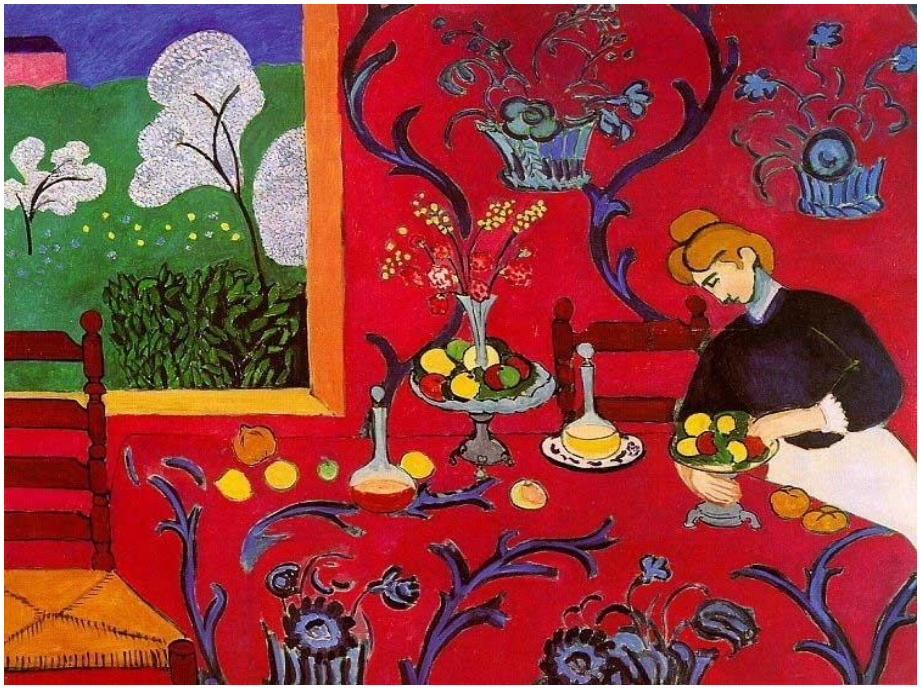
Αντικειμενοστρέφεια και Εξαιρέσεις (στη Java)



Henri Matisse, *Harmony in Red,* 1908 και Vincent van Gogh, *Irises*, 1889

Κωστής Σαγώνας <kostis@cs.ntua.gr>
Νίκος Παπασπύρου <nickie@softlab.ntua.gr>

Ορισμοί αντικειμενοστρέφειας

- Ποιοι είναι οι ορισμοί των παρακάτω;
  - Αντικειμενοστρεφής γλώσσα προγραμματισμού
  - Αντικειμενοστρεφής προγραμματισμός
- Αλλά από την άλλη μεριά, για ποιο λόγο να τους ξέρουμε;

Κάποιες γενικές παρατηρήσεις:

- Ο αντικειμενοστρεφής προγραμματισμός δεν είναι απλά προγραμματισμός σε μια αντικειμενοστρεφή γλώσσα
- Οι αντικειμενοστρεφείς γλώσσες δεν είναι όλες σαν τη Java

Χαρακτηριστικά αντικειμενοστρεφών γλωσσών

- Κλάσεις
- Πρωτότυπα
- Κληρονομικότητα (inheritance)
- Ενθυλάκωση (encapsulation)
- Πολυμορφισμός

Πολυμορφισμός χωρίς αντικειμενοστρέφεια

- Σε προηγούμενη διάλεξη είδαμε μια διαπροσωπεία **`Worklist`** να υλοποιείται από μια **`Stack`**, **`Queue`**, κ.λπ.

- Υπάρχει ένα κοινό τρικ υποστήριξης της συγκεκριμένης δυνατότητας πολυμορφισμού σε μη αντικειμενοστρεφείς γλώσσες
- Κάθε εγγραφή (record) αρχίζει με ένα στοιχείο κάποιας απαρίθμησης, που προσδιορίζει το είδος της **`Worklist`**

Μη αντικειμενοστρεφής `Worklist`

```
public class Worklist {
  public static final int STACK = 0;
  public static final int QUEUE = 1;
  public static final int PRIORITYQUEUE = 2;
  public int type; // one of the above Worklist types
  public Node front; // front Node in the list
  public Node rear;  // unused when type == STACK
  public int length; // unused when type == STACK
}
```

- Το πεδίο **`type`** προσδιορίζει το είδος της **`Worklist`**
- Η ερμηνεία των άλλων πεδίων εξαρτάται από το **`type`**
- Οι μέθοδοι που διαχειρίζονται τις **`Worklist`** εγγραφές έχουν κάποια διακλάδωση με βάση την τιμή του **`type`**…

Διακλάδωση με βάση τον τύπο

```
private static void add(Worklist w, String data) {
  if (w.type == Worklist.STACK) {
    Node n = new Node();
    n.data = data;
    n.link = w.front;
    w.front = n;
  }
  else if (w.type == Worklist.QUEUE) {
    η υλοποίηση της add για ουρές
  }
  else if (w.type == Worklist.PRIORITYQUEUE) {
    η υλοποίηση της add για ουρές με προτεραιότητα
  }
}
```

