ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

http://www.corelab.ece.ntua.gr/~pagour/introcs/shmmy/

Επιμέλεια: Νίκος Παπασπύρου (nickie@softlab.ntua.gr)

Μοντέλα Προγραμματισμού

- ✓Συναρτησιακός προγραμματισμός
- ✓Λογικός προγραμματισμός
- ✓Αντικειμενοστρεφής προγραμματισμός

Κύρια προγραμματιστικά μοντέλα (i)

- Προστακτικός προγραμματισμός (*imperative programming*)
  - FORTRAN, Algol, COBOL, BASIC, C, Pascal, Modula-2, Ada
- Συναρτησιακός προγραμματισμός (*functional programming*)
  - LISP, ML, Scheme, Miranda, Haskell
- Λογικός προγραμματισμός (*logic programming*)
  - Prolog

Κύρια προγραμματιστικά μοντέλα (ii)

- Αντικειμενοστρεφής προγραμματισμός (*object-oriented programming*)
  - Simula, Smalltalk, C++, Eiffel, Java
- Παράλληλος/κατανεμημένος προγραμματισμός (*parallel/concurrent/distributed programming*)
  - OCCAM, Concurrent C, Ada, Java

Συναρτησιακός προγραμματισμός (i)

- Πλεονεκτήματα
  - Συντομία (*2-10 φορές μικρότερος κώδικας*)
  - Ευκολία στην κατανόηση
  - Λιγότερα σφάλματα εκτέλεσης
  - Επαναχρησιμοποίηση, αφαίρεση, δόμηση
  - Αυτόματη διαχείριση μνήμης

Παράδειγμα: QuickSort σε Haskell

```
qsort [] = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort ge
  where lt = [y | y <- xs, y < x]
        ge = [y | y <- xs, y >= x]
```

Συναρτησιακός προγραμματισμός (ii)

- Μειονεκτήματα
  - Μειωμένη απόδοση
  - Μεγαλύτερες απαιτήσεις μνήμης
- Όχι μειονεκτήματα ☺
  αλλαγή φιλοσοφίας στον προγραμματισμό
  - Όχι μεταβλητές, όχι εντολές
  - Εκφράσεις και συναρτήσεις
- Τα παραδείγματα που ακολουθούν είναι σε Haskell http://www.haskell.org/

Δηλώσεις και εξαγωγή τύπων

- Δήλωση συναρτήσεων

```
inc n = n+1
f t = t * inc t
```

- Δήλωση τιμών

```
x = f 6
y = f (f 2)
```

- Εξαγωγή τύπων (*type inference*)
  - Οι τύποι υπολογίζονται αυτόματα

```
inc, f :: Int -> Int
x, y   :: Int
```

Υπολογισμοί τιμών

- Υπολογισμός τιμής

```
x → f 6 → 6 * inc 6
  → 6*(6+1) → 6*7 → 42
```

- Το αποτέλεσμα είναι ανεξάρτητο της σειράς των επιμέρους υπολογισμών *(υπό κ.σ.)*

```
y → f (f 2) → f (2 * inc 2)
  → f (2*(2+1)) → f (2*3) → f 6
  → 6 * inc 6 → 6*(6+1) → 6*7 → 42

y → f (f 2) → f 2 * inc (f 2)
  → (2 * inc 2) * inc (2 * inc 2)
  → (2*(2+1)) * inc (2*(2+1))
  → (2*(2+1)) * (2*(2+1)+1) → … → 42
```


Τοπικές δηλώσεις

- Με χρήση του let

```
x = let inc n = n+1
        f t = t * inc t
    in  f 6
```

- … ή με χρήση του where

```
x = f 6
  where inc n = n+1
        f t = t * inc t
```

- Οι τοπικές δηλώσεις ακολουθούν κανόνες εμβέλειας όπως π.χ. της Pascal

Πλειάδες τιμών

- Συναρτήσεις με πολλές παραμέτρους

```
add :: (Int, Int) -> Int
add (x, y) = x+y
```

