

# **Тема 6**

**Пример за използване на  
списъци: програмна система за  
манипулиране на картинки**

# Манипулиране на „картинки“ (Pictures: A Case Study)

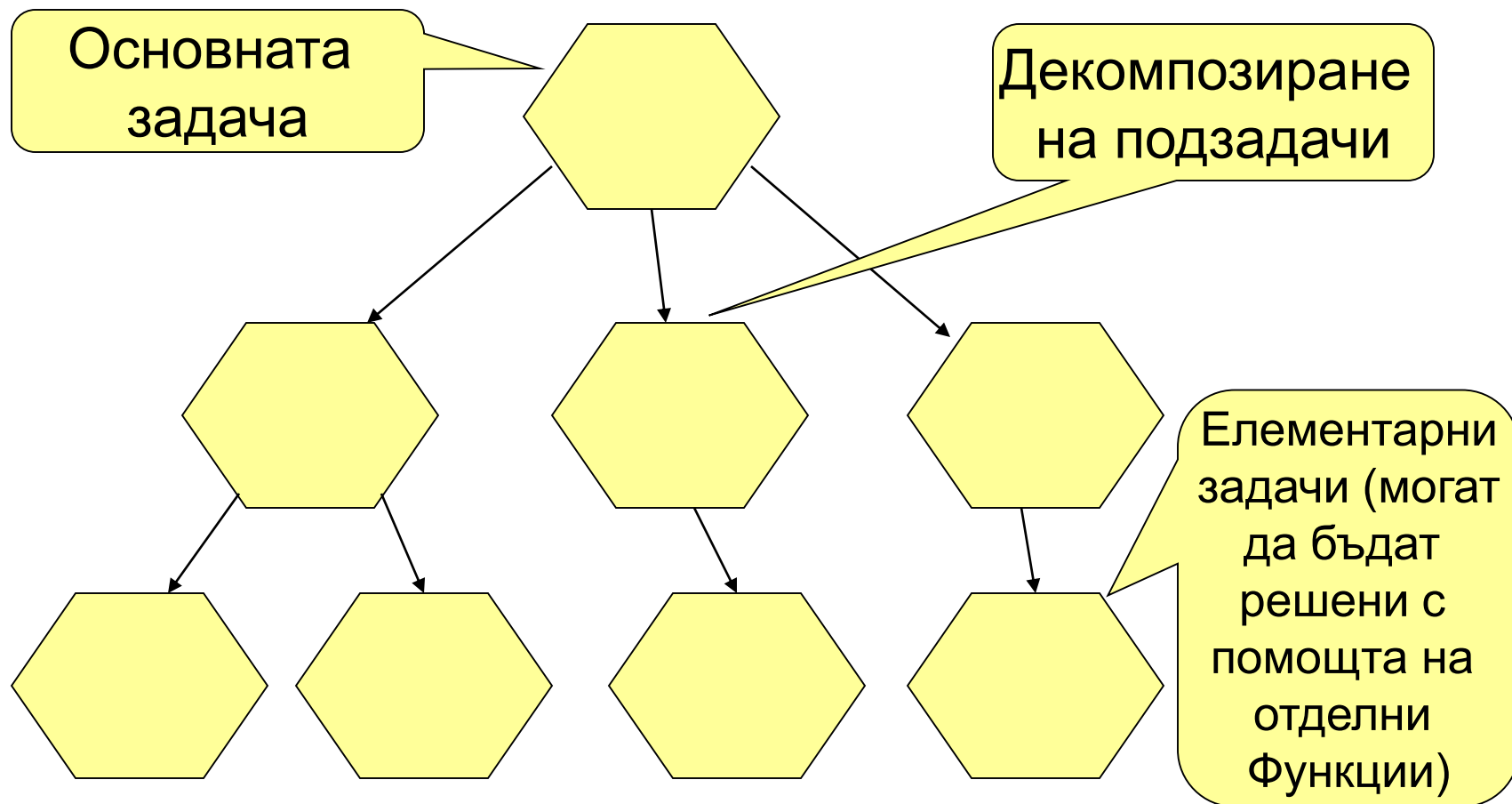
## Цели на проекта

Ще разгледаме един по-обширен пример за проектиране на част от програмна система: библиотека от функции за манипулиране на „картинки“.

## С каква цел?

- Разглеждане на примери за програмиране със списъци.
- Упражняване на работата със знакове (данни от тип Char).
- Анализ на основните подходи при проектиране на програмни системи.