Κάθε μέθοδος που χρησιμοποιεί μια **`Worklist`** πρέπει να έχει κάποια αντίστοιχη διακλάδωση τύπου

Μειονεκτήματα

- Η επανάληψη του κώδικα που υλοποιεί τη διακλάδωση είναι βαρετή και επιρρεπής σε προγραμματιστικά λάθη
- Ανάλογα με τη γλώσσα, μπορεί να μην υπάρχει τρόπος αποφυγής της σπατάλης χώρου εάν διαφορετικά είδη εγγραφών απαιτούν διαφορετικά πεδία
- Κάποιες τυπικές προγραμματιστικές λειτουργίες, όπως για παράδειγμα η πρόσθεση κάποιου νέου είδους/τύπου αντικειμένου, δυσκολεύουν αρκετά

Πλεονεκτήματα αντικειμενοστρέφειας

- Όταν καλούμε μια μέθοδο κάποιας διαπροσωπείας, η υλοποίηση της γλώσσας αυτόματα αποστέλλει (dispatches) την εκτέλεση στο σωστό κώδικα της μεθόδου για το συγκεκριμένο αντικείμενο
- Οι διαφορετικές υλοποιήσεις μιας διαπροσωπείας δεν είναι απαραίτητο να μοιράζονται πεδία
- Η πρόσθεση μιας κλάσης που υλοποιεί μια διαπροσωπεία είναι πολύ εύκολη διότι δεν απαιτεί την τροποποίηση κάποιου υπάρχοντα κώδικα

Κάποιες σκέψεις

- Ο αντικειμενοστρεφής προγραμματισμός δεν είναι το ίδιο πράγμα με τον προγραμματισμό σε μια αντικειμενοστρεφή γλώσσα
  - Μπορεί να γίνει σε μια μη αντικειμενοστρεφή γλώσσα
  - Μπορεί να μη γίνει σε μια αντικειμενοστρεφή γλώσσα
- Συνήθως, οι αντικειμενοστρεφείς γλώσσες συνδυάζονται με τα αντικειμενοστρεφή στυλ προγραμματισμού
  - *Συνήθως* ένα πρόγραμμα σε ML που έχει γραφεί με χρήση αντικειμενοστρεφούς στυλ προγραμματισμού δεν είναι ότι καλύτερο υπάρχει από πλευράς σχεδιασμού
  - Επίσης *συνήθως* ένα πρόγραμμα σε Java που έχει γραφεί με χρήση του αντικειμενοστρεφούς στυλ προγραμματισμού έχει καλύτερο σχεδιασμό από ότι ένα που έχει γραφεί χωρίς να ακολουθηθεί το συγκεκριμένο στυλ

Χαρακτηριστικά αντικειμενοστρέφειας

Κλάσεις

- Οι περισσότερες αντικειμενοστρεφείς γλώσσες έχουν κάποιον τρόπο ορισμού κλάσεων
- Οι κλάσεις εξυπηρετούν διάφορους σκοπούς:
  - Ομαδοποιούν πεδία και μεθόδους
  - Στιγμιοτυποποιούνται: το πρόγραμμα μπορεί να δημιουργήσει όσα αντικείμενα των κλάσεων χρειάζεται για την εκτέλεσή του
  - Αποτελούν μονάδες κληρονομικότητας: μια παραγόμενη κλάση κληρονομεί από όλες τις βασικές κλάσεις της
  - Αποτελούν τύπους: τα αντικείμενα (ή οι αναφορές τους) έχουν κάποιο όνομα κλάσης ως στατικό τύπο
  - Στεγάζουν στατικά πεδία και μεθόδους και αυτή η στέγαση γίνεται ανά κλάση, όχι ανά στιγμιότυπο
  - Λειτουργούν ως χώροι ονομάτων και ελέγχουν την ορατότητα του περιεχομένου μιας κλάσης εκτός της κλάσης

Αντικειμενοστρέφεια χωρίς κλάσεις

- Σε μια γλώσσα με κλάσεις δημιουργούμε αντικείμενα με το να φτιάχνουμε στιγμιότυπα των κλάσεων

- Σε μια γλώσσα χωρίς κλάσεις δημιουργούμε αντικείμενα
  - είτε από το μηδέν με το να ορίσουμε τα πεδία τους και να γράψουμε τον κώδικα των μεθόδων τους
  - ή με κλωνοποίηση ενός υπάρχοντος πρωτοτύπου αντικειμένου και τροποποίηση κάποιων από τα μέρη του

```
x = new Stack();
```

Με κλάσεις:
δημιουργία
στιγμιότυπου

```
x = {
  private Node top = null;
  public boolean hasMore() {
    return (top != null);
  }
  public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
  }
  …
}
```

