Κληρονομικότητα

Η κληρονομικότητα (inheritance) αποτελεί έναν από τους χαρακτηριστικότερους μηχανισμούς των αντικειμενοστρεφών γλωσσών προγραμματισμού. Επιτρέπει την δημιουργία μιας νέας κλάσης απορροφώντας "κληρονομώντας" όλες τις ιδιότητες και μεθόδους μιας υπάρχουσας κλάσης. Με τον τρόπο αυτό μπορούμε να δημιουργούμε "ειδικότερες" κατηγορίες αντικειμένων προσθέτοντας επιπλέον χαρακτηριστικά ή λειτουργίες.

Έστω για παράδειγμα η κλάση Employee που αναφέρεται σε έναν υπάλληλο επιχείρησης, με μοναδικές ιδιότητες (χάριν απλότητας) το όνομα και το μηνιαίο μισθό του υπαλλήλου, και μοναδική μέθοδο (πλην του κατασκευαστή) την printStatement() που εκτυπώνει το όνομα και το μηνιαίο μισθό:

```
public class Employee {

	private String name;
	private double salary;

	public Employee(String text, double amount) {
		name = text;
		salary = amount;
	}

	public void printStatement() {
		System.out.println("Employee: " + name + " with salary: " + salary);
	}
}
```

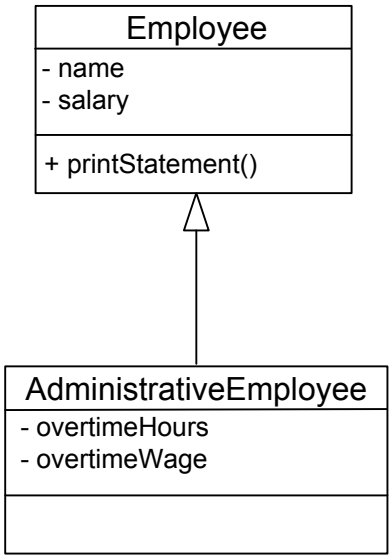
Αν θεωρήσουμε ότι το σύστημα πρέπει να διαχειρίζεται και μια ειδικότερη κατηγορία υπαλλήλου (π.χ. Διοικητικό Υπάλληλο) ο οποίος πέραν του ονόματος και του μισθού έχει επιπρόσθετα ως ιδιότητες τις ώρες υπερωριακής απασχόλησης ανά μήνα καθώς και την αμοιβή ανά ώρα υπερωρίας, μια λύση θα ήταν να "αντιγράψουμε" την κλάση Employee σε μια νέα κλάση και να προσθέσουμε τα επιπλέον χαρακτηριστικά. Ωστόσο, λαμβάνοντας υπόψη ότι στην ανάπτυξη λογισμικού αποφεύγουμε τη δημιουργία "κλώνων" (τμημάτων κώδικα που επαναλαμβάνονται) η ενδεικνυόμενη λύση είναι η χρήση κληρονομικότητας, δηλαδή η δημιουργία μιας νέας κλάσης AdministrativeEmployee που κληρονομεί την υπάρχουσα:

```
public class AdministrativeEmployee extends Employee{

	private int overtimeHours;
	private double overtimeWage;

}
```

Η νέα κλάση "απορροφά" από την υπάρχουσα όλες τις ιδιότητες και τη συμπεριφορά της και μπορεί να δηλώσει επιπλέον ιδιότητες ή/και μεθόδους. Σε ένα διάγραμμα κλάσεων της UML ο συμβολισμός της κληρονομικότητας πραγματοποιείται με τη χρήση μιας προσανατολισμένης ακμής (με κλειστό τρίγωνο στο άκρο της κλάσης από την οποία κληρονομούμε) ως εξής:



Η κλάση Employee από την οποία κληρονομεί η AdministrativeEmployee ονομάζεται υπερκλάση (και συνήθως απεικονίζεται ψηλότερα) ενώ η κλάση που κληρονομεί ιδιότητες και λειτουργίες ονομάζεται υποκλάση (σε άλλες γλώσσες χρησιμοποιούνται οι όροι "βασική κλάση/παράγωγη κλάση" ή και "γονική κλάση/κλάση απόγονος".