# Низходящо проектиране (Top-Down Design)

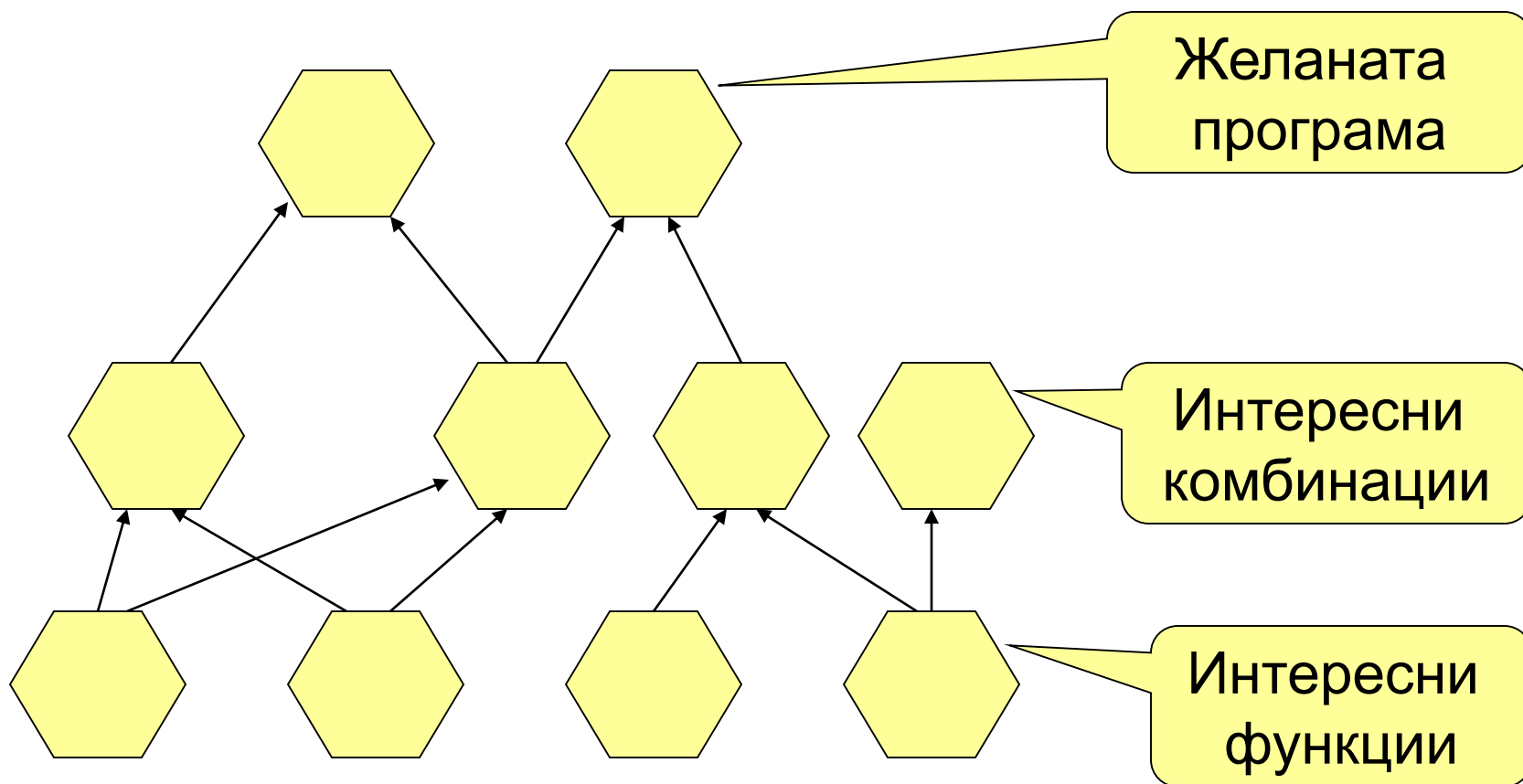


# Низходящо проектиране

*Какви функции биха били полезни за решаването на основната задача?*

- Декомпозиране на дадената сложна задача на обозрими подзадачи – ефективна стратегия за решаване на задачи.
- Работата е насочена директно към решаване на основната задача – избягване на излишно за конкретния случай програмиране.
- Не се очаква получаване на по-общи функции като страничен резултат.