Χωρίς κλάσεις:
δημιουργία
αντικειμένου
από το μηδέν

```
x = y.clone();
x.top = null;
```

Χωρίς κλάσεις:
κλωνοποίηση
πρωτοτύπου

Πρωτότυπα

- Ένα **πρωτότυπο (prototype)** είναι ένα αντικείμενο το οποίο μπορεί να αντιγραφεί ώστε να δημιουργήσει παρόμοια αντικείμενα
- Κατά τη δημιουργία αντιγράφων, το πρόγραμμα μπορεί να τροποποιήσει τις τιμές κάποιων πεδίων, να προσθέσει ή να αφαιρέσει πεδία και μεθόδους
- Οι γλώσσες που βασίζονται σε πρωτότυπα (όπως η Self) χρησιμοποιούν αυτήν την ιδέα αντί για κλάσεις

Χωρίς κλάσεις αλλά με πρωτότυπα

- Η εύκολη δημιουργία στιγμιότυπων είναι μία από τις βασικές χρησιμότητες των κλάσεων
- Κάποια άλλα χαρακτηριστικά που πρέπει να αποχωριστούν οι γλώσσες που βασίζονται σε πρωτότυπα:
  - Τις κλάσεις ως τύπους: οι περισσότερες γλώσσες που βασίζονται σε πρωτότυπα έχουν δυναμικούς τύπους
  - Την κληρονομικότητα: οι γλώσσες που βασίζονται σε πρωτότυπα χρησιμοποιούν μια δυναμική τεχνική η οποία είναι σχετική με την κληρονομικότητα και λέγεται **αντιπροσωπεία (delegation)**

Κληρονομικότητα

- Σε επιφανειακό επίπεδο, η έννοια της κληρονομικότητας είναι αρκετά εύκολη να κατανοηθεί
  - Η επέκταση κλάσεων δημιουργεί μια σχέση μεταξύ δύο κλάσεων: μια σχέση μεταξύ μιας παραγόμενης και μιας βασικής κλάσης
  - Η παραγόμενη κλάση κληρονομεί πεδία και μεθόδους από τη βασική κλάση
- Αλλά το τι κληρονομείται από τη βασική κλάση (ή κλάσεις) καθορίζεται από τη γλώσσα
- Θα δούμε ότι διαφορετικές γλώσσες έχουν διαφορετικές προσεγγίσεις πάνω σε θέματα κληρονομικότητας

Ερωτήσεις κληρονομικότητας

- Επιτρέπονται περισσότερες από μία βασικές κλάσεις;
  - Απλή κληρονομικότητα: Smalltalk, Java
  - Πολλαπλή κληρονομικότητα : C++, CLOS, Eiffel
- Οι υποκλάσεις κληρονομούν τα πάντα;
  - Στη Java: η παραγόμενη κλάση κληρονομεί όλες τις μεθόδους και τα πεδία
  - Στη Sather: η παραγόμενη κλάση μπορεί να μετονομάσει τις κληρονομημένες μεθόδους της (κάτι που είναι χρήσιμο όταν υπάρχει πολλαπλή κληρονομικότητα), ή απλώς να τις ξε-ορίσει
- Υπάρχει κάποια καθολική βασική κλάση;
  - Μια κλάση από την οποία όλες οι κλάσεις κληρονομούν: η κλάση `Object` της Java
  - Δεν υπάρχει κάποια τέτοια κλάση στη C++

Ερωτήσεις κληρονομικότητας

- Κληρονομούνται οι προδιαγραφές;
  - Ως υποχρεώσεις των μεθόδων για υλοποίηση, όπως στη Java
  - Ως επιπλέον προδιαγραφές: invariants, όπως στην Eiffel
- Κληρονομούνται οι τύποι;
  - Στη Java κληρονομούνται όλοι οι τύποι της βασικής κλάσης
- Υποστηρίζεται υπερκάλυψη, απόκρυψη, κ.λπ.;
  - Τι συμβαίνει στη Java, στο περίπου (χωρίς πολλές λεπτομέρειες):
    - Οι κατασκευαστές μπορούν να προσπελάσουν τους κατασκευαστές των βασικών τους κλάσεων (Αυτό γίνεται με χρήση του **`super`** που καλεί τον κατασκευαστή της άμεσης υπερκλάσης.)
    - Μια νέα μέθοδος με το ίδιο όνομα και τύπο με την κληρονομημένη μέθοδο την υπερκαλύπτει (Η υπερκαλυμμένη μέθοδος μπορεί να κληθεί με χρήση του προσδιοριστή **`super`**.)
    - Ένα νέο πεδίο ή στατική μέθοδος αποκρύπτει τις κληρονομημένες, οι οποίες όμως μπορούν να προσπελαστούν με χρήση του **`super`** ή με στατικούς τύπους της βασικής κλάσης

Ενθυλάκωση (encapsulation)