- ... και πολλά αποτελέσματα

```
solve2eq :: (Double, Double, Double)
                -> (Double, Double)
solve2eq (a, b, c) =
    let d  = b*b - 4*a*c
        x1 = (-b - sqrt(d)) / (2*a)
        x2 = (-b + sqrt(d)) / (2*a)
    in  (x1, x2)
```


Αναδρομή

- Στο συναρτησιακό προγραμματισμό είναι ο κύριος τρόπος επαναληπτικών υπολογισμών
  - Υπολογισμός παραγοντικού

```
factorial n =
  if n <= 1 then 1
            else n * factorial (n-1)
```

  - Υπολογισμός Μ.Κ.Δ. (αλγόριθμος Ευκλείδη)

```
gcd (n, 0) = n
gcd (n, m) = gcd(m, n `mod` m)
```

αν $m \neq 0$

pattern matching στις παραμέτρους

Συναρτήσεις υψηλής τάξης

- Συναρτήσεις που παίρνουν ως παραμέτρους άλλες συναρτήσεις

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

inc n = n + 1
plus2 x = twice (inc, x)
```

- ... ή που έχουν ως αποτέλεσμα συναρτήσεις

```
plusN :: Int -> (Int -> Int)
plusN x = let f y = x + y
          in  f
```


Ανώνυμες συναρτήσεις

- "Η συνάρτηση που απεικονίζει κάθε *n* στο *n*+1" $\lambda n.\ n+1$

```
\n -> n+1
```

- Παράδειγμα

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

plus2 :: Int -> Int
plus2 x = twice (\n -> n+1, x)

plusN :: Int -> (Int -> Int)
plusN x = \y -> x + y
```

Παραμέτρων συνέχεια (i)

- “Currying” *(Haskell B. Curry)*
  - Μια συνάρτηση με δύο παραμέτρους ισοδυναμεί με μια συνάρτηση που δέχεται την πρώτη παράμετρο και επιστρέφει μια συνάρτηση που δέχεται τη δεύτερη

```
add :: (Int, Int) -> Int
add (x, y) = x+y
```

```
add′ :: Int -> (Int -> Int)
add′ x = \y -> x+y
```

Curried version

```
add (x, y) == (add′ x) y
```


Παραμέτρων συνέχεια (ii)

- Απλούστερη γραφή curried συναρτήσεων

```
add :: Int -> Int -> Int
add x y = x+y
```

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

- Με τις curried συναρτήσεις επιτρέπεται η “μερική εφαρμογή”

```
twice (add 20) 2 ➔ add 20 (add 20 2)
  ➔ add 20 (20+2) ➔ 20+(20+2) ➔ 42
```

Λίστες (i)

◆ Ακολουθίες ομοειδών στοιχείων

```
digits :: [Int]
digits = [0,1,2,3,4,5,6,7,8,9]

dictionary :: [(String, String)]
dictionary = [("apple", "μήλο"),
              ("pear", "αχλάδι"),
              ("pencil", "μολύβι")]
```

◆ Παραδείγματα με λίστες

- Εύρεση μήκους

```
length [] = 0
length (x:xs) = 1 + length xs
```


Λίστες (ii)

◆ Παραδείγματα με λίστες (συνέχεια)

- Συνένωση δύο λιστών

```
concat [] ys = ys
concat (x:xs) ys = x : concat xs ys
```

- Αντιστροφή λίστας

```
reverse [] = []
reverse (x:xs) =
   concat (reverse xs) [x]
```

- Αντιστροφή λίστας *(καλύτερη υλοποίηση)*

```
reverse xs = aux xs []
  where aux [] ys = ys
        aux (x:xs) ys = aux xs (x:ys)
```

Λίστες (iii)

- Παραδείγματα με λίστες και συναρτήσεις υψηλής τάξης
  - Εφαρμογή μιας συνάρτησης σε όλα τα στοιχεία μιας λίστας

```
map f [] = []
map f (x:xs) = f x : map f xs
```

  - "Φιλτράρισμα" των στοιχείων μιας λίστας

```
filter f [] = []
filter f (x:xs) =
  if f x then x : filter f xs
         else filter f xs
```


