ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ ΠΟΛΥΜΟΡΦΙΣΜΟΣ

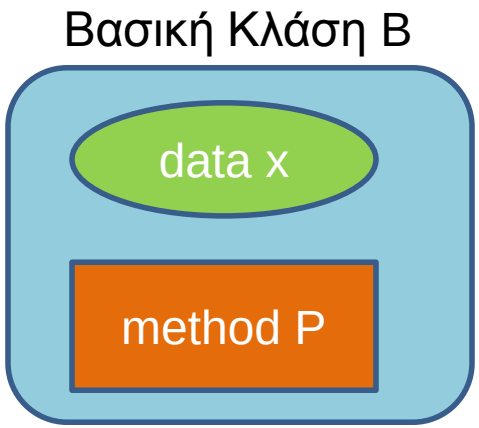
ΑΝΑΚΕΦΑΛΑΙΩΣΗ

Θεματολόγιο

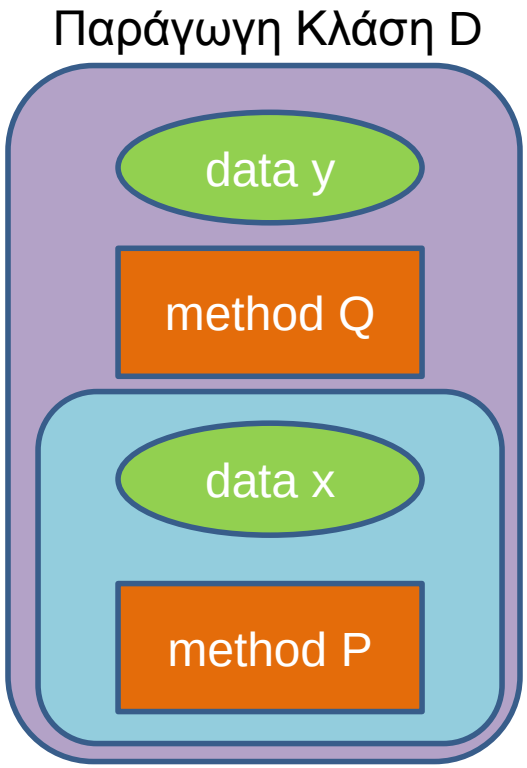
- Κληρονομικότητα
  - Παράδειγμα
  - Κληρονομικότητα – Βελτιωμένο Παράδειγμα
  - Ενθυλάκωση : public – private - protected πεδία
  - Ενθυλάκωση : public – private - protected κληρονομικότητα
  - Ανάθεση αντικειμένων και κληρονομικότητα
  - Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και κληρονομικότητα
  - Πέρασμα παραμέτρων και κληρονομικότητα
  - Άλλα θέματα

Κληρονομικότητα

Έχουμε μια Βασική Κλάση (Base Class) Β, με κάποια πεδία και μεθόδους.



Θέλουμε να δημιουργήσουμε μια νέα κλάση D η οποία να έχει όλα τα χαρακτηριστικά της Β, αλλά και κάποια επιπλέον.



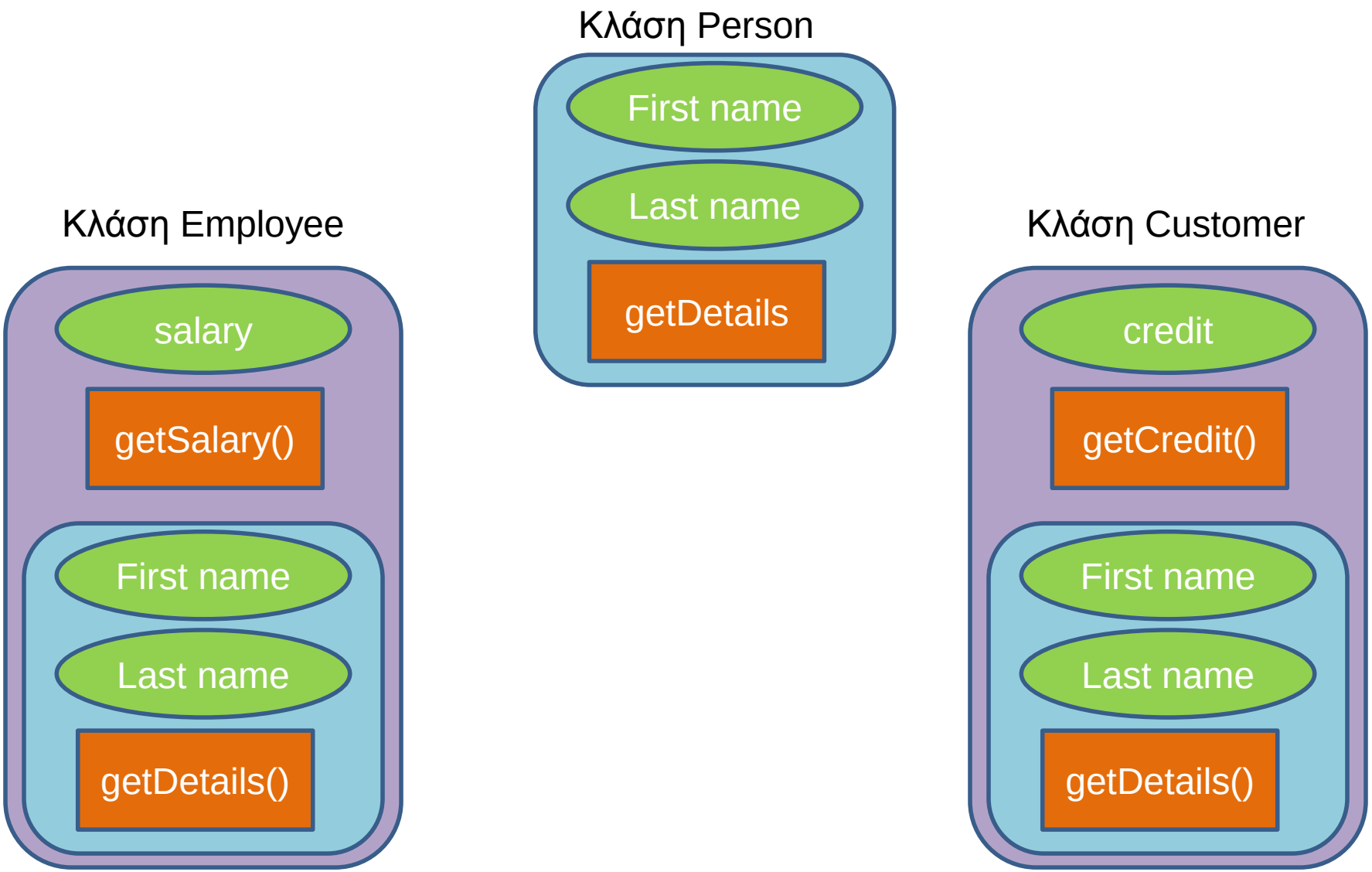
Αντί να ξαναγράψουμε τον ίδιο κώδικα δημιουργούμε μια Παράγωγη Κλάση (Derivative Class) D, η οποία κληρονομεί όλη τη λειτουργικότητα της Βασικής Κλάσης Β και στην οποία προσθέτουμε τα νέα πεδία και μεθόδους.