# Възходящо проектиране (Bottom-Up Design)

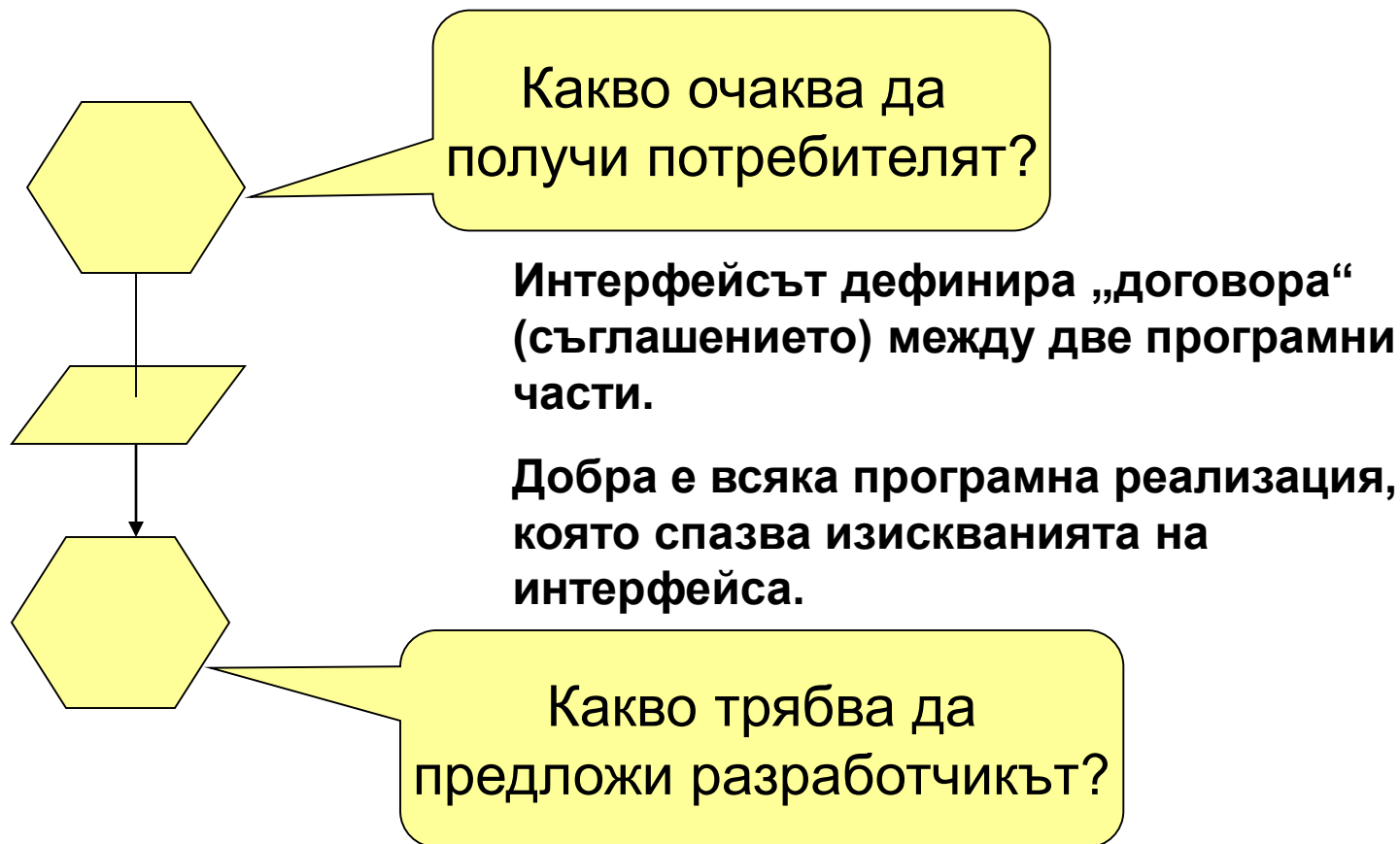


# Възходящо проектиране

*Какво би могло да се постигне  
с помощта на достъпните функции?*

- Ефективен начин за изграждане на програмни компоненти с голяма изразителна мощност.
- Работата се насочва към създаването на полезни функции, които ще могат да бъдат използвани (reused) в много случаи.
- Решение, което очаква своята задача!

# Интерфейси (Interfaces)



# Пример: Картинки

*Top-down:* Поставена е задача за създаване на програмна система за манипулиране на правоъгълни картинки.

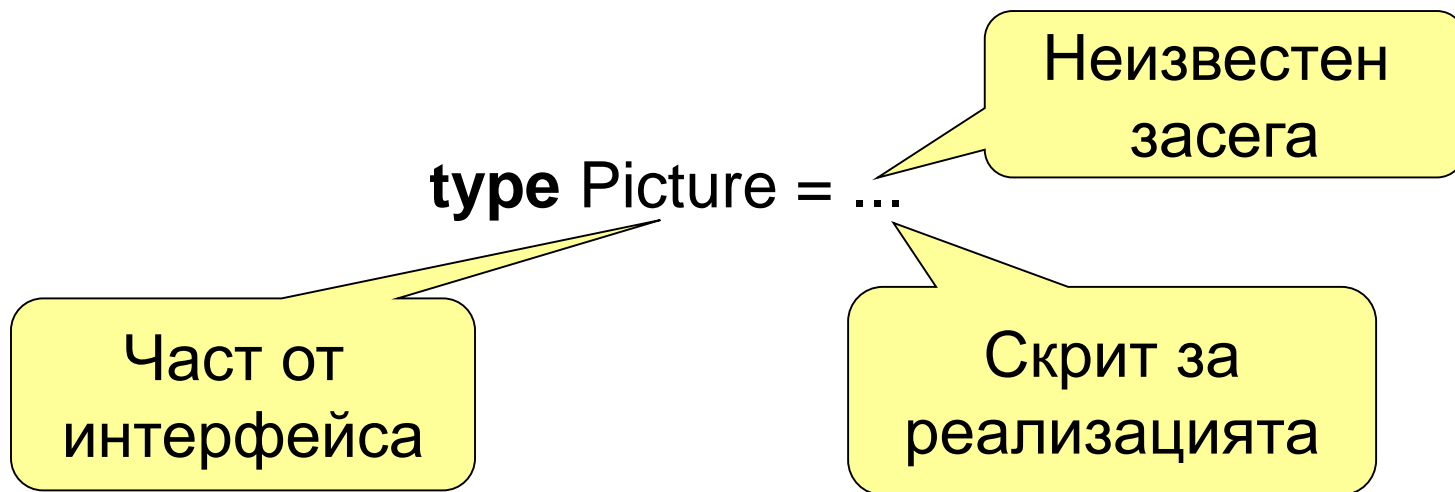
*Bottom-up:* Какви интересни функции можем да предвидим?

*Interface:* Какъв трябва да е „договорът“ между потребителите и разработчиците на програмната система?



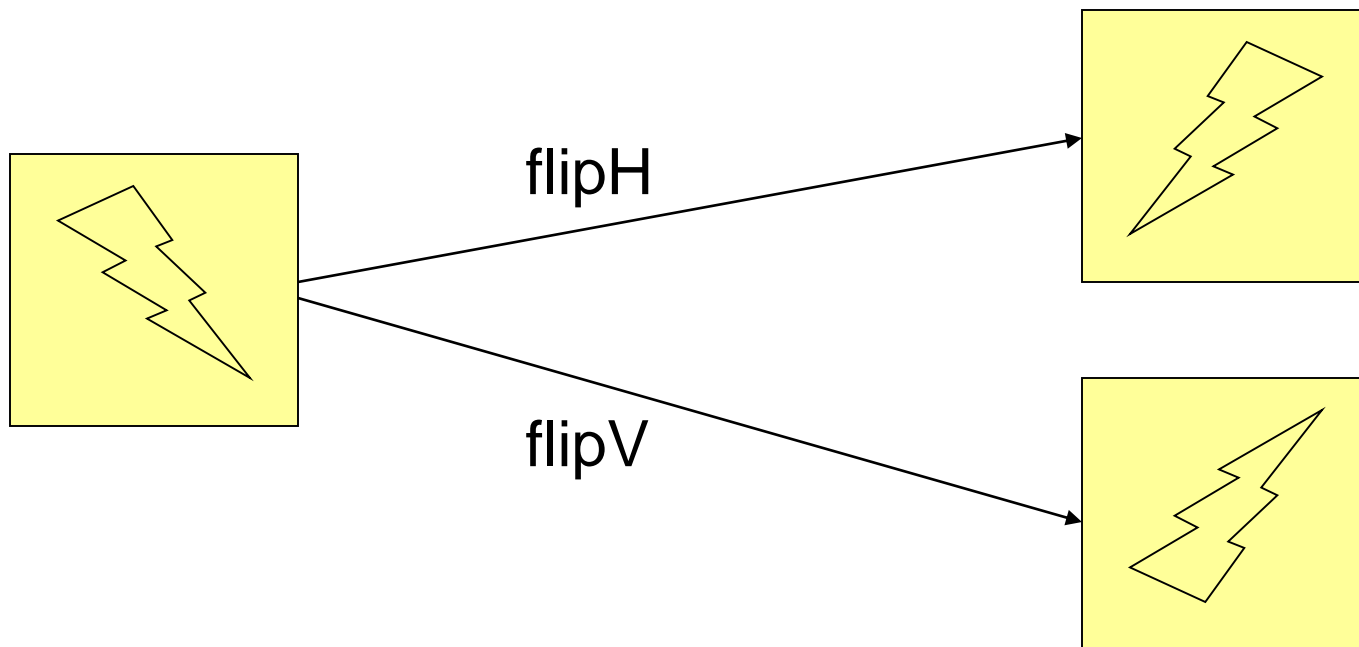
# Тип на картинките

Картинките са вид данни; следователно, необходимо е да се дефинира специален *тип* за тях.