Στη γλώσσα Java δεν επιτρέπεται η πολλαπλή κληρονομικότητα (δηλαδή μια κλάση δεν μπορεί να έχει περισσότερες από μία υπερκλάσεις), ενώ μπορεί να υπάρχει κληρονομικότητα και σε βάθος μεγαλύτερο του 1. Δηλαδή μια κλάση μπορεί να κληρονομεί μια κλάση, η οποία κληρονομεί από κάποια άλλη, κ.ο.κ. Ωστόσο, για λόγους συντηρησιμότητας, δεν συνιστάται η χρήση ιεραρχιών κληρονομικότητας μεγάλου βάθους.

Σημειώνεται ότι στην Java οι κατασκευαστές δεν κληρονομούνται. Κατά συνέπεια, αν θέλουμε να δημιουργούνται αντικείμενα της κλάσης AdministrativeEmployee παρέχοντας όνομα, μισθό, ώρες υπερωρίας και υπερωριακή αμοιβή, θα πρέπει να εφοδιάσουμε την υποκλάση με κατάλληλο κατασκευαστή.

public AdministrativeEmployee(String text, double amount, int hours, double wage)

και εντός του σώματος του κατασκευαστή μπορούμε να αναθέσουμε αρχικές τιμές στις αντίστοιχες ιδιότητες:

```
public AdministrativeEmployee(String text, double amount, int hours, double wage) {
            name = text;
            salary = amount;
            overtimeHours = hours;
            overtimeWage = wage;
      }
```

ωστόσο, μια ιδιότητα που έχει δηλωθεί με ιδιωτική ορατότητα (private), αν και κληρονομείται από την υποκλάση, δεν επιτρέπεται η πρόσβαση (ανάγνωση ή αλλαγή τιμής). Για να επιτραπεί η πρόσβαση σε ιδιότητες στις υποκλάσεις το προσδιοριστικό ορατότητας θα πρέπει να μεταβληθεί σε protected

(προστατευμένη) που υποδηλώνει ότι μια ιδιότητα δεν είναι προσπελάσιμη από αντικείμενα άλλων κλάσεων, επιτρέπεται όμως η πρόσβαση από αντικείμενα των υποκλάσεων.

Μια εναλλακτική ιδιότητα θα ήταν η προσπέλαση των ιδιοτήτων της υπερκλάσης μέσω κατάλληλων δημόσια διαθέσιμων μεθόδων (get() και set()) που θα μπορούσε να περιλαμβάνει για το σκοπό αυτό η υπερκλάση.

Μια τρίτη δυνατότητα είναι η κλήση του κατασκευαστή της υπερκλάσης ώστε η ανάθεση των τιμών των δύο πρώτων παραμέτρων στις ιδιότητας name και salary να πραγματοποιηθεί από τον κώδικα του κατασκευαστή της υπερκλάσης. Για την κλήση του κατασκευαστή χρησιμοποιείται η δεσμευμένη λέξη super() μεταβιβάζοντας τις αντίστοιχες παραμέτρους. Για τις υπόλοιπες δύο ιδιότητες που δηλώνονται στην υποκλάση η ανάθεση τιμών γίνεται κατά τα γνωστά:

```
public AdministrativeEmployee(String text, double amount, int hours, double wage) {
        super(text, amount);
        overtimeHours = hours;
        overtimeWage = wage;
}
```

Αν δεν προστεθεί οποιαδήποτε άλλη μέθοδος, η υποκλάση διατηρεί τις μεθόδους της υπερκλάσης (δηλαδή είναι απολύτως επιτρεπτό να αποστείλουμε το μήνυμα printStatement() σε αντικείμενο τύπου AdministrativeEmployee και ως αποτέλεσμα θα εκτυπωθεί το όνομα και ο μισθός).