Αυτή είναι η ιδιότητα της κληρονομικότητας

Κληρονομικότητα

- Η κληρονομικότητα είναι χρήσιμη όταν
  - Θέλουμε να έχουμε αντικείμενα και της κλάσης B και της κλάσης D.
  - Θέλουμε να ορίσουμε πολλαπλές παράγωγες κλάσεις D1, D2, … που η κάθε μία επεκτείνει την B με διαφορετικό τρόπο.

Παράδειγμα


Κλάση Person
First name
Last name
getDetails
Κλάση Employee
salary
getSalary()
First name
Last name
getDetails()
Κλάση Customer
credit
getCredit()
First name
Last name
getDetails()

Παράδειγμα

```
class Person
{
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
  ~Person();
   char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[])
{
  strcpy(fname, fn);
  strcpy(lname, ln);
}

Person::~Person()
{
 cout << "~Person() called\n";
}

char *Person::getPersonalDetails()
{
  char *ret = new char [strlen(fname) + strlen(lname) +40];
  sprintf(ret, "1st Name %s -Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
class Employee : public Person
{
private:
   int basicSalary;

public:
   Employee(char fn[], char ln[], int sal);
   ~Employee();
   int getSalary();
};

Employee::Employee(char fn[],char ln[],int sal):Person(fn,ln)
{
   basicSalary = sal;
}

Employee::~Employee(){ cout << "~Employee() called"; }

int Employee::getSalary()
{
   return basicSalary;
}
```

Παράδειγμα

```
class Customer : public Person
{
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  ~Customer();
  void chargeCredit(int amount);
  int getCredit();
};

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln)
{
  credit = 0;
  strcpy(creditType, ct);
}

Customer::~Customer(){ cout << "~Customer called\n"; }

void Customer::chargeCredit(int amount)
{
  credit -= amount;
}

int Customer::getCredit()
{
  return credit;
}
```