- Απαντάται σχεδόν σε όλες τις μοντέρνες γλώσσες προγραμματισμού, όχι μόνο στις αντικειμενοστρεφείς
- Τα μέρη του προγράμματος που ενθυλακώνονται:
  - Παρουσιάζουν μια ελεγχόμενη διαπροσωπεία στους χρήστες τους
  - Αποκρύπτουν όλα τα άλλα (ιδιωτικά) μέρη τους
- Στις αντικειμενοστρεφείς γλώσσες, τα αντικείμενα ενθυλακώνονται
- (Διαφορετικές γλώσσες υλοποιούν την ενθυλάκωση με διαφορετικούς τρόπους)

Ορατότητα των πεδίων και των μεθόδων

- Στη Java υπάρχουν τέσσερα επίπεδα ορατότητας
  - **`private`**: ορατά μόνο μέσα στην κλάση
  - (default): ορατά μόνο εντός του πακέτου (package)
  - **`protected`**: ορατά μόνο στο ίδιο πακέτο και στις παραγόμενες κλάσεις
  - **`public`**: παντού
- Κάποιες αντικειμενοστρεφείς γλώσσες (Smalltalk, Self) έχουν λιγότερα επίπεδα ελέγχου ορατότητας: όλα κοινά (**`public`**)
- Άλλες έχουν περισσότερα: στην Eiffel, πεδία και μέθοδοι μπορεί να γίνουν ορατά μόνο σε ένα συγκεκριμένο σύνολο από κλάσεις-πελάτες

Πολυμορφισμός

- Συναντιέται σε πολλές γλώσσες, όχι μόνο στις αντικειμενοστρεφείς
- Κάποιες από τις βασικότερες εκφράσεις του πολυμορφισμού στις αντικειμενοστρεφείς γλώσσες:
  - Όταν διαφορετικές κλάσεις έχουν μεθόδους του ίδιου ονόματος και τύπου
    - Π.χ. μια κλάση στοίβας και μια κλάση ουράς μπορούν και οι δύο να έχουν μια μέθοδο ονόματι **`add`**
  - Όταν η γλώσσα επιτρέπει μια κλήση μεθόδου σε στιγμές που η κλάση του αντικειμένου δεν είναι γνωστή στατικά

Παράδειγμα σε Java

```
public static void flashoff(Drawable d, int k) {
  for (int i = 0; i < k; i++) {
    d.show(0,0);
    d.hide();
  }
}
```

- Εδώ, η **`Drawable`** είναι μια διαπροσωπεία
- Η κλάση του αντικειμένου στην οποία αναφέρεται η αναφορά **`d`** δεν είναι γνωστή στο χρόνο μεταγλώττισης

Δυναμική αποστολή (dynamic dispatch)

- Στη Java, ο στατικός τύπος μιας αναφοράς μπορεί να είναι μια υπερκλάση ή μια διαπροσωπεία κάποιας κλάσης
- Στο χρόνο εκτέλεσης, το σύστημα υλοποίησης της γλώσσας θα πρέπει κάπως να βρει και να καλέσει τη σωστή μέθοδο της πραγματικής κλάσης
- Αυτό αναφέρεται ως **δυναμική αποστολή (dynamic dispatch)**: η κρυφή και έμμεση διακλάδωση πάνω στην κλάση κατά την κλήση των μεθόδων
  - Η δυναμική αποστολή είναι προαιρετική στη C++
  - Χρησιμοποιείται πάντα στη Java και στις περισσότερες άλλες αντικειμενοστρεφείς γλώσσες

Υλοποιήσεις και τύποι

- Στη Java υπάρχουν δύο μηχανισμοί:
  - Μια κλάση κληρονομεί τόσο τους τύπους όσο και την υλοποίηση της βασικής της κλάσης
  - Μια κλάση παίρνει πρόσθετους τύπους (αλλά όχι αυτόματα κάποια υλοποίηση) με την υλοποίηση διαπροσωπειών
- Το παραπάνω μερικώς διαχωρίζει την κληρονομικότητα της υλοποίησης από την κληρονομικότητα του τύπου
- Άλλες αντικειμενοστρεφείς γλώσσες διαφέρουν στο κατά πόσο ξεχωρίζουν τα δύο παραπάνω

Υλοποιήσεις και τύποι

- Στη C++ δεν υπάρχει κάποιος αντίστοιχος διαχωρισμός
  - Υπάρχει ένας μόνο μηχανισμός για την κληρονομικότητα
  - Για την κληρονομιά τύπου μόνο, μπορούμε να χρησιμοποιήσουμε μια αφηρημένη βασική κλάση χωρίς κάποια υλοποίηση
- Από την άλλη μεριά στη Sather ο διαχωρισμός υλοποιήσεων και τύπων είναι πλήρης:
  - Μια κλάση μπορεί να δηλώσει ότι **περιλαμβάνει** κάποια άλλη κλάση, οπότε κληρονομεί την υλοποίηση αλλά όχι τον τύπο
  - Μια κλάση μπορεί να δηλώσει ότι είναι μια υποκλάση μιας αφηρημένης κλάσης, οπότε κληρονομεί τον τύπο αλλά όχι την υλοποίηση (όπως συμβαίνει με τις διαπροσωπείες στη Java)