Всеки тип, чието име е открито (public), но представянето му е скрито (private), се нарича *абстрактен тип данни* (*abstract data type*).

# Функции за работа с (манипулиране на) картинки



`flipH, flipV :: Picture -> Picture`

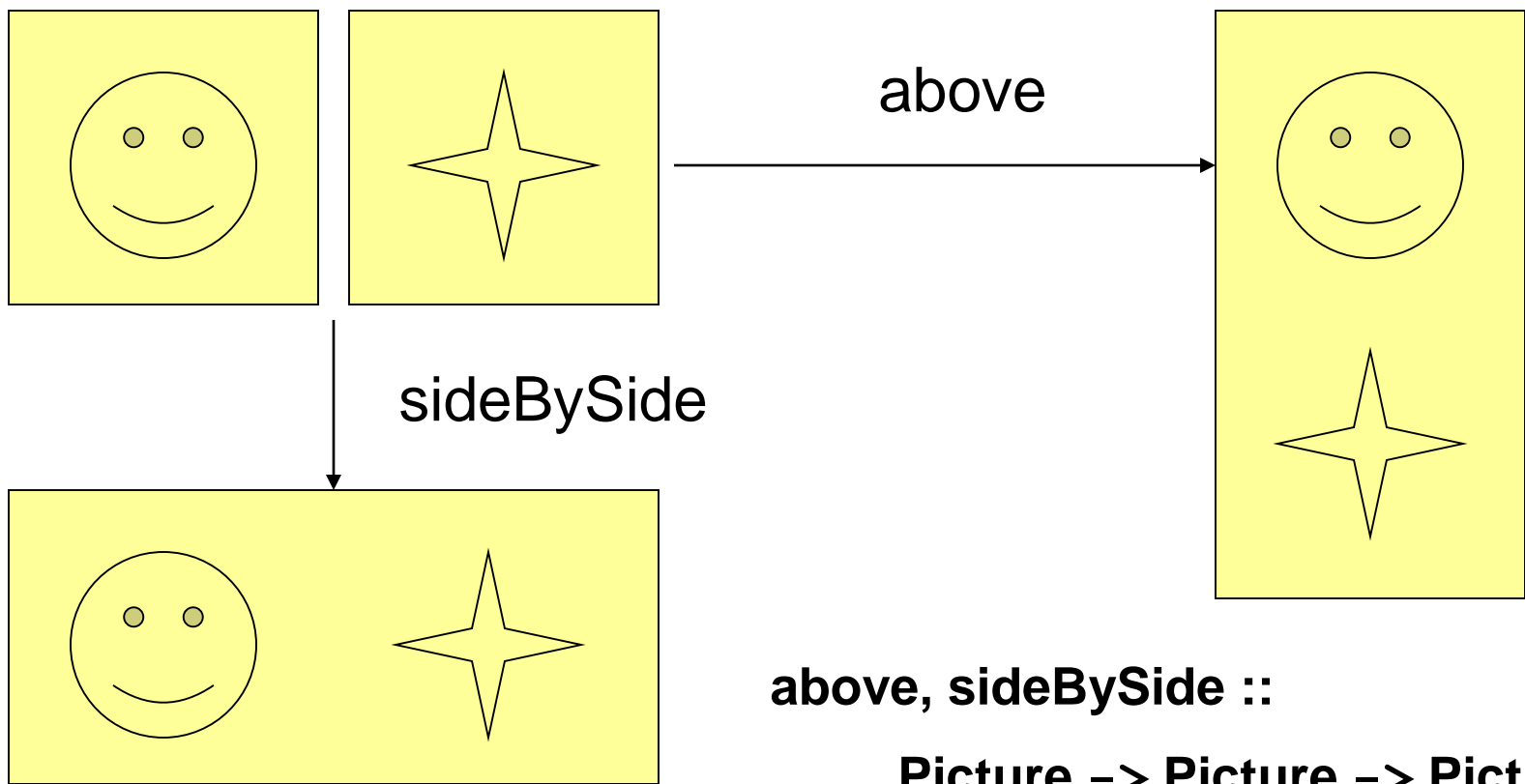
# Свойства на завъртането (flipping)

- flipH – хоризонтално завъртане (завъртане спрямо вертикалната ос);
- flipV - вертикално завъртане (завъртане спрямо хоризонталната ос).
  - flipH (flipH pic) == pic
  - flipV (flipV pic) == pic
  - flipH (flipV pic) == flipV (flipH pic)