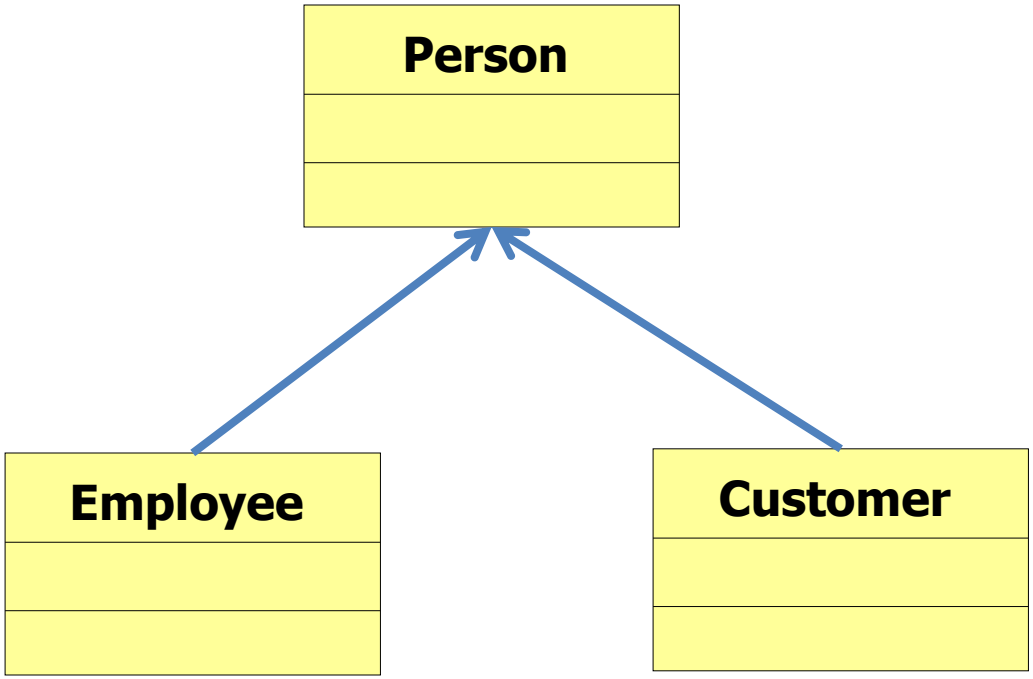
Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = john.getPersonalDetails();
  sal = john.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  pet.chargeCredit(250);
  // οι destructors εκτελούνται αντίστροφα ….
}
```

Κληρονομικότητα

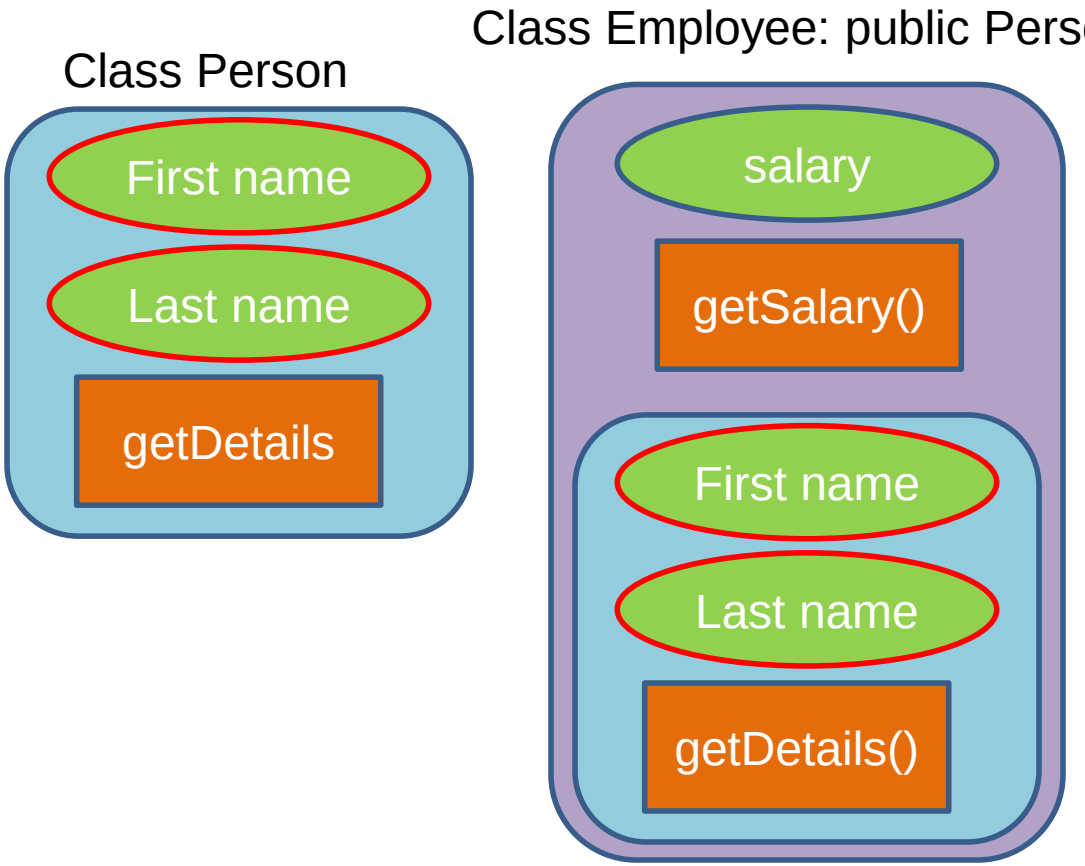
- Μπορούμε να χρησιμοποιούμε την ιδιότητα της κληρονομικότητας με διάφορους τρόπους αλλά ο πιο χρήσιμος και κατανοητός είναι για να ορίσουμε μια εξειδίκευση/επέκταση μιας κλάσης.
  - Σχέση is-a ή is-like-a

UML διαγράμματα



Παράδειγμα – protected members

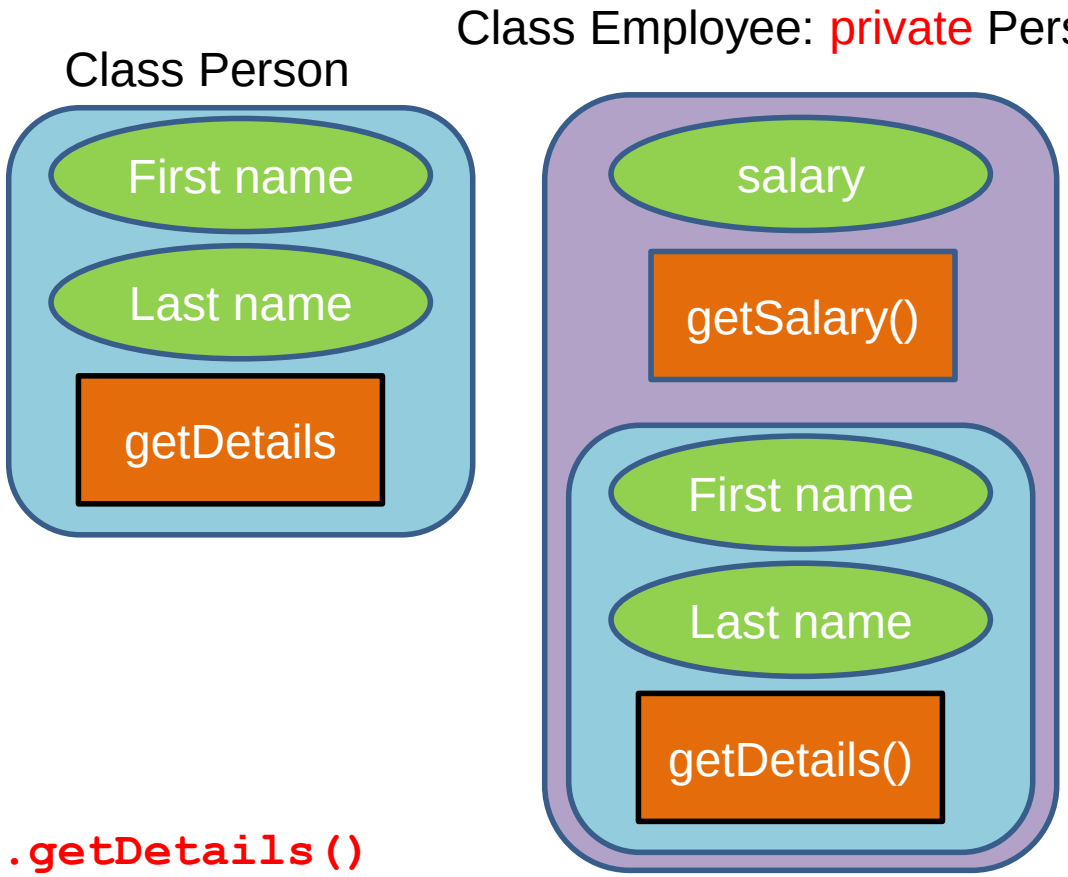
Για να έχει πρόσβαση στα πεδία (ή μεθόδους) της Person η κλάση Employee θα πρέπει να είναι ορισμένα ως Public, ή Protected




```
class Person
{
protected:
   char fname[40];
   char lname[40];
public:
   Person(char fn[],
  char ln[]);
  ~Person();
   char
  *getPersonalDetails();
};
```

Παράδειγμα – private inheritance

Η μέθοδος getDetails() της κλάσης Person, δεν είναι πλέον προσβάσιμη από αντικείμενα της κλάσης Employee




```
int main()
{
    Employee emp;
    // cannot call emp.getDetails()
}
```

Γιατί αυτό είναι χρήσιμο?

```
class Array
{
protected:
  int *A;
  int size;
public:
  Array(int s);
  GetElement(int i);
  ~Array();
};
```

```
class SafeArray: private Array
{
public:
  GetElementSafely(int i);
};

SafeArray::GetElementSafely(int i)
{
  if (i < 0 || i >= size){
    exit(1);
  }
  return A[i];
}
```

Η SafeArray δεν επιτρέπει την χρήση της GetElement(i)

Ανάθεση Αντικειμένων και Κληρονομικότητα

- Στην ανάθεση μεταξύ δύο αντικειμένων x = y πρέπει
  - τα δύο αντικείμενα να είναι της ίδιας κλάσης ή
  - η κλάση του αντικειμένου που ανατίθεται (y) να κληρονομεί (άμεσα ή έμμεσα) από την κλάση του αντικειμένου στο οποίο γίνεται η ανάθεση (x)
    - Δηλαδή το y να παράγεται (έμμεσα η άμεσα) από την κλάση x
    - Στην περίπτωση αυτή λέμε ότι κάναμε upcasting.

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  cout << john.getPersonalDetails() << "-Salary: " <<
 john.getSalary() << endl;

  pet.chargeCredit(250);
  cout << pet.getPersonalDetails() << "-Credit: " <<
 pet.getCredit() << endl;

  ……
```