Χρήση δυναμικού συστήματος τύπων

- Κάποιες αντικειμενοστρεφείς γλώσσες χρησιμοποιούν δυναμικό σύστημα τύπων: π.χ. η Smalltalk και η Self
- Ένα αντικείμενο μπορεί να είναι ή να μην είναι σε θέση να απαντήσει σε ένα συγκεκριμένο μήνυμα – και αυτό μπορεί να μην μπορεί να ελεγχθεί κατά το χρόνο μετάφρασης
- Αυτό δίνει απόλυτη ελευθερία: το πρόγραμμα μπορεί να δοκιμάσει τη χρήση οποιασδήποτε μεθόδου σε οποιοδήποτε αντικείμενο
- (Ο πολυμορφισμός δεν έχει σχέση με τη συγκεκριμένη ελευθερία)

Συμπερασματικά

- Σήμερα, ακολουθήσαμε μια κοσμοπολίτικη προσέγγιση:
  - Ο αντικειμενοστρεφής προγραμματισμός δεν είναι το ίδιο πράγμα με τον προγραμματισμό με μια αντικειμενοστρεφή γλώσσα
  - Οι αντικειμενοστρεφείς γλώσσες δεν είναι όλες σαν τη Java
- Το τι είναι αντικειμενοστρεφές στυλ προγραμματισμού

  δεν είναι μονοσήμαντα ορισμένο:

  υπάρχει αρκετή διαφωνία ως προς το ακριβές σύνολο των χαρακτηριστικών της αντικειμενοστρέφειας και τα χαρακτηριστικά αυτά συνεχώς εξελίσσονται
- Δείξτε σκεπτικισμό στους ορισμούς!

Εξαιρέσεις (στη Java)

Εξαιρέσεις στη Java

```
public class Test {
  public static void main(String[] args) {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
}
```

```
> javac Test.java
> java Test 4 2
2
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at Test.main(Test.java:3)
> java Test 42 0
Exception in thread "main"
        java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:5)
```

Περιεχόμενα

- Ριπτόμενες κλάσεις (throwable classes)
- Πιάσιμο εξαιρέσεων (catching exceptions)
- Ρίψη εξαιρέσεων (throwing exceptions)
- Ελεγχόμενες εξαιρέσεις (checked exceptions)
- Χειρισμός σφαλμάτων (error handling)
- Η πρόταση **`finally`**

Κάποιες προκαθορισμένες εξαιρέσεις

Εξαίρεση της Java	Κώδικας που την εγείρει
`NullPointerException`	`String s = null;` `s.length();`

`ArithmeticException`	`int a = 42;`
	`int b = 0;`
	`int q = a/b;`

`ArrayIndexOutOfBoundsException`	`int[] a = new int[10];` `a[42];`

`ClassCastException`	`Object x =`
	`  new Integer(42);`
	`String s = (String) x;`

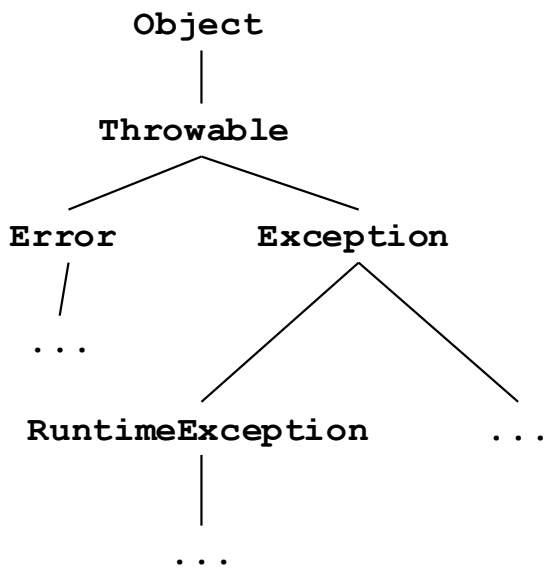
`StringIndexOutOfBoundsException`	`String s = "Hello";` `s.charAt(8);`