Свойства, изисквани  
от потребителя

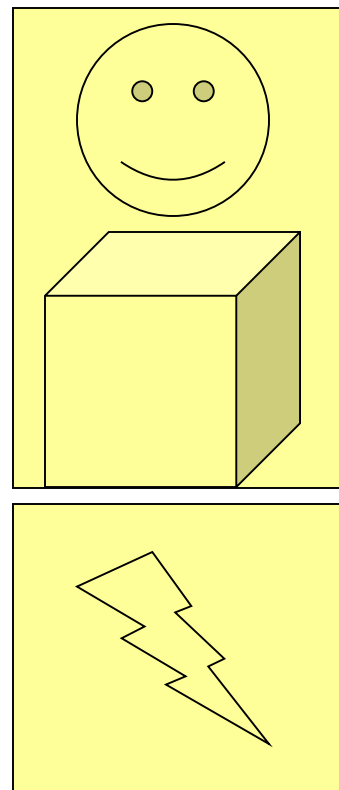
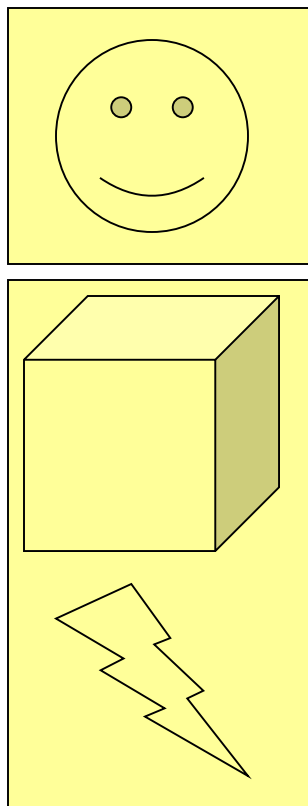
Свойства, които трябва да бъдат  
гарантирани от реализацията

# Комбиниране на картинки



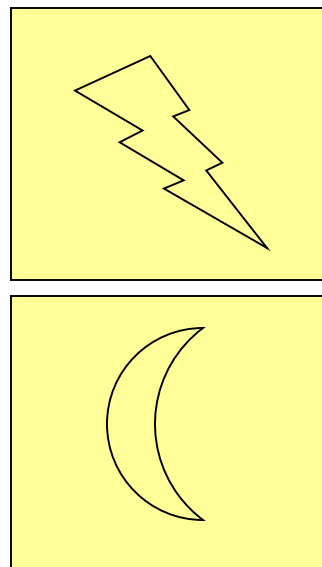
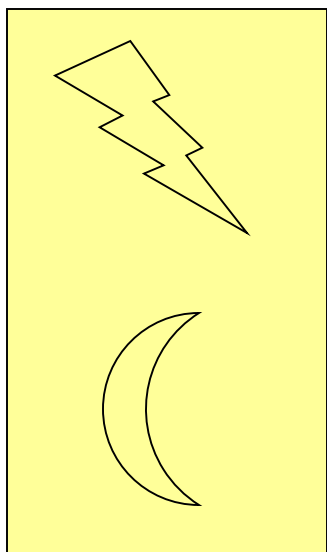
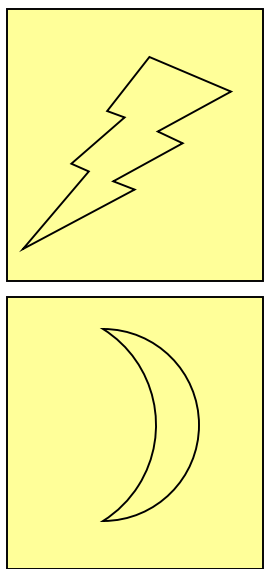
# Свойства на комбинациите

$a \text{ `above` } (b \text{ `above` } c) == (a \text{ `above` } b) \text{ `above` } c$



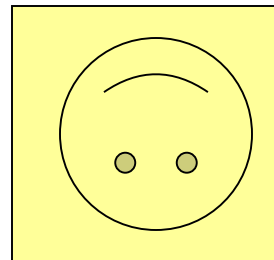
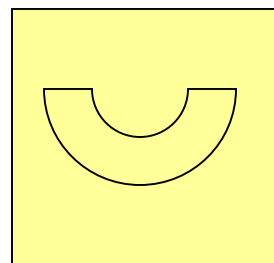
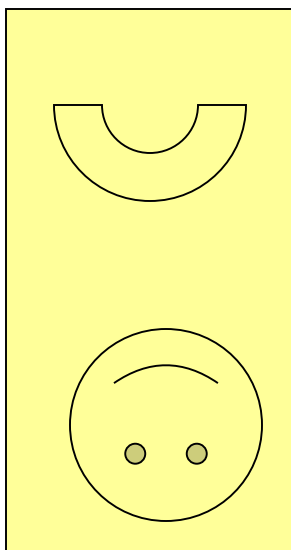
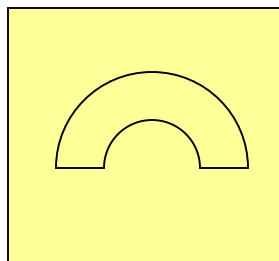
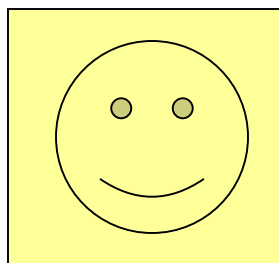
# Свойства на комбинациите

$\text{flipH} (a \text{ `above` } b) == \text{flipH } a \text{ `above` } \text{flipH } b$



# Свойства на комбинациите

$\text{flipV} (a \text{ `above` } b) == \text{flipV } b \text{ `above` } \text{flipV } a$



# Отпечатване на картинки

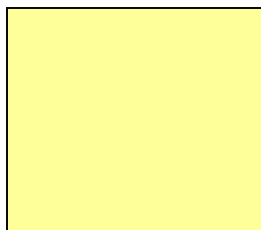
**Какво можем да правим с една картинка, когато тя вече е конструирана?**