Αν όμως στόχος είναι, η εκτύπωση των μηνιαίων αποδοχών να πραγματοποιείται με διαφορετικό τρόπο για τα αντικείμενα της υποκλάσης (κάτι που είναι το συνηθέστερο δεδομένου ότι προφανώς θέλουμε να εκτυπώνονται και οι αποδοχές λόγω υπερωριακής απασχόλησης) τότε θα πρέπει να **επικαλύψουμε** ή να **επανα-ορίσουμε** τη μέθοδο printStatement() για την υποκλάση, δηλαδή να ορίσουμε τη μέθοδο με την ίδια υπογραφή όπως στην υπερκλάση αλλά με διαφορετική υλοποίηση:

```
public void printStatement() {
      super.printStatement();
      System.out.println("and extra income: " + (overtimeHours * overtimeWage));
}
```

Η εντολή super.printStatement() έχει ως αποτέλεσμα την κλήση της μεθόδου printStatement() της υπερκλάσης. (θα ήταν αδύνατο να κληθεί η printStatement() της κλάσης AdministrativeEmployee δεδομένου ότι σε εκείνο το σημείο ορίζεται η ίδια η μέθοδος printStatement()). Χρησιμοποιώντας τη δεσμευμένη λέξη super και το όνομα μιας μεθόδου της υπερκλάσης μπορούμε να καλέσουμε οποιαδήποτε μέθοδο της υπερκλάσης.

Πλέον, αν κληθεί η μέθοδος printStatement() επί ενός αντικειμένου AdministrativeEmployee θα κληθεί η δική του μέθοδος και όχι αυτή που κληρονομείται από την υπερκλάση.

Αλέξανδρος Χατζηγεωργίου, 2012

Αρχή της Υποκατάστασης

Σύμφωνα με την αρχή της υποκατάστασης (substitution principle), οπουδήποτε χρησιμοποιείται σε ένα πρόγραμμα-πελάτη ένα αντικείμενο μιας υπερκλάσης, θα πρέπει να είναι απολύτως επιτρεπτό (και να μην δημιουργούνται προβλήματα εκτέλεσης ή εσφαλμένης συμπεριφοράς του προγράμματος) να χρησιμοποιούνται στη θέση του και αντικείμενα υποκλάσεων. Για παράδειγμα, αν σε μια κλάση χρησιμοποιούνται αντικείμενα της κλάσης Employee θα πρέπει να είναι απολύτως επιτρεπτό στη θέση των αντικειμένων Employee να χρησιμοποιηθούν αντικείμενα AdministrativeEmployee.

Ο λόγος είναι ότι μια σχέση κληρονομικότητας αποτελεί μια σχέση τύπου "είναι" (is-a). Με άλλα λόγια, τα αντικείμενα των υποκλάσεων "είναι" και στιγμιότυπα των υπερκλάσεών τους. Για παράδειγμα, ένας Διοικητικός Υπάλληλος έχει προφανώς τα ειδικά χαρακτηριστικά ενός Διοικητικού Υπαλλήλου, δεν παύει όμως να είναι και ένας "Υπάλληλος" της γενικής κατηγορίας, περιλαμβάνοντας τις ιδιότητες και λειτουργίες ενός υπαλλήλου. Αν παραβιάζεται εννοιολογικά η αρχή αυτή για τη σημασία της σχέσης κληρονομικότητας, τότε η χρήση της κληρονομικότητας είναι εσφαλμένη. Με άλλα λόγια ένας Υπάλληλος δεν μπορεί κληρονομεί από μια κλάση "Επιχείρηση" δεδομένου ότι ένας Υπάλληλος δεν "είναι" και επιχείρηση.

Στα πλαίσια της αρχής της υποκατάστασης σε μια αναφορά προς αντικείμενα της υπερκλάσης, π.χ. μια αναφορά Employee:

```
Employee ref;
```

που μπορούμε φυσικά να αναθέσουμε τις διευθύνσεις αντικειμένων τύπου Employee, δηλαδή:

```
Employee ref = new Employee("John", 850);
```

είναι απολύτως επιτρεπτό να αναθέσουμε και τις διευθύνσεις αντικειμένων τύπου AdministrativeEmployee:

```
Employee ref = new AdministrativeEmployee("John", 850, 10, 15);
```

Το ερώτημα που τίθεται είναι γιατί μπορεί να είναι χρήσιμη αυτή η δυνατότητα; Για να γίνει αντιληπτό, ας θεωρήσουμε ότι σε μια κλάση που χρησιμοποιεί την ιεραρχία Υπαλλήλων-Διοικητικών Υπαλλήλων (κλάση-πελάτης), έστω τη Main, διατηρούμε μια λίστα όλων των υπαλλήλων. Κανονικά θα έπρεπε να έχουμε μια ξεχωριστή λίστα για κάθε κατηγορία υπαλλήλων. Κάθε εργασία που θα έπρεπε να εκτελείται επί όλων των υπαλλήλων (π.χ. εκτύπωση της μισθοδοσίας τους καλώντας τη μέθοδο printStatement()) θα έπρεπε να πραγματοποιείται "σαρώνοντας" όλες τις δομές δεδομένων για κάθε κατηγορία υπαλλήλου. Σε ένα πραγματικό σύστημα οι εργασίες επί των υπαλλήλων θα ήταν πολλές και κατά συνέπεια θα έπρεπε σε όλα τα αντίστοιχα σημεία του κώδικα να γίνονται ενέργειες για όλες τις δομές.