Παράδειγμα

```
……..……..……..……..……..……..……..……..……..……..
  Person johnPrivateLife = john; // upcasting
  Person petPrivateLife = pet;       // upcasting

  cout << johnPrivateLife.getPersonalDetails() << endl;
  cout << petPrivateLife.getPersonalDetails() << endl;
  // petPrivateLife.chargeCredit(130);
  // compile error ..

  cin >> fname >> lname;
  petPrivateLife.setPersonalDetails(fname, lname);

  cout << petPrivateLife.getPersonalDetails() << endl;
  cout << pet.getPersonalDetails() << "-Credit: " <<
 pet.getCredit() << endl;
}
```

Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και κληρονομικότητα

- Στην ανάθεση της διεύθυνσης ενός αντικειμένου σε ένα pointer p = & x πρέπει
  - ο p να είναι pointer της κλάσης στην οποία ανήκει το αντικείμενο x ή
  - pointer κλάσης από την οποία κληρονομεί η κλάση του αντικειμένου x.
- Το ποιες μέθοδοι μπορούν να κληθούν μέσω του pointer, καθορίζεται από τον τύπο του pointer και όχι από τον τύπο του αντικειμένου.

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  cout << john.getPersonalDetails() << "-Salary: "
       << john.getSalary() << endl;

  pet.chargeCredit(250);
  cout << pet.getPersonalDetails() << "-Credit: "
       << pet.getCredit() << endl;

  ……
```

Παράδειγμα

```
.....
   Person *johnPrivateLife = &john;
   Person *petPrivateLife = &pet;

   cout << johnPrivateLife->getPersonalDetails() << endl;
   cout << petPrivateLife->getPersonalDetails() << endl;
   // petPrivateLife->chargeCredit(130);
   // compile error ..

   cin >> fname >> lname;
   petPrivateLife->setPersonalDetails(fname, lname);

   cout << petPrivateLife->getPersonalDetails() << endl;
   cout << pet.getPersonalDetails() << "-Credit: "
        << pet.getCredit() << endl;
}
```

Θεματολόγιο

- Κληρονομικότητα
  - Παράδειγμα
  - Κληρονομικότητα – Βελτιωμένο Παράδειγμα
  - Ενθυλάκωση : public – private - protected πεδία
  - Ενθυλάκωση : public – private - protected κληρονομικότητα
  - Ανάθεση αντικειμένων και κληρονομικότητα
  - Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και κληρονομικότητα
  - Πέρασμα παραμέτρων και κληρονομικότητα
  - Υποσκέλιση (overriding) συναρτήσεων

ΠΕΡΑΣΜΑ ΠΑΡΑΜΕΤΡΩΝ ΚΑΙ ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ

Πέρασμα παραμέτρων και Κληρονομικότητα

- Είδαμε στα προηγούμενα παραδείγματα ότι
  - Σε ένα αντικείμενο βασικής κλάσης μπορεί να ανατεθεί ένα αντικείμενο παραγόμενης κλάσης
  - Σε ένα δείκτη σε αντικείμενα βασικής κλάσης μπορεί να ανατεθεί η διεύθυνση ενός αντικειμένου παραγόμενης κλάσης
- Η δυνατότητα του upcasting αποκτά εξαιρετική χρησιμότητα στην περίπτωση που το αντικείμενο ή ο δείκτης βασικής κλάσης είναι παράμετρος σε μια άλλη συνάρτηση ή μέθοδο του προγράμματος μας.
  - Η συνάρτηση μας μπορεί να εκτελεστεί επιτυχημένα χωρίς καμία αλλαγή για αντικείμενα οποιασδήποτε κλάσης παράγεται από τη βασική κλάση….
  - => η συνάρτηση μας είναι επαναχρησιμοποιήσιμη

Παράδειγμα

- Έστω ότι στο πρόγραμμά μας θέλουμε να προσθέσουμε την επιπλέον δυνατότητα να μετατρέπει τις πληροφορίες που διαχειρίζεται για τους πελάτες και τους υπαλλήλους σε html σελίδες και να τις εκτυπώνει ….

Παράδειγμα – Πέρασμα δια τιμής

```
void HTMLFormattedPrint( Person p ){
  cout << "<HTML>" << p.getPersonalDetails()
       << "</HTML>" << endl;
  p.setPersonalDetails(“”, “”);
}
```

για να δούμε τη διαφορά περάσματος δια τιμής, και περάσματος με δείκτη ή αναφορά

Παράδειγμα – Πέρασμα δια τιμής

```
int main(){
   char fname[40];
   char lname[40];
   int sal;
   int credit;

   cin >> fname >> lname >> sal;
   Employee john(fname, lname, sal);
   cin >> fname >> lname;
   Customer pete(fname, lname, "VISA");

   ……..
   HTMLFormattedPrint(pete);
   HTMLFormattedPrint(john);

   cout << pete.getPersonalDetails()
        << " - Credit : " << pete.getCredit() << endl;
   return 0;
}
```

Η ίδια συνάρτηση καλείται για Customers και Employees μέσω upcasting

Το αντικείμενο περνάει δια τιμής – οι αλλαγές δεν είναι εμφανείς στη main ()

Παράδειγμα – Πέρασμα δείκτη

```
void HTMLFormattedPrint( Person *p ){
  cout << "<HTML>" << p->getPersonalDetails()
       << "</HTML>" << endl;
  p->setPersonalDetails("", "");
}
```

Παράδειγμα – Πέρασμα δείκτη

```
int main(){
   char fname[40];
   char lname[40];
   int sal;
   int credit;

   cin >> fname >> lname >> sal;
   Employee john(fname, lname, sal);
   cin >> fname >> lname;
   Customer pete(fname, lname, "VISA");

   ……..
   HTMLFormattedPrint(&pete);
   HTMLFormattedPrint(&john);

   cout << pete.getPersonalDetails() <<
        " - Credit : " << pete.getCredit() << endl;
   return 0;
}
```

Το αντικείμενο περνάει δια τιμής –
οι αλλαγές είναι εμφανείς στη main ()

Παράδειγμα

- Λόγω της κληρονομικότητας, μια μόνο επιπλέον συνάρτηση μας αρκεί για να υλοποιήσουμε την επιπλέον δυνατότητα τόσο για τους πελάτες όσο και για τους υπάλληλους του καταστήματος….

Παράδειγμα

- Παρόλα αυτά το πρόγραμμά μας μπορεί να μετατρέψει σε html μόνο τις πληροφορίες που περιλαμβάνονται σαν χαρακτηριστικά στη βασική κλάση Person …. ΓΙΑΤΙ ??
  - Τι γίνεται αν θέλουμε να τυπώνουμε και τα επιμέρους χαρακτηριστικά που περιλαμβάνουν οι κλάσεις Employee και Customer ?

METHOD OVERRIDING ΚΑΙ ΠΟΛΥΜΟΡΦΙΣΜΟΣ

Παράδειγμα