`printPicture :: Picture -> IO ()`

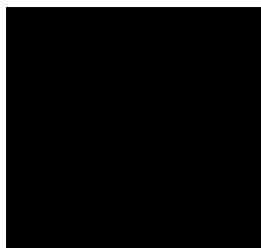


# Упражнение: Използване на картинки

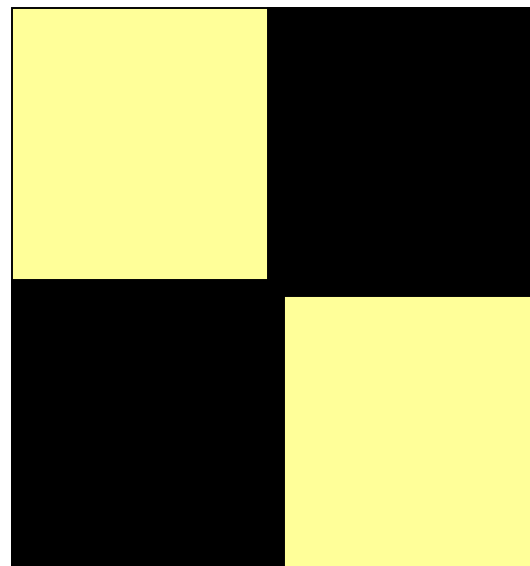
white



black

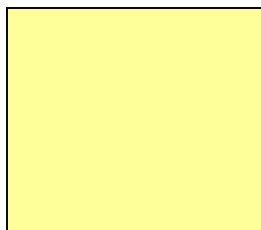


check

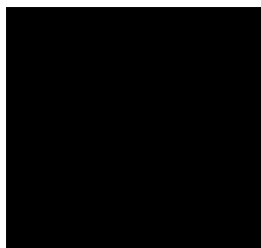


# Упражнение: Използване на картинки

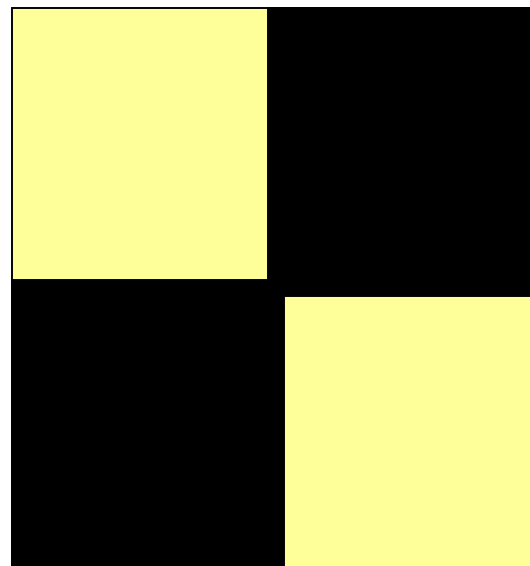
white



black



check



`wb = white `sideBySide` black`

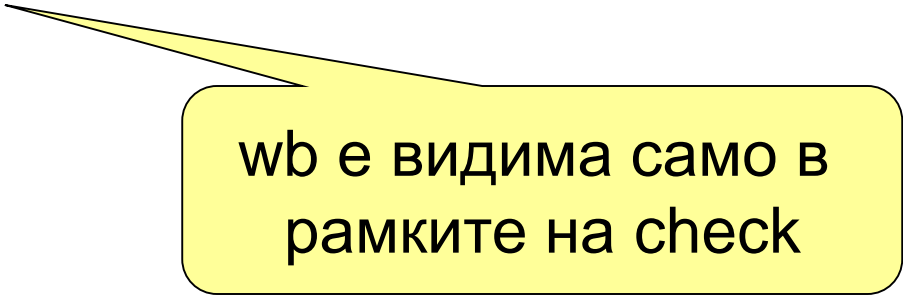
`check = wb `above` flipH wb`

# Локални дефиниции

wb се използва само при дефинирането на check;  
следователно, подходящо е използването на локална дефиниция:

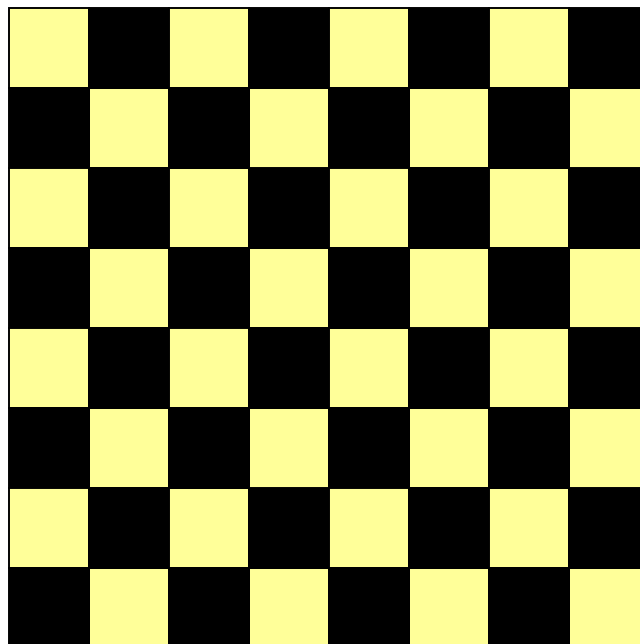
```
check = wb `above` flipH wb
```

```
where wb = white `sideBySide` black
```

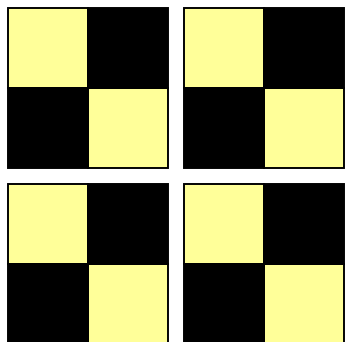


wb е видима само в  
рамките на check

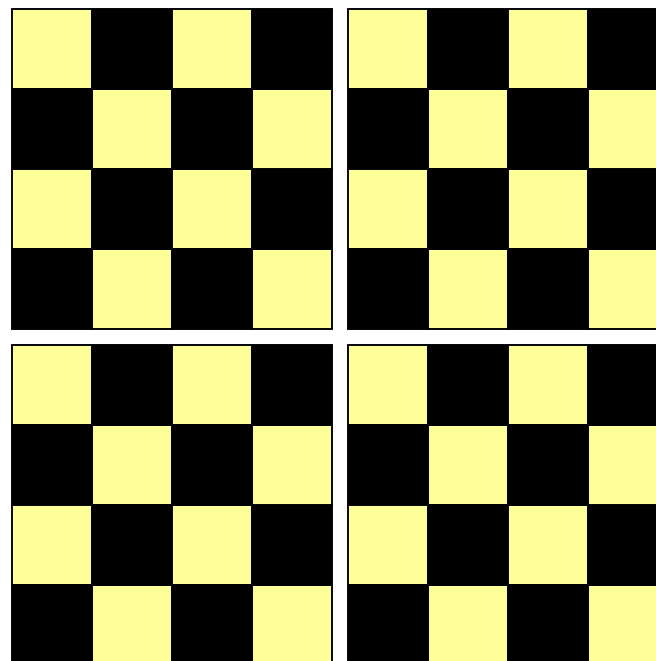
# Упражнение: Конструиране на шахматна дъска



