В резултат от реализирането на това много от кода, зависим от указатели се премахва;

Указатели и референции

Указателите и преобразуването от C/C++ на Java.

- Конвертирането на кода за създаването и унищожаването на обекти.

  Последователност на работа с динамичната памет при програмиране на C++:

  - Създаване на обекти и получаване на указатели към тях с използването на оператора new. Полученият адрес в динамичната памет се присвоява на указателя;
  - Работа по получената динамична памет чрез указателя;
  - След като е приключена работата с обекта, се извиква оператора delete, за да се освободи паметта.

  Тази процедура е подобна и при Java **без последната стъпка**.

Указатели и референции

Указателите и преобразуването на код от C/C++ на Java.

- Логична стъпка при прехвърлянето на код от C++ към Java е да се подменят указателите в C++ кода с референции, преди да се започне работата по прехвърлянето. Трябва да се преработи C++ кода изцяло на референции и тогава да се започне работата на Java.

Указатели и референции

Пример за създаване **на референция** към низ:
String s = new String(“тестов низ”);

- Когато се създават обекти от класове на Java с оператора **new**, резултатът е референция (или *handle* в паметта);
- Java обекти, които не се използват, се освобождават автоматично от системата на Java garbage collection, която обикновено е системна нишка с нисък приоритет;
- Тъй като системата се грижи за освобождаването на неизползваните обекти, не е нужно програмистът да освобождава паметта, която заема.

Указатели и референции

Пример за преобразуване на код с указатели:

```
// C++ клас CMemTest.
    class CMemTest {
      MemBlock* pMem;

      CMemTest() {
        pMem = new MemBlock();
      }

      ~CMemTest() {
        if (pMem != NULL) {
          delete pMem;
          pMem = NULL;
        }
      }
    };
```

Указатели и референции

Пример Java клас CMemTest:

```
public class CMemTest {
      private MemBlock pMem;

      public CMemTest() {
        pMem = new MemBlock();
      }
}
```

Предимства на Java кода:

- По-кратък;
- Променливата pMem не е нужно да се инициализира с NULL – всички член-променливи, ако са създадени без да им е присвоена стойност, се инициализират със стойности 0 или null;
- Менажера на паметта (Garbage collector) на Java се грижи за освобождаването на обекта pMem, когато той не се използва.

Указатели и референции

Особености:

- Използването на метода **finalize , който** се извиква от менажера на паметта (garbage collector) при унищожаването на обекта, не гарантира неговото извикване, поради спецификата на garbage collector;
- Указателите в Java се използват масово, но като **референции**. Референциите на Java са по-скоро указатели;
- Когато се свикне с ограниченията на референциите, се вижда, че няма съществена разлика между тях.

Указатели и референции

Разлики между референциите на C++ и Java

- За разлика от C++, където референциите трябва да се инициализират при обявяването им, т.е. те не могат да останат не присвоени, в Java това не е задължително;
- Референциите в C++ не могат да се променят след тяхното създаване. Изискване за непроменливост в Java няма, което доближава референциите на Java до указателите;
- Възможността за дефиниране на неприсвоена референция също доближава референциите по-скоро към указателите;
- В C и C++ указателите могат да адресират памет, независимо какво сочи. За неопитните програмисти това може да е източник на грешки. В Java адресирането на незащитен обект е недопустимо.

Указатели и референции

Разлики между референциите на C++ и Java (продължение)

- Указателите са ефективен начин за адресиране на масиви от прости типове данни. Масивите в Java допускат да се прави такава адресация при това по начин, защитен от грешки.
- Предаването на указатели към методи не е проблем, защото няма глобални функции, а само класове и те могат да предават референции към обектите. В този смисъл, референциите в Java се разглеждат като "ограничени указатели". Ограничението засяга основно липсата на изчисления с указателите, която няма еквивалент в Java;
- Създаването на указатели е допустимо в методи, наречени native методи;

Множествено наследяване

Същност:

Множественото наследяване е свойство на C++, което позволява наследяването на повече базови класове. Подобни конструкции са сложни за реализация от компилатора. Въпреки че е доста мощно средство, множественото наследяване трудно се ползва правилно и може да доведе до грешки. Тъй като Java е основана на модел с общ корен в йерархията множественото наследяване не се вмества в данновия му модел.