Το πραγματικό όμως πρόβλημα σχετίζεται με τη συντήρηση του λογισμικού: Αν προστεθούν νέες υποκατηγορίες υπαλλήλων θα έπρεπε να προσθέτουμε νέες δομές δεδομένων για να φιλοξενήσουν τα αντικείμενα της κάθε νέας υποκατηγορίας.

Η λύση στο πρόβλημα είναι η χρήση μιας δομής δεδομένων τύπου Employee η οποία λόγω της αρχής της υποκατάστασης θα μπορεί να δεχθεί αντικείμενα τύπου Employee αλλά και αντικείμενα οποιαδήποτε υποκλάσης της Employee. Η μέθοδος main για παράδειγμα θα μπορούσε να είναι:

```
public static void main(String[] args) {

            ArrayList<Employee> employees = new ArrayList<Employee>();

            employees.add(new Employee("John", 900));
            employees.add(new AdminEmployee("Bob", 800, 15, 10));
            employees.add(new AdminEmployee("Nick", 700, 10, 12));

      }
```

Πολυμορφισμός

Αν στη συνέχεια επιθυμούμε την εκτέλεση της μεθόδου printStatement() σε κάθε είδος υπαλλήλου που περιλαμβάνεται στη δομή employees μπορούμε στη main να προσθέσουμε το κάτωθι τμήμα κώδικα που "σαρώνει" τη δομή των υπαλλήλων και αποστέλλει σε κάθε αντικείμενο της δομής το ίδιο μήνυμα printStatement():

```
            for(int i=0; i<employees.size(); i++)
                  employees.get(i).printStatement();  //πολυμορφική κλήση
```

Η κλήση της μεθόδου printStatement() στο ανωτέρω for loop παρουσιάζει ιδιαίτερο ενδιαφέρον στα πλαίσια του αντικειμενοστρεφούς προγραμματισμού. Η Java δεν γνωρίζει κατά τη διάρκεια της μεταγλώττισης (at compile time) τι αντικείμενο περιλαμβάνεται σε κάθε θέση της δομής employee. Κατά συνέπεια δεν γνωρίζει και τι αντικείμενο επιστρέφεται από την εκτέλεση της εντολής employees.get(i) σε κάθε επανάληψη (Υπάλληλος ή Διοικητικός Υπάλληλος). Για το λόγο αυτό δεν μπορεί να "αποφασίσει" κατά τη διάρκεια της μεταγλώττισης ποια μέθοδος printStatement() θα πρέπει να κληθεί (αυτή της κλάσης Employee ή αυτή της κλάσης AdminEmployee).

Για την αντιμετώπιση του ζητήματος η Java "καθυστερεί" την απόφαση αυτή μέχρι την εκτέλεση του προγράμματος όπου είναι πλέον γνωστό τι αντικείμενο βρίσκεται σε κάθε θέση της δομής employees. Αν η Java διαπιστώσει ότι πρόκειται για αντικείμενο τύπου Employee καλείται η μέθοδος printStatement() της κλάσης Employee. Αν η Java διαπιστώσει ότι πρόκειται για αντικείμενο τύπου AdminEmployee καλείται η μέθοδος printStatement() της κλάσης AdminEmployee (που έχει διαφορετική υλοποίηση σύμφωνα με τα προηγούμενα). Η δυνατότητα αυτή της Java (και άλλων αντικειμενοστρεφών γλωσσών) ονομάζεται "δυναμική διασύνδεση" ή "καθυστερημένη διασύνδεση" μεθόδου και αντίστοιχης κλάσης (dynamic ή late binding).