# Упражнение: Конструирание на шахматна дъска

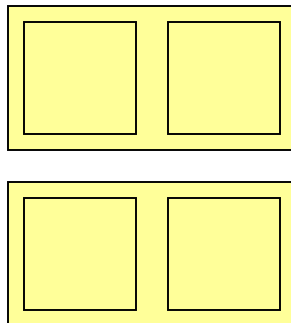


quartet check



quartet (quartet check)

## Дефиниране на quartet



`quartet :: Picture -> Picture`

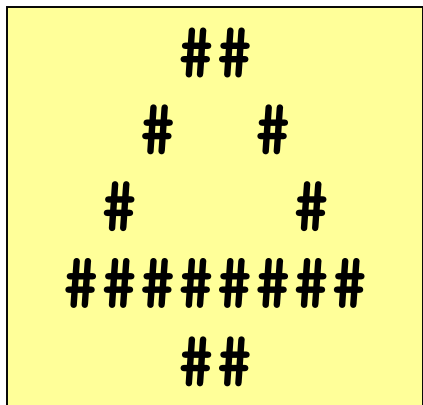
`quartet pic = two `above` two`

**where** `two = pic `sideBySide` pic`

Локалните дефиниции могат да използват аргументите на функцията

# Представяне на картинките

Ще изберем представяне, което позволява лесно отпечатване в рамките на използваната среда за програмиране – не е красиво, но е просто.



```
[ "      ##      ",  
  "    #    #    ",  
  "    #      #   ",  
  " #####",  
  "      ##      " ]
```

**type** Picture = [String]

# Същност на символните низове

Всеки низ е списък от знакове.

**type** String = [Char]

Типът на  
знаковете

**Примери:**

'a', 'b', 'c' :: Char

'' :: Char

Интервалът също  
е знак



# Вертикално завъртане (завъртане спрямо хоризонталната ос)

flipV

[ "        ##        " ,	[ "        ##        " ,
"        #    #        " ,	" ##### " ,
"        #        #        " ,	"        #        #        " ,
" ##### " ,	"        #    #        " ,
"        ##        "]"	"        ##        "]"

flipV :: Picture -> Picture

flipV pic = reverse pic

## Упражнение: Хоризонтално завъртане (завъртане спрямо вертикалната ос)

flipH

[ "       #       " ,	→	[ "       #       " ,
"           #       " ,		"       #       " ,
" ##### " ,		" ##### " ,
"           #       " ,		"       #       " ,
"       #       " ]		"       #       " ]

Recall:                      Picture = [[Char]]