```
using namespace std;

class Person {
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
   char *getPersonalDetails();
   void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person
{
private:
   int basicSalary;

public:
   Employee(char fn[], char ln[], int sal);
   int getSalary();
};

class Customer : public Person
{
private:
   int credit;
   char creditType[10];
public:
   Customer(char fn[], char ln[], char ct[]);
   void chargeCredit(int amount);
   int getCredit();
   char *getCreditType();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) +40];
  sprintf(ret, "1st Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) :
 Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) :
  Person(fn, ln) {
   credit = 0;
   strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
   credit -= amount;
}

int Customer::getCredit(){
   return credit;
}

char *Customer::getCreditType(){
   return creditType;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pete(fname, lname, "VISA");

  pete.chargeCredit(250);

  cout << john.getPersonalDetails() << " "
       << john.getSalary() << endl;

  cout << pete.getPersonalDetails() << " "
       << pete.getCreditType() << " "
       << pete.getCredit() << endl;

  return 0;
}
```

Παράδειγμα

- Η σχεδίαση του κώδικα μας δεν είναι πλέον καλή γιατί χρειαζόμαστε διαφορετικές εντολές για να τυπώσουμε τα στοιχεία αντικειμένων από διαφορετικές κλάσεις.
  - Μια νέα παραγόμενη κλάση θα δημιουργούσε επιπλέον προβλήματα.
  - Θέλουμε μια μοναδική συνάρτηση που να μας δίνει τα στοιχεία του αντικειμένου.

Method Overriding

- Σε κάθε μία από τις παραγόμενες κλάσεις θα επανα-ορίσουμε την μέθοδο getPersonalDetails()
- Ο ορισμός στην παραγόμενη κλάση υποσκελίζει (overrides) τον ορισμό στη βασική κλάση.

Παράδειγμα

```
# include <iostream>
# include <cstring>

using namespace std;

class Person {
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
   char *getPersonalDetails();
   void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person
{
private:
  int basicSalary;
public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};

class Customer : public Person
{
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname)+ 40];
  sprintf(ret, "1st Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln)
{
  basicSalary = sal;
}
int Employee::getSalary(){
  return basicSalary;
}


char *Employee::getPersonalDetails(){
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + 10];
  sprintf(ret, "%s %d", name, basicSalary);
  return ret;
}
```

Εδώ καλείται και η μέθοδος που επανα-ορίσαμε.
Για να μην έχουμε αναδρομή, καθορίζουμε ότι καλούμε τη μέθοδο που κληρονομούμε από την Person

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]): Person(fn, ln)
{
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}

char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getPersonalDetails(){
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + strlen(creditType) + 10];
  sprintf(ret, "%s %s %d", name, creditType, credit);
  return ret;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pete(fname, lname, "VISA");

  pete.chargeCredit(250);

  cout << john.getPersonalDetails() << endl;

  cout << pete.getPersonalDetails() << endl;

  return 0;
}
```

Πιο περίπλοκο παράδειγμα

```
int main(){
  Person *all[10];

  for (int i = 0; i < 10; i++){
    char fname[40];
    char lname[40];
    int sal;
    char choice;
    cin >> choice;
    if (choice == 'p'){
       cin >> fname >> lname;
       all[i] = new Person(fname, lname);
    }
    if (choice == 'e'){
       cin >> fname >> lname >> sal;
       all[i] = new Employee(fname, lname, sal);
    }
    if (choice == 'c'){
       cin >> fname >> lname;
       all[i] = new Customer(fname, lname, "VISA");
     }
  }

 for (int i = 0; i < 10; i++){
   cout << all[i]->getPersonalDetails() << endl;
 }
}
```

Τι output θα πάρουμε?

Πιο περίπλοκο παράδειγμα

```
int main(){
  Person *all[10];

  for (int i = 0; i < 10; i++){
    char fname[40];
    char lname[40];
    int sal;
    char choice;
    cin >> choice;
    if (choice == 'p'){
       cin >> fname >> lname;
       all[i] = new Person(fname, lname);
    }
    if (choice == 'e'){
       cin >> fname >> lname >> sal;
       all[i] = new Employee(fname, lname, sal);
    }
    if (choice == 'c'){
       cin >> fname >> lname;
       all[i] = new Customer(fname, lname, "VISA");
     }
  }
```

Θέλω να καλώ την επανα-ορισμένη μέθοδο της αντίστοιχης παραγόμενης κλάσης

Καλείται η μέθοδος της Person

```
 for (int i = 0; i < 10; i++){
   cout << all[i]->getPersonalDetails() << endl;
 }
}
```

Εξήγηση

- Στη C++ αν έχω:
  - μια βασική κλάση A και παραγόμενες κλάσεις B, Γ οι οποίες επανα-ορίζουν μια μέθοδο f() της A
  - κώδικα (συνάρτηση, μέθοδο, κλπ.) στον οποίο
    - χρησιμοποιείται ένας δείκτης p βασικής κλάσης A που μπορεί να δείχνει σε αντικείμενα των κλάσεων A, B, ή Γ
    - μέσω του δείκτη καλείται η μέθοδος f() στο αντικείμενο στο οποίο δείχνει ο δείκτης p
  - η υλοποίηση της μεθόδου f() που θα καλεστεί καθορίζεται από τον τύπο του δείκτη
    - δηλαδή καλείται η υλοποίηση της μεθόδου f() της βασικής κλάσης A
  - η επιλογή γίνεται από τον compiler κατά τη διάρκεια της μετάφρασης του προγράμματος (early binding)

Virtual Methods and Late Binding

- Για να κάνουμε την επιλογή της μεθόδου f() που θα καλεστεί να καθορίζεται από τον τύπο του αντικειμένου στο οποίο δείχνει ο δείκτης p (δηλαδή να καλείται η υλοποίηση της μεθόδου f() των παραγόμενων κλάσεων Β, Γ) πρέπει η μέθοδος f() να δηλωθεί στην κλάση Α ως virtual μέθοδος.
- Στην περίπτωση των virtual μεθόδων δίνεται η οδηγία στον compiler ότι η επιλογή της υλοποίησης της f() δεν θα γίνει κατά τη μετάφραση του προγράμματος, με βάση τον τύπο του δείκτη, αλλά κατά την εκτέλεση του προγράμματος με βάση τον τύπο του αντικειμένου στο οποίο δείχνει ο δείκτης κάθε φορά που εκτελείται ο κώδικας (late binding).