Παραμετρικός πολυμορφισμός (i)

- Ποιος ο τύπος της ταυτοτικής συνάρτησης;

```
id x = x
```

- Μπορεί να δεχθεί παραμέτρους κάθε τύπου

```
id 42       → 42       :: Int
id "Hello"  → "Hello"  :: String
id inc      → inc      :: Int -> Int
```

- Είναι μια πολυμορφική συνάρτηση

`id :: a -> a` (*για κάθε τύπο* `a`)

Παραμετρικός πολυμορφισμός (ii)

- Περισσότερες πολυμορφικές συναρτήσεις

```
length  :: [a] -> Int
concat  :: [a] -> [a] -> [a]
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

  - και με περισσότερους "άγνωστους" τύπους

```
map :: (a -> b) -> [a] -> [b]
```

  - ένα ακόμα σύνθετο παράδειγμα

```
zip :: [a] -> [b] -> [(a, b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```


Ορισμοί τύπων (i)

- Απλές απαριθμήσεις

```
data Light = Red | Green | Yellow
next :: Light -> Light
next Green = Yellow
next Yellow = Red
next Red = Green
```

- Πιο σύνθετοι τύποι δεδομένων

```
data Number = NInteger Int
            | NReal    Double
            | NComplex Double Double
neg (NInteger n) = NInteger (-n)
neg (NReal r) = NReal (-r)
neg (NComplex x y) = NComplex (-x) (-y)
```

Ορισμοί τύπων (ii)

◆ Αναδρομικά ορισμένοι τύποι

- Συνδεδεμένες λίστες

```
data ListOfInt = Nil | Cons Int List
```

ή και πολυμορφικές λίστες

```
data List a = Nil | Cons a (List a)
```

- Παραδείγματα

```
sum :: List Int -> Int
sum Nil = 0
sum (Cons x xs) = x + sum xs

length :: List a -> Int
length Nil = 0
length (Cons x xs) = length xs
```


Ορισμοί τύπων (iii)

◆ Αναδρομικά ορισμένοι τύποι (συνέχεια)

- Πολυμορφικά δυαδικά δέντρα

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)
```

- Μέτρηση κόμβων

```
count :: Tree a -> Int
count Nil = 0
count (Node a left right) =
  1 + count left + count right
```

Ορισμοί τύπων (iv)

◆ Αναδρομικά ορισμένοι τύποι (συνέχεια)

- Διάσχιση κατά βάθος

```
preorder :: Tree a -> [a]
preorder Nil = []
preorder (Node a left right) =
   a : preorder left ++ preorder right
```

- Διάσχιση κατά πλάτος

```
traverseBF :: Tree a -> [a]
traverseBF t = aux [t]
   where aux [] = []
         aux (Nil : ts) = aux ts
         aux (Node a left right : ts) =
           a : aux (ts ++ [left, right])
```


Ορισμοί τύπων (v)

◆ Αναδρομικά ορισμένοι τύποι (συνέχεια)

- Διάσχιση κατά βάθος *(καλύτερη υλοποίηση)*

```
preorder t = aux t []
   where aux Nil ts = ts
         aux (Node a left right) ts =
           a : aux left (aux right ts)
```

- Διάσχιση κατά πλάτος *(καλύτερη υλοποίηση)*

```
traverseBF t = aux [t] []
   where
     aux [] [] = []
     aux [] ys = aux (reverse ys) []
     aux (Nil : xs) ys = aux xs ys
     aux (Node a left right : xs) ys =
       a : aux xs (right : left : ys)
```

Πρόθυμη και οκνηρή αποτίμηση (i)

- **Πρόθυμη αποτίμηση** (*eager evaluation*)
  - Οι υπολογισμοί γίνονται το νωρίτερο δυνατόν
  - Οι παράμετροι των συναρτήσεων αποτιμώνται πριν την κλήση
  - π.χ. LISP, ML, Scheme
- **Οκνηρή αποτίμηση** (*lazy evaluation*)
  - Οι υπολογισμοί γίνονται το αργότερο δυνατόν, δηλαδή μόνο αν χρειαστεί το αποτέλεσμά τους
  - Οι παράμετροι των συναρτήσεων αποτιμώνται την πρώτη φορά που θα χρειαστεί η τιμή
  - π.χ. Miranda, Haskell

Πρόθυμη και οκνηρή αποτίμηση (ii)

- **Παράδειγμα: άπειρη αναδρομή**