# Упражнение: Хоризонтално завъртане (завъртане спрямо вертикалната ос)

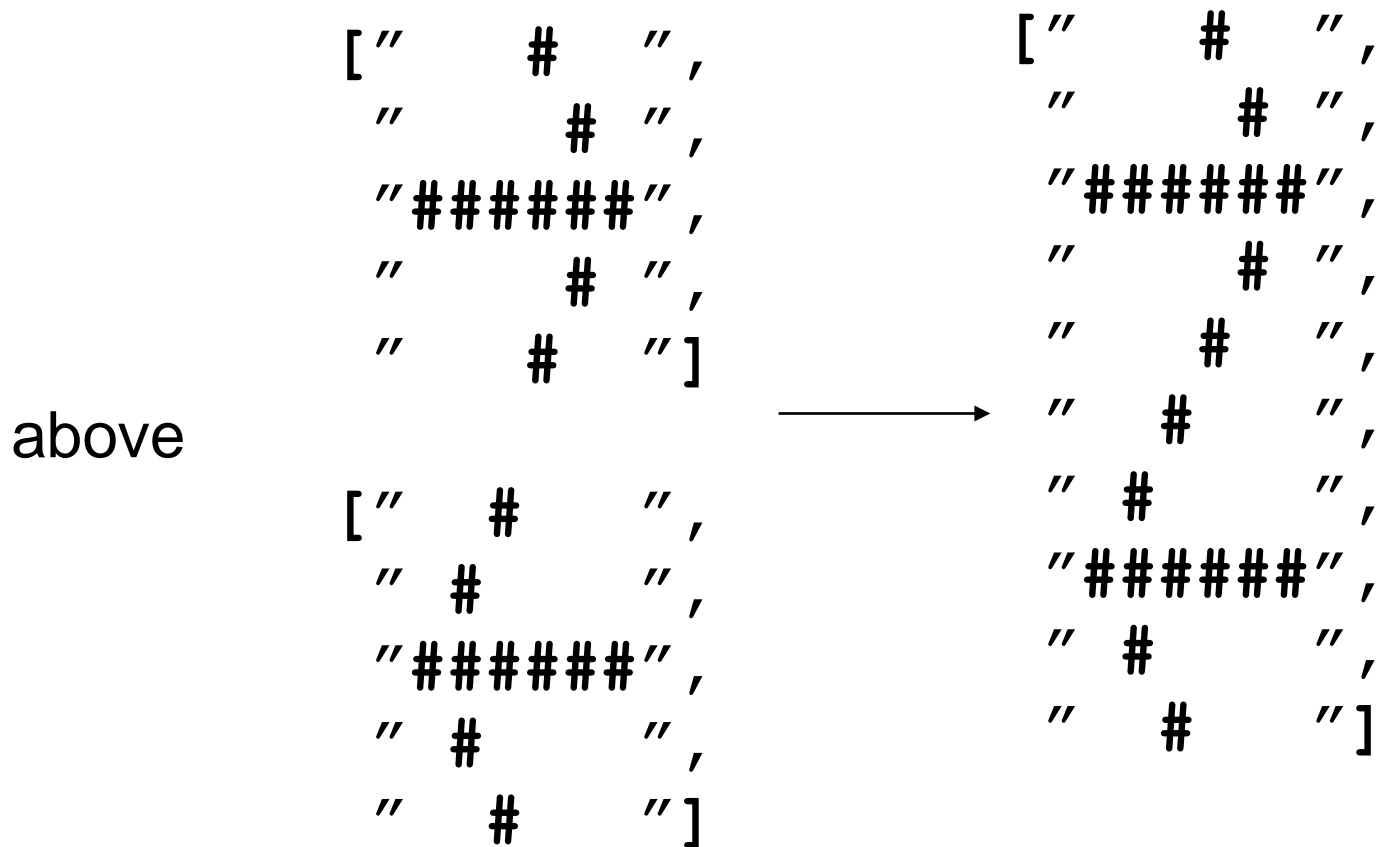
flipH

[ "       #       " ,	→	[ "       #       " ,
"           #       " ,		"       #       " ,
" ##### " ,		" ##### " ,
"           #       " ,		"       #       " ,
"       #       "]"		"       #       "]"

flipH :: Picture -> Picture

flipH pic = [reverse line | line <- pic]

# Вертикално комбиниране на картинки



above p q = p ++ q

# Хоризонтално комбинирање на картинки

```
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    " ]`sideBySide`
```

```
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    " ]
```

→

```
[ "    #    #    ",  
  "    #    #    ",  
  "#####",  
  "    #    #    ",  
  "    #    #    " ]
```

# Хоризонтално комбинирање на картинки

```
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    "]`sideBySide`  
[ "    #    ",  
  "    #    ",  
  "#####",  
  "    #    ",  
  "    #    "]
```

**sideBySide :: Picture -> Picture -> Picture**

**p `sideBySide` q = [pline ++ qline | (pline,qline) <- zip p q]**

# Отпечатване

Необходимо е да разполагаме със средство за отпечатване на низове. Haskell предвижда специална *команда* за тази цел.

`putStr :: String -> IO ()`

Команда, която не връща стойност. Като резултат от изпълнението ѝ се отпечатва даденият низ

Main> putStr "Hello!"

Hello!

Main>

Резултатът от изпълнението на командата

Main> "Hello!"

"Hello!"

Main>

Оценката на аргумента

# Преминаване на нов ред (Line Breaks)

Как можем да отпечатаваме повече от един ред?

Специалният знак `'\n'` означава край на ред.

```
Main> putStr  
"Hello\nWorld"
```

Hello

World

Main>

```
Main> "Hello\nWorld"
```

"Hello\nWorld"

Main>



## Отпечатване на картинка

Ще преобразуваме списъка `["###",  
"# #",  
"###"]` в низа

`"###\n# #\n###\n"`

За целта можем да използваме стандартната функция

`unlines :: [String] -> String`

`printPicture :: Picture -> IO ()`

`printPicture pic = putStr (unlines pic)`

# Разширение: Наслагване на картинки

superimpose

[ " # " ,	[ " # " ,
" # " ,	" # " ,
" ##### " ,	" ##### " ,
" # " ,	" # " ,
" # " ]	" # " ]

→

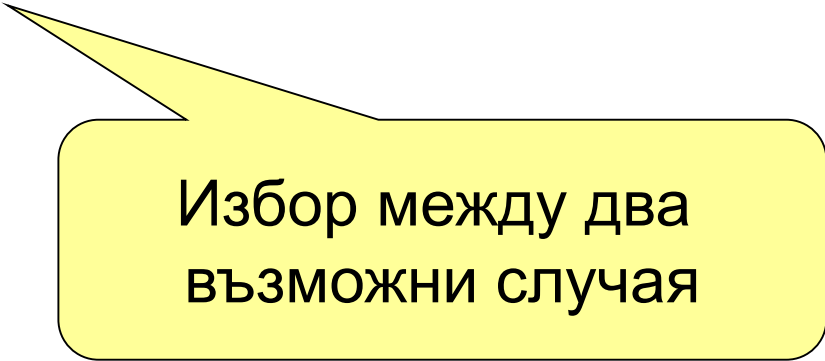
[ " ## " ,
" # # " ,
" ##### " ,
" # # " ,
" ## " ]

# По-проста задача: наслагване на знакове

Дефиниция на съответната помощна функция:

```
superimposeChar :: Char -> Char -> Char
```

```
superimposeChar c c' = if c=='#' then '#' else c'
```



Избор между два  
възможни случая

# По-проста задача: наслагване на редове

Дефиниция на съответната помощна функция:

```
superimposeLine :: String -> String -> String
```

```
superimposeLine s s' =
```

```
    [superimposeChar c c' | (c,c') <- zip s s']
```

```
superimposeLine "# "" #"
```

```
    [superimposeChar '#' ' ',
```

```
    → superimposeChar ' ' ' ', → "# #"
```

```
    superimposeChar ' ' '#']
```

## Решение: Наслагване на картинки

`superimpose :: Picture -> Picture -> Picture`

`superimpose p q =`

`[superimposeLine pline qline | (pline, qline) <- zip p q]`

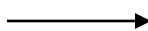
# Библиотеки от функции

Функциите за манипулиране на картинки могат да бъдат полезни за много други програми.

Вместо да копираме дефинициите на тези функции в други програми, можем да ги включим в подходяща *библиотека*, която другите програми ще имат право да използват.

Pictures.hs

```
module Pictures where  
type Picture = ...  
flipV pic = ...
```



Other.hs

```
import Pictures  
... flipV ... flipH ...
```

# Дефиниране на интерфейса

```
module Pictures  
  (Picture,  
   flipV, flipH,  
   above, sideBySide,  
   printPicture) where
```

...

Тези дефиниции  
могат да бъдат  
използвани от  
други програми

Останалите дефиниции са  
невидими за други програми