Παράδειγμα

```
# include <iostream>
# include <cstring>

using namespace std;

class Person {
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
   virtual char *getPersonalDetails();
   void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person
{
private:
  int basicSalary;
public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};


class Customer : public Person
{
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) +40];
  sprintf(ret, "1st Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[],char ln[],int sal):Person(fn, ln)
{
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getPersonalDetails(){
 // εδώ καλείται και η μέθοδος που επανα-ορίσαμε
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + 10];
  sprintf(ret, "%s %d", name, basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]): Person(fn, ln)
{
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}

char *Customer::getCreditType(){
  return creditType;
}


char *Customer::getPersonalDetails(){
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + strlen(creditType)+ 10];
  sprintf(ret, "%s %s %d", name, creditType, credit);
  return ret;
}
```

Παράδειγμα

Καλείται η μέθοδος του αντίστοιχου αντικειμένου

```
int main(){
  Person *all[10];

  for (int i = 0; i < 10; i++){
    char fname[40];
    char lname[40];
    int sal;
    char choice;
    cin >> choice;
    if (choice == 'p'){
       cin >> fname >> lname;
       all[i] = new Person(fname, lname);
    }
    if (choice == 'e'){
       cin >> fname >> lname >> sal;
       all[i] = new Employee(fname, lname, sal);
    }
    if (choice == 'c'){
       cin >> fname >> lname;
       all[i] = new Customer(fname, lname, "VISA");
    }
  }

  for (int i = 0; i < 10; i++){
    cout << all[i]->getPersonalDetails() << endl;
  }
}
```

Παράδειγμα – Πέρασμα δείκτη

```
void HTMLFormattedPrint( Person *p ){
  cout << "<HTML>" << p->getPersonalDetails()
       << "</HTML>" << endl;
}
```

Τι θα παίρναμε αν δεν είχαμε ορίσει την virtual μέθοδο?

Με τη virtual μέθοδο, η συνάρτηση πλέον δουλεύει για οποιοδήποτε αντικείμενο κλάσης Person, Employee, Customer και τυπώνει σωστά τα αντίστοιχα πεδία.
Η συνάρτηση **`HTMLFormattedPrint`** είναι πλέον επαναχρησιμοποιήσιμη για οποιαδήποτε παράγωγη κλάση της Person.

Πολυμορφισμός

- Τα παραπάνω είναι παραδείγματα πολυμορφισμού.
- Τι είναι πολυμορφισμός?
  - (πολυμορφισμός = πολλές μορφές. Η χρήση μιας μεθόδου με διαφορετικούς τρόπους).
- Δύο είδη πολυμορφισμού:
  - Στατικός Πολυμορφισμός (compile time – early binding)
    - Method Overloading
    - Method Overriding
  - Δυναμικός Πολυμορφισμός (run time – late binding)
    - Virtual methods που αποφασίζονται δυναμικά με βάσει το αντικείμενο.

ΠΟΛΥΜΟΡΦΙΣΜΟΣ ΚΑΙ ΑΦΗΡΗΜΕΝΕΣ ΚΛΑΣΕΙΣ

Παράδειγμα

- Έστω ότι στο πρόγραμμά μας θέλουμε να προσθέσουμε μια μέθοδο που μετατρέπει τις πληροφορίες που διαχειρίζεται για τους πελάτες και τους υπαλλήλους σε html σελίδες και να τις εκτυπώνει.
  - Η μέθοδος παίρνει μία παράμετρο που παίρνει τιμή 1, ή 2:
    - Με την επιλογή 1, τυπώνει μόνο τα βασικά χαρακτηριστικά της κλάσης Person
    - Με την επιλογή 2, τυπώνει μόνο τα οικονομικά επιμέρους χαρακτηριστικά που περιλαμβάνουν οι κλάσεις Employee και Customer

Παράδειγμα

```
# include <iostream>
# include <cstring>

using namespace std;

class Person
{
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
   char *getPersonalDetails();
   virtual char* getFinancialDetails();
   // εικονική μέθοδος χωρίς υλοποίηση
   void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person
{
private:
  int basicSalary;
public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getFinancialDetails();
};

class Customer : public Person
{
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getFinancialDetails();
};
```

Παράδειγμα

```
Person::Person (char fn[], char ln[])
{
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[])
{
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails()
{
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "1st Name %s - Last Name %s", fname, lname);

  return ret;
}

char* getFinancialDetails() {
  return NULL;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) :
                    Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getFinancialDetails(){
  char * ret = new char [30];
  sprintf(ret, " Basic Salary %d ", basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln)
{
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}

char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getFinancialDetails(){
  char * ret = new char [strlen(creditType) + 40];
  sprintf(ret, "Credit Type %s Credit %d", creditType, credit);
  return ret;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p, int choice)
{
  if(choice == 1){
    cout << "<HTML>" << p->getPersonalDetails()
         << "</HTML>" << endl;
  }
  else {
    if(choice == 2){
      cout << "<HTML>" << p->getFinancialDetails()
           << "</HTML>" << endl;
    }
  }
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;
  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pete(fname, lname, "VISA");
  pet.chargeCredit(250);
  HTMLFormattedPrint (&pete, 1);
  HTMLFormattedPrint (&john, 1);
  HTMLFormattedPrint (&pete, 2);
  HTMLFormattedPrint (&john, 2);
  return 0;
}
```

Εικονικές μεθόδοι

- Στην ουσία κατασκευάσαμε μια βασική κλάση που περιλαμβάνει μια τετριμμένη εικονική μέθοδο **`getFinancialDetails`** μόνο και μόνο για να μπορούμε να την επανα-ορίσουμε στις παραγόμενες κλάσεις και να την καλούμε στην μέθοδο **`HTMLFormattedPrint`**
  - Η τετριμμένη μέθοδος όμως επιστρέφει NULL και είναι αρκετά επικίνδυνη …
  - Κανείς δεν μπορεί να εγγυηθεί ότι ένας προγραμματιστής που θα επεκτείνει την κλάση Person θα επανα-ορίσει την **`getFinancialDetails`**
    - αν δεν επανα-ορίστει η μέθοδος, η NULL τιμή που επιστρέφει μπορεί να δημιουργήσει προβλήματα….
  - Θέλουμε να εξασφαλίσουμε ότι η **`getFinancialDetails`** πάντα θα επανα-ορίζεται και δεν θα χρησιμοποιείται ποτέ η τετριμένη μέθοδος