```
loop n = loop (n+1)
foo x y = if x == 1 then y else 42
```

  - Πρόθυμη αποτίμηση

```
foo 7 (loop 0) ➔ foo 7 (loop (0+1))
 ➔ foo 7 (loop 1) ➔ foo 7 (loop (1+1))
 ➔ foo 7 (loop 2) ➔ …
```

(δεν τερματίζεται)

  - Οκνηρή αποτίμηση

```
foo 7 (loop 0)
 ➔ if 7 == 1 then loop 0 else 42
 ➔ 42
```

Πρόθυμη και οκνηρή αποτίμηση (iii)

- Παράδειγμα: άπειρη λίστα πρώτων αριθμών με το κόσκινο του Ερατοσθένη

```
primes :: [Int]
primes = sieve (natsgt 2)
  where
    natsgt n = n : natsgt (n+1)
    sieve (x:xs) =
      x : sieve (filter (ndiv x) xs)
    ndiv x y = y `mod` x /= 0
```

```
primes == [2,3,5,7,11,13,17,19,23,29,…]
```

```
allnats = 0 : map (\n -> n+1) allnats
```


Πέρα από το συναρτησιακό μοντέλο

- “Αγνός” συναρτησιακός προγραμματισμός (*purely functional programming*)
- Παρενέργειες (*side effects*)
  - Μεταβλητές (*mutable variables*)
    - ανάθεση (αποθήκευση τιμής)
    - προσπέλαση (ανάκληση τιμής)
  - Είσοδος/έξοδος (*input/output*)
    - εκτύπωση σε οθόνη ή σε αρχείο
    - ανάγνωση από το πληκτρολόγιο ή από αρχείο

Λογικός προγραμματισμός

- **Κεντρική ιδέα**
  - Το πρόγραμμα είναι εκφρασμένο σε μια μορφή συμβολικής λογικής
  - Η εκτέλεση του προγράμματος ισοδυναμεί με τη διεξαγωγή συλλογισμών σε αυτή τη λογική
- **Η γλώσσα Prolog**
  - W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 4th edition, Springer-Verlag, New York, 1997.

Κατηγορηματική λογική (i)

(*predicate logic*)

- **Προτάσεις**
  - Δηλώσεις σε συμβολική μορφή που είναι είτε αληθείς είτε όχι αληθείς
  - Αναφέρονται σε αντικείμενα και σε σχέσεις μεταξύ αυτών
  - Ατομική πρόταση: `κατηγόρημα(ορίσματα)`
  - Τελεστές: $\neg$ (όχι) $\wedge$ (και) $\vee$ (ή) $\Rightarrow$ (συνεπάγεται) $\Leftarrow$ (προκύπτει από) $\Leftrightarrow$ (ισοδυναμεί) $\forall$ (για κάθε) $\exists$ (υπάρχει)

Κατηγορηματική λογική (ii)

- Κανονική μορφή και προτάσεις Horn
  - Κάθε πρόταση μπορεί να γραφεί στην παρακάτω κανονική μορφή

$$B_1 \vee B_2 \vee \ldots \vee B_m \Leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

όπου $A_1, A_2, \ldots A_n$ και $B_1, B_2, \ldots B_m$ είναι ατομικές προτάσεις

  - Πολλές (αλλά όχι όλες) οι προτάσεις μπορούν να γραφούν σε μορφή *πρότασης Horn*

$$B \Leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

ή

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

Γεγονότα και κανόνες (i)

- Δήλωση γεγονότων

```
male(john).
male(george).
female(mary).
female(jenny).
parent(john,george).
parent(mary,george).
parent(john,jenny).
parent(mary,jenny).
```

- Δήλωση κανόνων