Μια εξαίρεση είναι ένα αντικείμενο

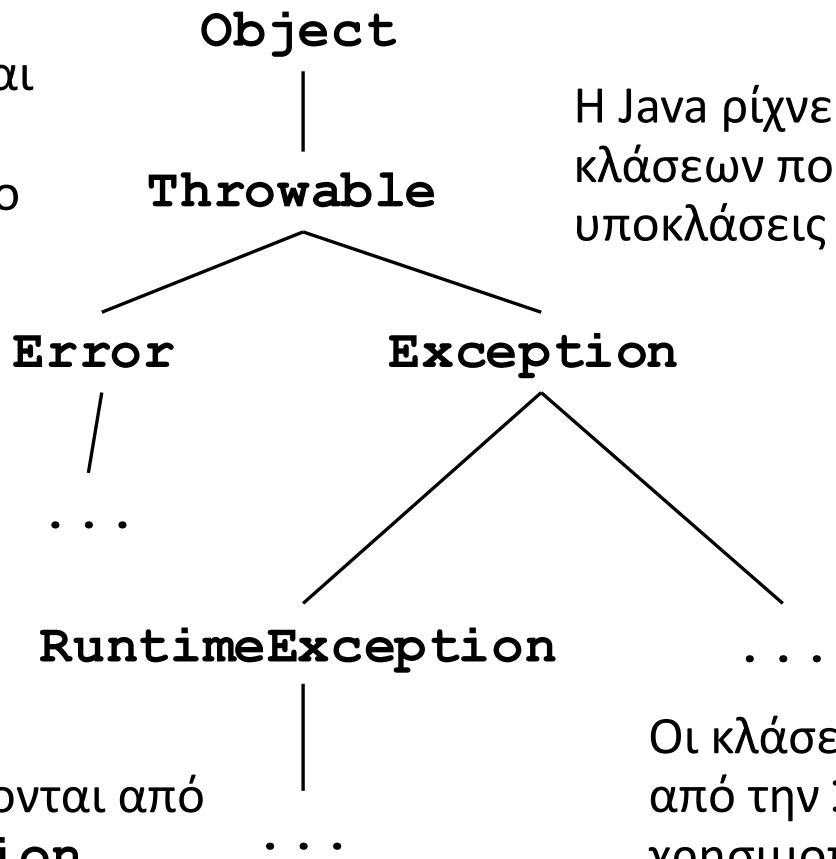
- Τα ονόματα των εξαιρέσεων είναι ονόματα κλάσεων, όπως π.χ. **`NullPointerException`**
- Οι εξαιρέσεις είναι αντικείμενα των συγκεκριμένων κλάσεων
- Στα προηγούμενα παραδείγματα, η υλοποίηση της Java δημιουργεί αυτόματα ένα αντικείμενο της συγκεκριμένης κλάσης εξαίρεσης και το **ρίχνει (throws)**
- Αν το πρόγραμμα δεν **πιάσει (catch)** αυτό το αντικείμενο, τότε η εκτέλεση του προγράμματος τερματίζεται με ένα μήνυμα λάθους

Ριπτόμενες κλάσεις

- Για να ριχθεί ως εξαίρεση, ένα αντικείμενο πρέπει να είναι κάποιας κλάσης η οποία κληρονομεί από την προκαθορισμένη κλάση **`Throwable`**
- Στο συγκεκριμένο μέρος της ιεραρχίας των κλάσεων της Java υπάρχουν τέσσερις σημαντικές προκαθορισμένες κλάσεις:
  - **`Throwable`**
  - **`Error`**
  - **`Exception`**
  - **`RuntimeException`**



Οι κλάσεις που παράγονται από την **Error** χρησιμοποιούνται για σημαντικά λάθη του συστήματος, όπως π.χ. το **OutOfMemoryError**, από τα οποία συνήθως δεν μπορούμε να ανακάμψουμε



Η Java ρίχνει αντικείμενα κλάσεων που είναι υποκλάσεις της **Throwable**

Οι κλάσεις που παράγονται από τη **RuntimeException** χρησιμοποιούνται για συνήθη λάθη του συστήματος, όπως π.χ. **ArithmeticException**

Οι κλάσεις που παράγονται από την **Exception** χρησιμοποιούνται για συνήθη λάθη τα οποία το πρόγραμμα μπορεί να θέλει να πιάσει και να ανακάμψει από αυτά

Πιάσιμο εξαιρέσεων

Η εντολή `try`

```
<try-statement>  ::=  <try-part>  <catch-part>
<try-part>  ::=  try  <compound-statement>
<catch-part>  ::=  catch  (<type>  <variable-name>)
                          <compound-statement>
```

- Η παραπάνω σύνταξη είναι απλοποιημένη… η πλήρης σύνταξη θα δοθεί αργότερα
- Το *<type>* είναι το όνομα μιας ριπτόμενης κλάσης
- Η εντολή εκτελεί το σώμα της **`try`**
- Εκτελεί το σκέλος **`catch`** μόνο εάν το σκέλος **`try`** ρίξει μια εξαίρεση του συγκεκριμένου τύπου *<type>*

Παράδειγμα

```
public class Test {
  public static void main(String[] args) {
    try {
      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i/j);
    }
    catch (ArithmeticException a) {
      System.out.println("You're dividing by zero!");
    }
  }
}
```

Ο παραπάνω κώδικας θα πιάσει και θα χειριστεί οποιαδήποτε **`ArithmeticException`**. Το σύστημα θα συμπεριφερθεί στις υπόλοιπες εξαιρέσεις σύμφωνα με τον προκαθορισμένο τρόπο για εξαιρέσεις για τις οποίες δεν υπάρχει κάποιος χειριστής.