Αφηρημένες βασικές κλάσεις

- Για να εξασφαλίσουμε ότι όσοι επεκτείνουν την κλάση Person θα επανα-ορίσουν την τετριμμένη μέθοδο **`getFinancialDetails`** ορίζουμε μια αφηρημένη βασική κλάση.
- Μια κλάση ονομάζεται αφηρημένη αν περιέχει τουλάχιστον μια εικονική μέθοδο που δεν περιλαμβάνει υλοποίηση
  - δηλώνεται ως virtual methodName() = 0;
- Η αφηρημένη κλάση λειτουργεί σαν καλούπι για την κατασκευή παραγόμενων που προσφέρουν εναλλακτικές υλοποιήσεις στις εικονικές μεθόδους.
- Η δημιουργία αντικειμένων αφηρημένης κλάσης ΔΕΝ επιτρέπεται από τον compiler
- Η μη υλοποίηση εικονικών μεθόδων ΔΕΝ επιτρέπεται από τον compiler

Παράδειγμα

```
# include <iostream>
# include <cstring>

using namespace std;

class Person {
private:
   char fname[40];
   char lname[40];
public:
   Person(char fn[], char ln[]);
   char *getPersonalDetails();
   virtual char* getFinancialDetails() = 0;
   // γνησια εικονική μέθοδος χωρίς υλοποίηση
   void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person
{
private:
  int basicSalary;
public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getFinancialDetails();
};

class Customer : public Person
{
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getFinancialDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) +40];
  sprintf(ret, "1st Name %s - Last Name %s", fname, lname);

  return ret;
}

// Δεν υπάρχει υλοποιηση της getFinancialDetails
// ούτε καν τετριμένη!!!
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) :
                    Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getFinancialDetails(){
  char * ret = new char [30];
  sprintf(ret, " Basic Salary %d ", basicSalary);

  return ret;
}

// Αν δεν ορίσουμε την getFinancialDetails θα έχουμε error
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
char *Customer::getCreditType(){
  return creditType;
}
```

```
char *Customer::getFinancialDetails(){
   char * ret = new char [strlen(creditType) + 40];
   sprintf(ret, "Credit Type %s Credit %d", creditType, credit);
   return ret;
}

// Αν δεν ορίσουμε την getFinancialDetails θα έχουμε error
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p, int choice){
  if(choice == 1)
    cout << "<HTML>" << p->getPersonalDetails() <<
              "</HTML>" << endl;
  else
    if(choice == 2)
      cout << "<HTML>" << p->getFinancialDetails() <<
                  "</HTML>" << endl;
}
```

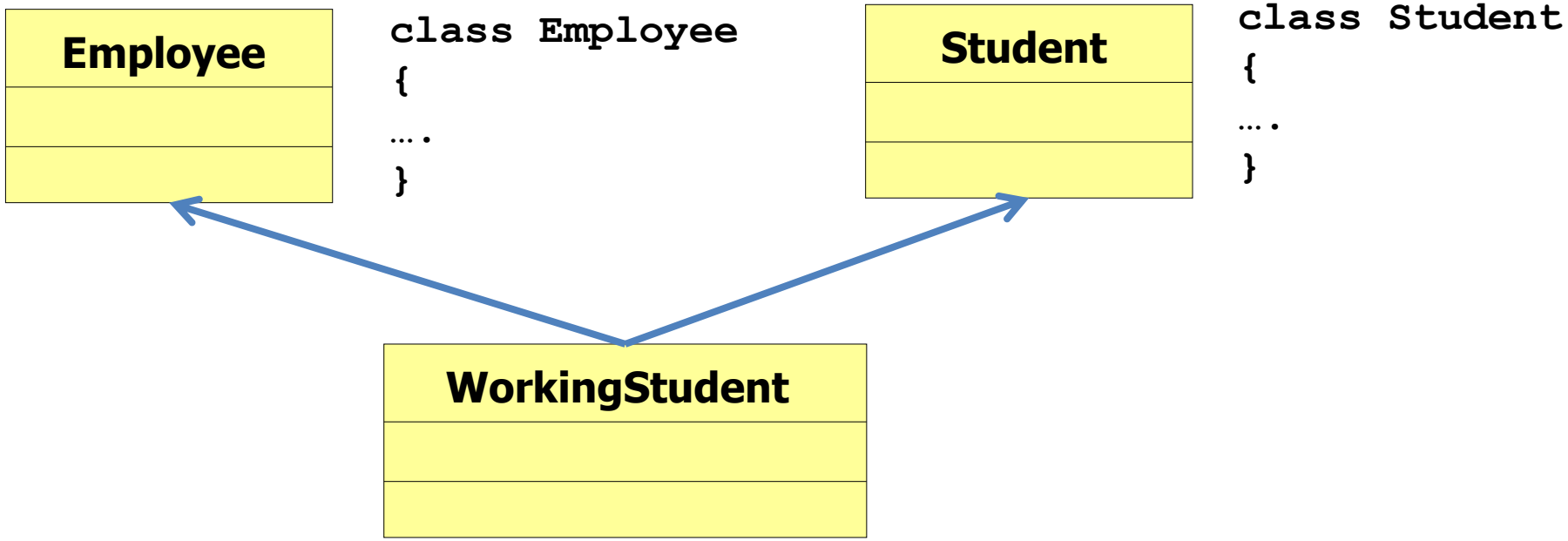
Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;

  cin >> fname >> lname >> sal;
  Employee john(fname, lname, sal);
  cin >> fname >> lname;
  Customer pete(fname, lname, "VISA");
  pete.chargeCredit(250);

  // Person mary(fname, lname) θα δώσει compile error!

  HTMLFormattedPrint (&pete, 1);
  HTMLFormattedPrint (&john, 1);
  HTMLFormattedPrint (&pete, 2);
  HTMLFormattedPrint (&john, 2);