Множествено наследяване

Java заобикаля и не предоставя директна поддръжка за множествено наследяване. Заместител:

- **Могат да се декларират интерфейси, които описват програмните връзки, за осъществяване на свързана функционалност.**
- **Интерфейсите на Java предлагат методи на обекти, но не предлагат тяхната имплементация.**
- **Класът може да имплементира един или повече интерфейси, като собствена уникална функционалност.**
- **Различни, несвързани класове могат да имплементират еднакъв интерфейс.**

С помощта на интерфейсите се реализира възможността за релационна връзка (да имат подобно поведение) между два независими класа!

Множествено наследяване

- Формалните параметри на методите могат да се декларират или като класове или като интерфейси. Ако са интерфейси, обектите на различни класове, които имплементират интерфейса могат да бъдат предавани като аргументи към метод.
- Концепцията на интерфейсите е значително опростена за създаване на множествено наследяване, но има ограничения. Като минимум е необходимо да се напише код, който да повтаря исканата функционалност във всеки от класовете, имплементиращи един интерфейс.

Множествено наследяване

Пример на C++, който използва множествено наследяване:

```
class InputDevice : public Clickable, public Draggable {
      // дефиниция на клас
};
```

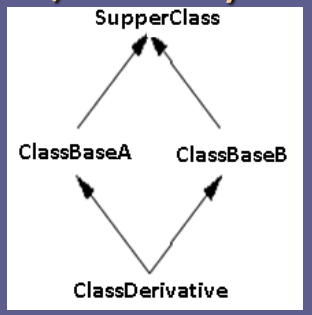
Най-близката имплементация в Java е създаването на интерфейси Clickable и Draggable, които могат да имат декларации на методи, но няма същински код на методите, както и нестатични член-променливи. Наследникът на интерфейсите-класът InputDevice може да имплементира тези интерфейси като използва ключовата дума implements:

```
public class InputDevice implements Clickable, Draggable {
  // дефиниция на класа
}
```

За да се направи прехвърлянето, трябва да подменят същинските тела на методите и да се имплементират в производни класове.

Множествено наследяване

С използване на вградени обекти (композиция) от класовете, които участват в множественото наследяване:




```
public abstract class SuperClass {
    public abstract void doSomething();
}
```

```
public class ClassBaseA extends
SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething
A");
    }
    //ClassBaseA own method
    public void methodA(){

    }
}
```

```
public class ClassBaseB extends
SuperClass{

    @Override
    public void doSomething(){
        System.out.println("doSomething
B");
    }
    //ClassBaseB own method
    public void methodB(){

    }
}
```

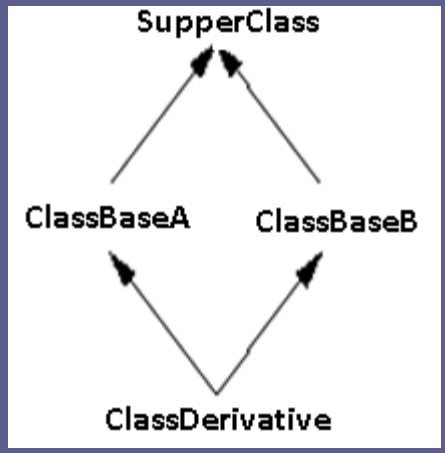
Множествено наследяване

С използване на вградени обекти (композиция) от класовете, които участват в множественото наследяване:

```
public class ClassDerivative{

    ClassBaseA objA = new ClassBaseA();
    ClassBaseB objB = new ClassBaseB();

    public void doSomethingFromA(){
        objA.doSomething();
    }
    public void doSomethingFromB(){
        objB.doSomething();
    }
    public void methodA(){
        objA.methodA();
    }
    public void methodB(){
        objB.methodB();
    }
}
```