```
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

Γεγονότα και κανόνες (ii)

- Δήλωση κανόνων (συνέχεια)

```
human(X)  :- male(X).
human(X)  :- female(X).
brother(X,Y) :- male(X), parent(Z,X),
                parent(Z,Y).
sister(X,Y) :- female(X), parent(Z,X),
               parent(Z,Y).
```

- Ερωτήσεις – στόχοι (*goals*)

```
?- male(john).
yes
?- male(mary).
no
```


Γεγονότα και κανόνες (iii)

- Ερωτήσεις – στόχοι (συνέχεια)

```
?- male(peter).
no
?- male(X).
X=john;
X=george;
no
?- human(X).
X=john;
X=george;
X=mary;
X=jenny;
no
```

Γεγονότα και κανόνες (iv)

- Ερωτήσεις – στόχοι (συνέχεια)

```
?- brother(george,jenny).
yes
?- mother(X,george).
X=mary;
no
?- sister(X,Y).
X=jenny, Y=george;
X=jenny, Y=jenny;
X=jenny, Y=george;
X=jenny, Y=jenny;
no
```


Επίλυση (i)

(*resolution*)

- Αλγόριθμος επίλυσης
  - Η ερώτηση-στόχος συγκρίνεται με τα γεγονότα και τα αριστερά μέλη των κανόνων
  - Ενοποίηση (*unification*): αυτή η σύγκριση προκαλεί πιθανώς τη συγκεκριμενοποίηση κάποιων μεταβλητών
  - Αν η σύγκριση επιτύχει για κάποιο γεγονός, ο στόχος έχει ικανοποιηθεί
  - Αν επιτύχει για το αριστερό μέλος κανόνα, οι υποθέσεις προστίθενται στη λίστα των στόχων

Επίλυση (ii)

- Αλγόριθμος επίλυσης (συνέχεια)
  - Επιστροφή (*backtracking*): αν ένας στόχος αποτύχει, δηλαδή δεν μπορεί να ικανοποιηθεί, ο αλγόριθμος επιστρέφει στον αμέσως προηγούμενο στόχο και επαναλαμβάνει με διαφορετική επιλογή γεγονότος ή κανόνα
- Παράδειγμα

```
?- mother(X,george).
parent(X,george), female(X).              backtracking
X=john
  parent(john,george), female(john).     ✗
X=mary
  parent(mary,george), female(mary).      ✓
```


Μειονεκτήματα της Prolog (i)

- Η σειρά εμφάνισης γεγονότων, κανόνων και στόχων είναι καθοριστική

```
ancestor(X,X).
ancestor(X,Y) :- ancestor(Z,Y),
                 parent(X,Z).
```

  - Για το λόγο αυτό υπάρχει η τομή ! (*cut*)
- Υπόθεση κλειστού κόσμου
  - Οι μόνες αληθείς προτάσεις είναι αυτές που αποδεικνύονται βάσει των γνωστών γεγονότων και κανόνων

Μειονεκτήματα της Prolog (ii)

- Το πρόβλημα της άρνησης
  - Μια αρνητική πρόταση **not** *A* είναι αληθής όταν ο στόχος *A* δεν μπορεί να ικανοποιηθεί
  - Αυτή η άρνηση δεν ταυτίζεται με τη λογική άρνηση, π.χ. **not not** $A \neq A$
- Το πρόβλημα των "προδιαγραφών"
  - Ένα πρόγραμμα ισοδυναμεί με τον ορισμό των προδιαγραφών του σε κατηγορηματική λογική
  - Ο μετασχηματισμός των προδιαγραφών σε αλγόριθμο επίλυσης είναι ένα άλυτο πρόβλημα

Παραδείγματα (i)

- Αριθμητικές πράξεις

```
daysOf(january,Y,31).
daysOf(february,Y,29) :-
    Y mod 400 =:= 0,
    Y mod 4000 =\= 0, !.
daysOf(february,Y,29) :-
    Y mod 4 =:= 0,
    Y mod 100 =\= 0, !.
daysOf(february,Y,28).
daysOf(march,Y,31).
...
daysOf(december,Y,31).
validDate(D,M,Y) :- daysOf(M,Y,X),
                    D>=1, D=<X.
```