Το όφελος που προκύπτει είναι εξαιρετικά σημαντικό: ουσιαστικά στον κώδικα του for αποστέλλουμε το ίδιο ακριβώς μήνυμα σε ένα πλήθος αντικειμένων (το μήνυμα printStatement()) και κάθε αντικείμενο-παραλήπτης ανταποκρίνεται με διαφορετικό τρόπο, αυτόν που αντιστοιχεί στην κλάση του αντικειμένου. (ένα μήνυμα -> πολλές μορφές απόκρισης). **Η δυνατότητα αυτή ονομάζεται πολυμορφισμός** και η αντίστοιχη κλήση της μεθόδου "πολυμορφική κλήση". Χάρη στον πολυμορφισμό, αν μελλοντικά

προστεθούν νέες υποκατηγορίες υπαλλήλων (π.χ. Τεχνικοί Υπάλληλοι, Εξωτερικοί Συνεργάτες, Πωλητές κλπ με ξεχωριστό τρόπο μισθοδοσίας και διαφορετική υλοποίηση της printStatement()), τότε προφανώς θα πρέπει να προστεθούν στο σύστημα οι αντίστοιχες κλάσεις, **αλλά ο κώδικας του ανωτέρου τμήματος (ο βρόχος επανάληψης for) δεν θα πρέπει να τροποποιηθεί καθόλου.** Δηλαδή ένα τμήμα κώδικα μπορεί να επεκτείνει τη λειτουργικότητά του (να χειριστεί και άλλες κατηγορίες υπαλλήλων) χωρίς να χρειάζεται η τροποποίησή του (Αρχή της Ανοικτής-Κλειστής Σχεδίασης).

Φυσικά στο ανωτέρω παράδειγμα το τμήμα κώδικα που επωφελείται από τον πολυμορφισμό είναι μικρό (ο κώδικας του for) αλλά σε ένα πραγματικό σύστημα την ίδια "ανεξαρτησία" μπορούν να έχουν πλήθος από κλάσεις ακόμα και ολόκληρα πληροφοριακά συστήματα.

Για να γίνει απολύτως κατανοητό ότι η Java δεν είναι σε θέση να γνωρίζει τι ακριβώς υπάρχει ή θα τοποθετηθεί σε κάθε θέση της δομής employees (και επομένως δεν μπορεί να ξέρει ποια μέθοδο πρέπει να καλέσει για κάθε αντικείμενο που βρίσκεται σε μια θέση της δομής) το πρόγραμμα θα μπορούσε να ρωτά τον χρήστη τι αντικείμενο επιθυμεί να εισάγει κάθε φορά:

```
public static void main(String[] args) {

            ArrayList<Employee> employees = new ArrayList<Employee>();

            Scanner scan = new Scanner(System.in);
            boolean more = true;

            while(more) {
              System.out.println("What kind of employee do you want to add?
                                  (1: Employee, 2 AdminEmployee)");
              int answer  = scan.nextInt();
              scan.nextLine();

              System.out.println("Enter name: ");
              String name = scan.nextLine();
              System.out.println("Enter salary: ");
              double salary = scan.nextDouble();

              if(answer == 1) {
                  employees.add(new Employee(name, salary));
              }
              else {
                  System.out.println("Enter overtime hours: ");
                  int hours = scan.nextInt();
                  System.out.println("Enter overtime wage: ");
                  double wage = scan.nextDouble();
                  employees.add(new AdminEmployee(name, salary, hours, wage));
              }

              scan.nextLine();
              //Ερώτημα για το αν θα επαναληφθεί εισαγωγή υπαλλήλου
              System.out.println("More employees? (y/n)");
              String moreData = scan.nextLine();
              if(moreData.equals("n"))
                  more = false;
            }
```

```
        for(int i=0; i<employees.size(); i++)
              employees.get(i).printStatement();  //πολυμορφική κλήση

    }
```

Στο ανωτέρω παράδειγμα είναι πλέον σαφές ότι κατά τη διάρκεια της μεταγλώττισης του τμήματος κώδικα που αφορά στην αποστολή του μηνύματος printStatement() σε όλους ανεξαιρέτως τους υπαλλήλους (βρόχος for στο τέλος) η Java δεν μπορεί να γνωρίζει τι αντικείμενο υπάρχει σε κάθε θέση της δομής αφού αυτό καθορίζεται από τον χρήστη κατά τη διάρκεια της εκτέλεσης.