Παράδειγμα

```
> java Test 4 2
2
> java Test 42 0
You're dividing by zero!
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
        at Test.main(Test.java:3)
```

- Ο τύπος του ορίσματος του **`catch`** επιλέγει το τι εξαίρεση θα πιαστεί από τον κώδικα:
  - Ο τύπος **`ArithmeticException`** θα πιάσει μόνο κάποια αριθμητική εξαίρεση (π.χ. διαίρεση με το μηδέν)
  - Ο τύπος **`RuntimeException`** θα πιάσει και τα δύο παραπάνω παραδείγματα λάθους χρήσης
  - Ο τύπος **`Throwable`** θα πιάσει όλες τις εξαιρέσεις

Έλεγχος ροής μετά την εντολή `try`

- Η εντολή **`try`** μπορεί να είναι κάποια από τις εντολές σε μια ακολουθία από εντολές
- Εάν δε συμβεί κάποια εξαίρεση στο σκέλος **`try`**, το σκέλος **`catch`** δεν εκτελείται
- Εάν δε συμβεί κάποια εξαίρεση στο σκέλος **`try`**, ή εάν συμβεί κάποια εξαίρεση την οποία το σκέλος **`catch`** πιάνει, η εκτέλεση θα συνεχίσει με την εντολή που είναι η αμέσως επόμενη από το σκέλος **`catch`** της εντολής **`try`**

Χειρισμός της εξαίρεσης

```
System.out.print("1, ");
try {
  String s = null;
  s.length();
}
catch (NullPointerException e) {
  System.out.print("2, ");
}
System.out.println("3");
```

Απλώς τυπώνει τη γραμμή

```
1, 2, 3
```

Ρίψη εξαίρεσης από κληθείσα μέθοδο

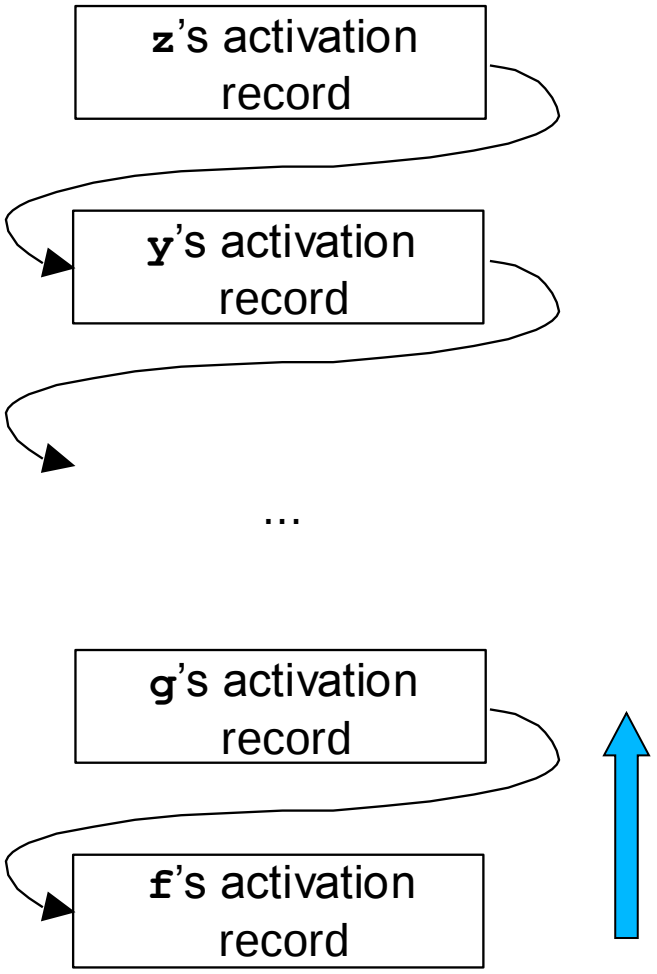
- Η εντολή **`try`** έχει την ευκαιρία να πιάσει εξαιρέσεις που πιθανώς να εγερθούν από την εκτέλεση του σκέλους **`try`**
- Αυτό περιλαμβάνει όλες τις εξαιρέσεις που ρίχνονται από μεθόδους που καλούνται (αμέσως ή εμμέσως) από το σώμα του **`try`**

Παράδειγμα

```
void f() {
   try {
      g();
   }
   catch (ArithmeticException a) {
      … // some action
   }
}
```

- Εάν η **g** ρίξει μια **ArithmeticException** και δεν την πιάσει η ίδια, η εξαίρεση θα προωθηθεί στην **f**
- Γενικά, ένα **throw** που θα ρίξει μια εξαίρεση και το **catch** που θα την πιάσει μπορεί να διαχωρίζονται από έναν απροσδιόριστο αριθμό δραστηριοποιήσεων μεθόδων