Παραδείγματα (ii)

- Λίστες

```
length([],0).
length([X|Xs],N) :-
  length(Xs,M), N is M+1.
member(X,[X|_]).
member(X,[_|Xs]) :-
  member(X,Xs).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
   append(Xs,Ys,Zs).
```


Παραδείγματα (iii)

- Λίστες (συνέχεια)

```
?- length([1,2,3,4,5],X).
X=5;
no
?- member(X,[1,2,3]).
X=1;
X=2;
X=3;
no
?- append([1,2,3],[4,5,6],L).
L=[1,2,3,4,5,6];
no
```

Παραδείγματα (iv)

- Λίστες (συνέχεια)

```
?- append(X,[4,5],[1,2,3,4,5]).
X=[1,2,3];
no
?- append([1|X],Y,[1,2,3]).
X=[], Y=[2,3];
X=[2], Y=[3];
X=[2,3], Y=[];
no
```


Παραδείγματα (v)

- Λίστες και (*αφελής*) ταξινόμηση

```
sort(L,SL) :-
  permutation(L,SL), sorted(SL), !.
permutation(L,[H|T]) :-
  append(V,[H|U],L),
  append(V,U,W),
  permutation(W,T).
permutation([],[]).
sorted([]).
sorted([H|T]) :- sortedx(H,T).
sortedx(_,[]).
sortedx(N,[H|T]) :- N=<H, sortedx(H,T).
```

Παραδείγματα (vi)

- Λίστες και (*αφελής*) ταξινόμηση (συνέχεια)

```
?- sort([42,13,77],L).
L=[13,42,77];
no
?- permutation([1,2,3],X).
X=[1,2,3];
X=[1,3,2];
X=[2,1,3];
X=[2,3,1];
X=[3,1,2];
X=[3,2,1];
no
```


Παραδείγματα (vii)

- Λίστες και QuickSort

```
qsort([H|T],S) :-
  split(H,T,A,B),
  qsort(A,SA),
  qsort(B,SB),
  append(SA,[H|SB],S).
split(H,[A|X],[A|Y],Z) :-
  A=<H, split(H,X,Y,Z).
split(H,[A|X],Y,[A|Z]) :-
  H<A, split(H,X,Y,Z).
split(_,[],[],[]).
```

Αντικειμενοστρεφής προγραμματισμός (i)

- Αντικειμενοστρεφές (*object-oriented*) μοντέλο ανάπτυξης λογισμικού
  - Το πρόγραμμα είναι οργανωμένο ως ένα σύνολο από αλληλεπιδρώντα αντικείμενα
  - Κάθε αντικείμενο περιέχει:
    - δεδομένα (*data*), που χαρακτηρίζουν την κατάστασή του
    - μεθόδους (*methods*), δηλαδή κώδικα που υλοποιεί τη συμπεριφορά του

    $\Rightarrow$ ενθυλάκωση (*encapsulation*)

Αντικειμενοστρεφής προγραμματισμός (ii)

- Βασικές αρχές
  - Κατά την ανάλυση και τη σχεδίαση, δείτε τα αντικείμενα βάσει της διαπροσωπείας τους (interface) και όχι βάσει της υλοποίησής τους
  - Προσπαθήστε να κρύψετε όσο το δυνατόν μεγαλύτερο μέρος της υλοποίησης
  - Προσπαθήστε να επαναχρησιμοποιήσετε αντικείμενα
  - Προσπαθήστε να ελαχιστοποιήσετε τις διασυνδέσεις μεταξύ αντικειμένων

Αντικειμενοστρεφής προγραμματισμός (iii)