Наследяване на абстрактни класове

Синтаксис на наследяването:

При C++:

```
class B : public A {
    // дефиниция на класа
  };
```

За дефиниране на наследяване в Java се използват ключовите думи **extends** (за класове) и **implements (за интерфейси)**:

```
public class B extends A {
    // дефиниция на класа
  }
```

Наследяване на абстрактни класове

Пример за наследяване на абстрактен клас:

Точка: CPoint;

Цвят: CColor;

Фигура: CShape, Дефинира цвят и позиция (чрез Точка) на абстрактна фигура. Обектите са атрибути на класа;

Кръг: CCircle – наследник на фигура, дефинира радиуса;

Правоъгълник : CRectangle – наследник на фигура, дефинира височина и ширина;

Наследяване на абстрактни класове

```
public class CPoint
{
// Атрибути
private double myX;
private double myY;
// Конструктори
    public CPoint(double x, double y) {
        myX = x; myY = y;
    }
    public CPoint() {
        myX = 0.; myY = 0.;
    }
// Методи
    public double getX() {
        return myX;
    }
    public double getY(){
        return myY;
    }
    public void setPoint(double x, double y) {
        myX = x; myY = y;
    }
}
```

Наследяване на абстрактни класове

```
public class CColor {
// Атрибути
     private     int red;
     private     int green;
     private     int blue;
// Конструктори
     public  CColor()       { red = 255; green = 0; blue = 125; }
     public  CColor(int r, int g, int b)      { red = r; green = g; blue = b; }
// Методи
     public  static long RGB2Long(int r, int g, int b) {
        return (r | (g<<4) | (b<<8)); //red+16*green+256*blue;
       }
     public  int getR() { return red; }
     public  int getG() {return green;}
     public  int getB() { return blue; }
     public void setRGB(int r, int g, int b)      {
            red = r; green = g; blue = b;
     }
}
```

Наследяване на абстрактни класове

```
public  abstract class CShape{
        private CColor color; //цвят
         private CPoint origin;  //позиция
        protected CShape(final CColor col, final CPoint org) {
           origin = new CPoint(org.getX(), org.getY());
           color = new CColor(col.getR(),col.getG(), col.getB());
        }
        protected CShape(final CPoint org) {
          origin = new CPoint(org.getX(), org.getY());
          color = new CColor(0,0,0); //  черен
        }
        protected CShape() {
          origin = new CPoint(0.0, 0.0); // позиция 0.,0.
          color = new CColor(0,0,0); // черен
         }
         public abstract double area(); // реална функция - в наследника
         public abstract double perimeter(); // реална функция - в наследника
         public CColor getColor(){
          return color;
         }
         public void setColor(CColor obj){
           color = obj;
         }
}
```

Наследяване на абстрактни класове

```
public class CCircle extends CShape
{
    private double radius; // Атрибут
// Конструктори
    public CCircle()       {
        super();
        radius = 0.0;
    }
    public CCircle(final CColor col, final CPoint org, double rad) { // константни
        super(col, org);
        radius = rad;
    }
// Методи
    public double area() {
        return Math.PI * radius * radius; // import java.lang.Math;  (PI)
    }
    public double perimeter() {
        return 2 * Math.PI * radius; // 2 PI r достъп до библиотека (пакет)
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double r) {
        radius = r;
    }
}
```

Наследяване на абстрактни класове

```
// Създаване на CRectangle  - необходими за предефиниране методи в него
    public double area(){
        return height * width;
    }
    public double perimeter() {
        return 2 * (height + width);
    }
// главна функция:
     public static void main(String[] args) {
         CCircle obj = new CCircle(new CColor(),new CPoint(),1.0);
         double area=obj.area();  // получаване на площ за кръг
         double per =obj.perimeter(); // получаване на периметър за кръг
          CRectangle = new CRectangle(new CColor(), new CPoint(), 1.0, 2.0);
     }
}
```

ВЪПРОСИ?