  return 0;
}
```

ΑΛΛΑ ΘΕΜΑΤΑ

Πολλαπλή κληρονομικότητα

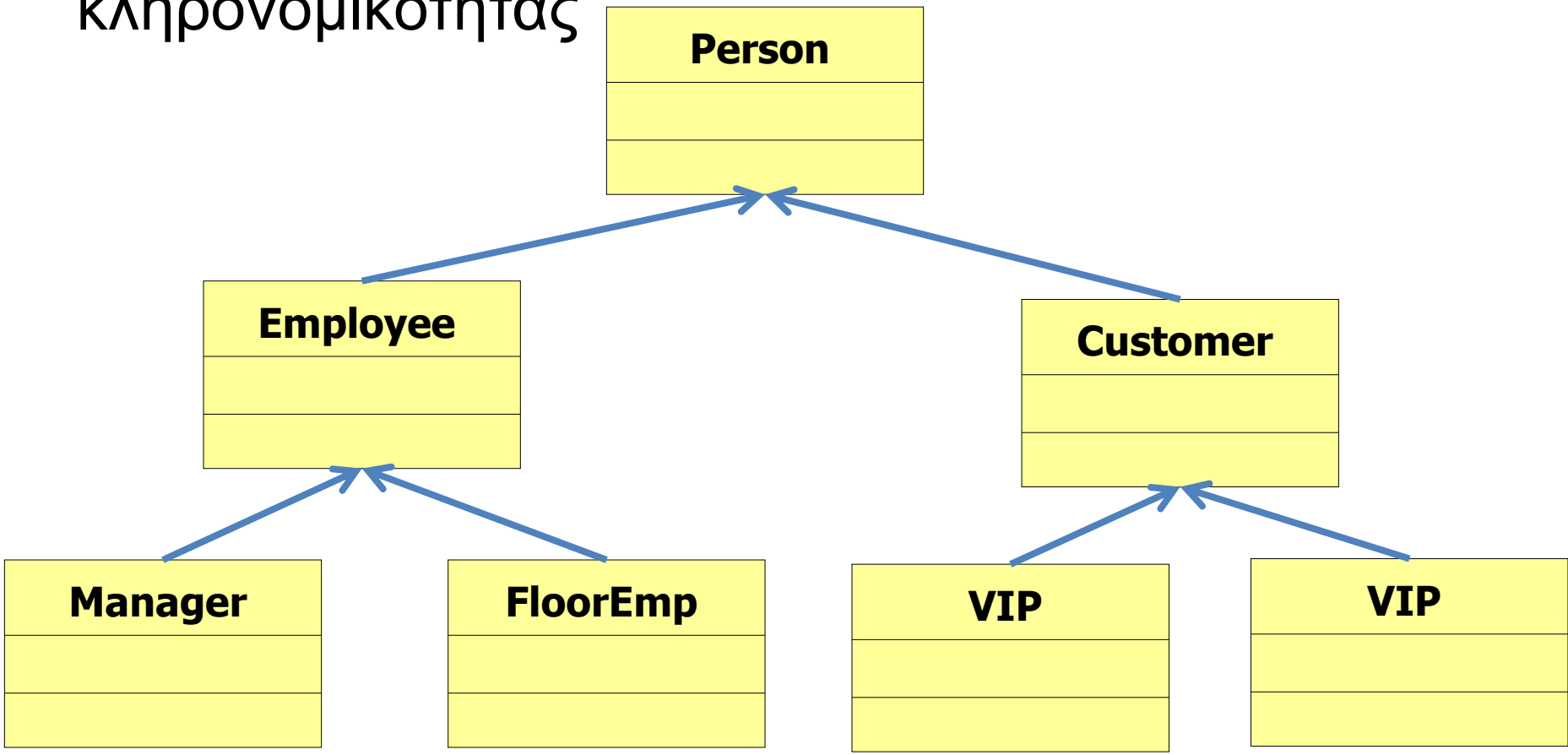
- Μία κλάση μπορεί να κληρονομεί (να παράγεται) από πολλαπλές άλλες κλάσεις



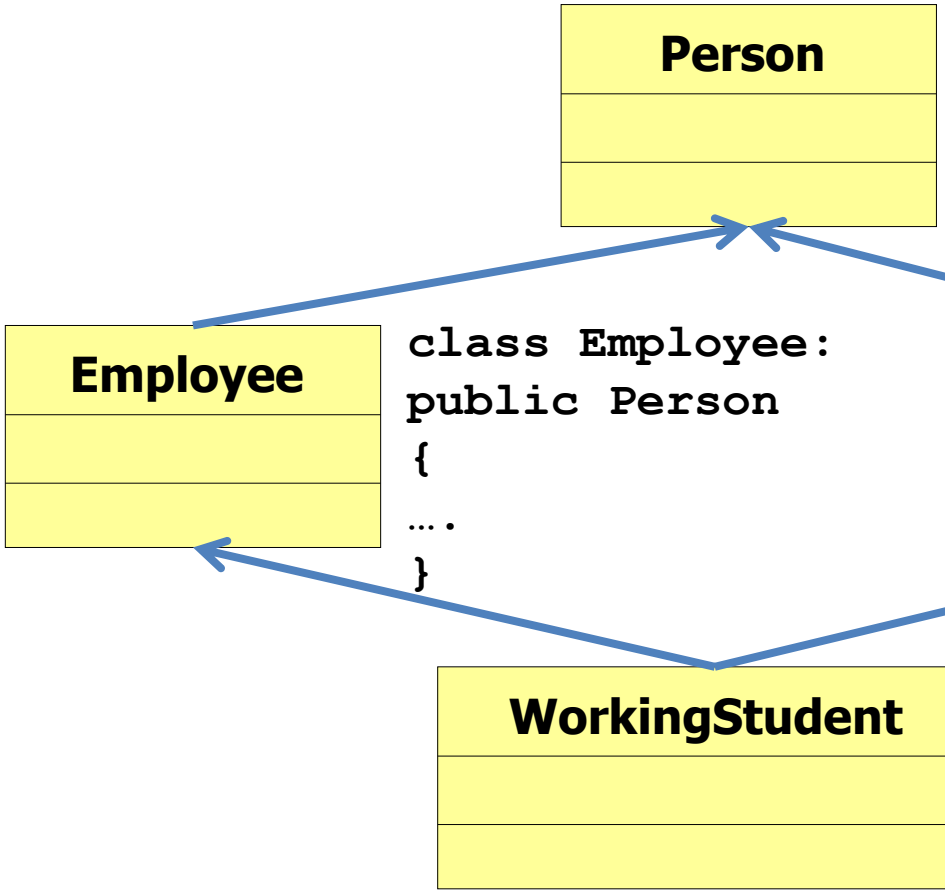

```
class WorkingStudent: public Employee, public Student
{
….
}
```

Επίπεδα κληρονομικότητας

- Μπορούμε επίσης να έχουμε πολλά επίπεδα κληρονομικότητας



Και συνδυασμό των δύο




```
class Person
{
….
}
```

```
class Employee:
public Person
{
….
}
```

```
class Student:
public Person
{
….
}
```

```
class WorkingStudent: public Employee, public Student
{
….
}
```

Πολλαπλή Κληρονομικότητα - Αμφισημία

```
class Base1{
 public:
      void f();
};
class Base2{
 public:
      void f();
};
```

```
main()
{
  Derived d;
  d.f();
  // υπάρχει αμφισημία!
}
```

```
class Derived: public Base1, public Base2
{
       ...
}
```

Πολλαπλή Κληρονομικότητα - Αμφισημία

```
class Base1{
 public:
      void f();
};
class Base2{
 public:
      void f();
};
```

```
main()
{
  Derived d;
  d.Base1::f();
  d.Base2::f();
}
```

```
class Derived: public Base1, public Base2
{
       ...
}
```