- Πλεονεκτήματα
  - Φυσική περιγραφή, τουλάχιστον για ορισμένες περιοχές εφαρμογών
  - Αφαίρεση, δόμηση, επαναχρησιμοποίηση
  - Ιδιαίτερα διαδεδομένος σήμερα, πληθώρα γλωσσών και εργαλείων τον υποστηρίζουν
- Μειονεκτήματα
  - Υπερτιμημένος... ☺
  - Διόγκωση κώδικα
- Τα παραδείγματα που ακολουθούν είναι σε Java http://java.sun.com/

Κλάσεις και αντικείμενα (i)

- Παράδειγμα: μετρητής

```
public class Counter
{
  private int value;
  Counter ()  { value = 0; }
  void inc () { value++; }
  int  get () { return value; }
}
```

Κλάσεις και αντικείμενα (ii)

- Παράδειγμα (συνέχεια)

```
public class DemoProgram
{
  public static
    void main (String [] args)
  {
    Counter c = new Counter();

    for (int i=0; i<42; i++) {
      c.inc();
      System.out.println(c.get());
    }
  }
}
```


Κλάσεις και αντικείμενα (iii)

- Παράδειγμα: μιγαδικοί αριθμοί

```
public class Complex
{
  private double re, im;

  Complex ()
    { re = im = 0.0; }
  Complex (double r)
    { re = r; im = 0.0; }
  Complex (double r, double i)
    { re = r; im = i; }

  void negate ()
    { re = -re; im = -im; }
```

Κλάσεις και αντικείμενα (iv)

- Παράδειγμα (συνέχεια)

```
    Complex add (Complex c) {
        return new Complex(re + c.re,
                           im + c.im);
    }

    void print () {
        System.out.print(re);
        if (im > 0.0)
            System.out.print("+");
        if (im != 0.0) {
            System.out.print(im);
            System.out.print("j");
        }
    }
}
```


Κλάσεις και αντικείμενα (v)

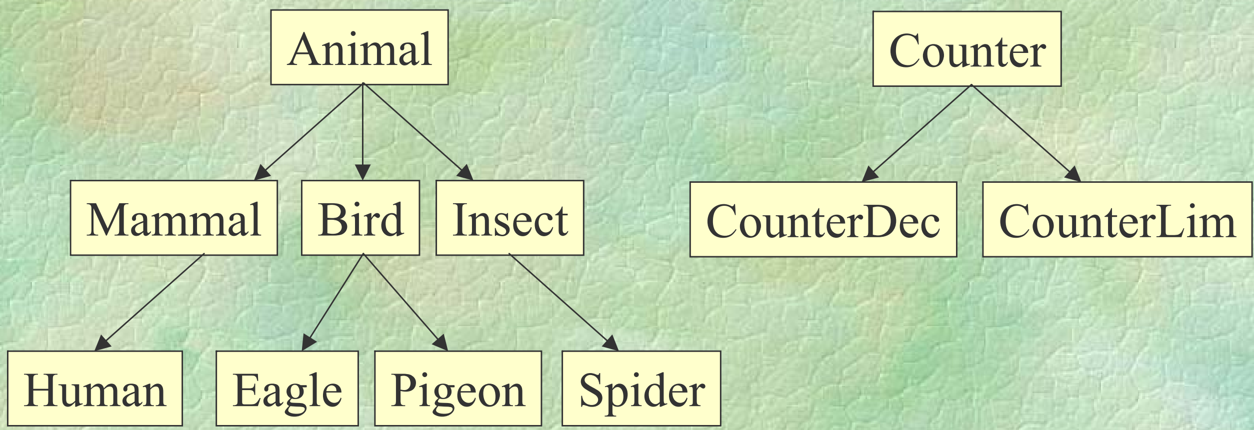
- Παράδειγμα (συνέχεια)

```
public class DemoProgram2
{
    public static
        void main (String [] args)
    {
        Complex c1 = new Complex(1);
        Complex cj = new Complex(0, 1);
        Complex c;

        c = c1.add(cj);
        c.negate();
        c.print();
        System.out.println("\n");
    }
}
```

Ιεραρχίες κλάσεων (i)

- Κληρονομικότητα (*inheritance*)
  - Εξειδίκευση των αντικειμένων μιας κλάσης, υποστηρίζοντας πρόσθετη ή διαφοροποιημένη συμπεριφορά



Ιεραρχίες κλάσεων (ii)

- Παράδειγμα: εξειδικευμένοι μετρητές

```
public class CounterDec
  extends Counter
{
  void dec () {
    if (value > 0) value--;
  }
}
```

επιπλέον μέθοδος

```
public class CounterLim
  extends Counter
{
  void inc () {
    if (value < 30) value++;
  }
}
```

υπερίσχυση μεθόδου

Ιεραρχίες κλάσεων (iii)

◆ Παράδειγμα (συνέχεια)

get και **inc** ορίζονται στην κλάση **Counter**

dec ορίζεται στην κλάση **CounterDec**

```
public class DemoProgram3
{
  public static
    void main (String [] args)
  {
    CounterDec c = new CounterDec();
    for (int i=0; i<42; i++) {
      c.inc();
      System.out.println(c.get());
    }
    for (int i=0; i<42; i++) {
      c.dec();
      System.out.println(c.get());
    }
  }
}
```


Ιεραρχίες κλάσεων (iv)

◆ Παράδειγμα (συνέχεια)

get ορίζεται στην κλάση **Counter**

inc ορίζεται στην κλάση **CounterLim**

```
public class DemoProgram4
{
  public static
    void main (String [] args)
  {
    CounterLim c = new CounterLim();
    for (int i=0; i<42; i++) {
      c.inc();
      System.out.println(c.get());
    }
  }
}
```

Σχέση υποτύπων (i)

- Τα αντικείμενα της εξειδικευμένης κλάσης μπορούν να χρησιμοποιούνται μέσω αναφορών σε αντικείμενα της βασικής κλάσης
- Παράδειγμα

```
Counter c = new CounterDec();

for (int i=0; i<42; i++) {
  c.inc();
  System.out.println(c.get());
}
```


Σχέση υποτύπων (ii)

- Πολυμορφισμός υποτύπων (*subtype polymorphism*)

```
Counter c1 = new Counter();
Counter c2 = new CounterLim();

for (int i=0; i<42; i++) {
  c1.inc();
  c2.inc();
}
```

- Ποιες θα είναι οι τιμές των μετρητών;

  ⇒ δυναμικό δέσιμο (*dynamic binding*)

  ⇒ η `c2.inc()` καλεί την `inc` της `CounterLim`

Αφηρημένες κλάσεις

- Κλάσεις που περιέχουν αφηρημένες (*abstract*) μεθόδους, οι οποίες
  - διαθέτουν μόνο επικεφαλίδα
  - η υλοποίησή τους δίνεται σε εξειδικευμένες κλάσεις
- Παράδειγμα

```
public abstract class Expression
{
  abstract double eval ();
}
```


Εκφράσεις σε δέντρα (i)

- Παράδειγμα: ένας υπολογιστής για αριθμητικές εκφράσεις
  - Σταθερές

```
public class Constant
  extends Expression
{
  private double value;
  Constant (double d)
    { value = d; }
  double eval ()
    { return value; }
}
```

Εκφράσεις σε δέντρα (ii)

◆ Παράδειγμα (συνέχεια)

- Αριθμητικές πράξεις

```
public abstract class Operation
  extends Expression
{
  private Expression left, right;

  Operation (Expression l,
             Expression r)
    { left = l; right = r; }
}
```


Εκφράσεις σε δέντρα (iii)

◆ Παράδειγμα (συνέχεια)

- Πρόσθεση

```
public class Plus extends Operation
{
  Plus (Expression l, Expression r)
    { super(l, r); }

  double eval ()
    { return left.eval() +
             right.eval(); }
}
```

Εκφράσεις σε δέντρα (iv)

- Παράδειγμα (συνέχεια)

```
public class DemoProgram7
{
  public static
    void main (String [] args)
  {
    Expression e1 =
      new Plus(new Constant(3),
               new Constant(4));
    Expression e2 =
      new Minus(new Constant(8),
                new Constant(2));
    Expression e = new Times(e1, e2);

    System.out.println(e.eval());
  }
}
```