- Εάν η **z** ρίξει μια εξαίρεση που δεν πιάνει, η δραστηριοποίηση της **z** σταματάει…
- …τότε η **y** έχει την ευκαιρία να πιάσει την εξαίρεση… εάν δε την πιάσει, η δραστηριοποίηση της **y** επίσης σταματάει…
- … κοκ …
- … μέχρι την εγγραφή δραστηριοποίησης της πρώτης κλήσης συνάρτησης (**f**)



Ρίψεις μεγάλου μήκους

- Οι εξαιρέσεις είναι δομές ελέγχου ροής
- Ένα από τα μεγαλύτερα πλεονεκτήματα του χειρισμού εξαιρέσεων είναι η δυνατότητα για ρίψεις μεγάλου μήκους
- Όλες οι δραστηριοποιήσεις που βρίσκονται μεταξύ του **`throw`** και του **`catch`** σταματούν την εκτέλεσή τους και απομακρύνονται από τη στοίβα
- Εάν δεν υπάρχει ρίψη ή πιάσιμο εξαιρέσεων, οι δραστηριοποιήσεις δε χρειάζεται να ξέρουν τίποτε για τις εξαιρέσεις

Πολλαπλά `catch`

```
<try-statement> ::= <try-part> <catch-parts>
<try-part> ::= try <compound-statement>
<catch-parts> ::= <catch-part> <catch-parts>
                | <catch-part>
<catch-part> ::= catch (<type> <variable-name>)
                   <compound-statement>
```

- Για να πιάσουμε περισσότερα είδη εξαιρέσεων, ένα **`catch`** μπορεί να δηλώσει κάποια πιο γενική υπερκλάση, όπως π.χ. **`RuntimeException`**
- Αλλά συνήθως για να χειριστούμε διαφορετικά είδη εξαιρέσεων με διαφορετικό τρόπο, χρησιμοποιούμε πολλαπλά **`catch`**

Παράδειγμα

```
public static void main(String[] args) {
  try {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
  catch (ArithmeticException a) {
    System.out.println("You're dividing by zero!");
  }
  catch (ArrayIndexOutOfBoundsException a) {
    System.out.println("Requires two parameters.");
  }
}
```

Ο κώδικας θα πιάσει και θα χειριστεί τόσο **ArithmeticException** όσο και **ArrayIndexOutOfBoundsException**

Επικαλυπτόμενες προτάσεις `catch`

- Εάν μια εξαίρεση από το σώμα του **`try`** ταιριάζει με περισσότερα από ένα από τα **`catch`**, μόνο το πρώτο που θα ταιριάξει εκτελείται
- Άρα προγραμματίζουμε ως εξής: γράφουμε σκέλη **`catch`** πρώτα για τις ειδικές περιπτώσεις και βάζουμε τα πιο γενικά στο τέλος

Παρατήρηση: Η Java δεν επιτρέπει απρόσιτα σκέλη **`catch`**, ή πιο γενικά την ύπαρξη απρόσιτου κώδικα

Παράδειγμα

```
public static void main(String[] args) {
  try {
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    System.out.println(i/j);
  }
  catch (ArithmeticException a) {
    System.out.println("You're dividing by zero!");
  }
  catch (ArrayIndexOutOfBoundsException a) {
    System.out.println("Requires two parameters.");
  }
  // last the superclass of all thrown exceptions
  catch (RuntimeException a) {
    System.out.println("Runtime exception.");
  }
}
```

Ρίψη εξαιρέσεων

Η εντολή `throw`

<*throw-statement*> **::= throw** <*expression*> **;**

- Οι περισσότερες εξαιρέσεις εγείρονται αυτόματα από το σύστημα υλοποίησης της γλώσσας
- Πολλές φορές όμως θέλουμε να εγείρουμε δικές μας εξαιρέσεις και τότε χρησιμοποιούμε την εντολή **`throw`**
- Η έκφραση <*expression*> είναι μια αναφορά σε ένα ριπτόμενο αντικείμενο, συνήθως ένα νέο αντικείμενο κάποιας κλάσης εξαίρεσης:

```
throw new NullPointerException();
```

Ριπτόμενες εξαιρέσεις ορισμένες από το χρήστη

```
public class OutOfGas extends Exception {
}
```

```
System.out.print("1, ");
try {
  throw new OutOfGas();
}
catch (OutOfGas e) {
  System.out.print("2, ");
}
System.out.println("3");
```

Χρήση των αντικειμένων εξαιρέσεων

- Η ριφθείσα εξαίρεση είναι διαθέσιμη στο μπλοκ του **`catch`** με τη μορφή παραμέτρου
- Μπορεί να χρησιμοποιηθεί για να περάσει πληροφορία από τον “ρίπτη” (thrower) στον “πιάνοντα” (catcher)
- Όλες οι κλάσεις που παράγονται από τη **`Throwable`** κληρονομούν μια μέθοδο **`printStackTrace`**
- Κληρονομούν επίσης ένα πεδίο τύπου **`String`** στο οποίο υπάρχει ένα λεπτομερές μήνυμα λάθους, όπως και μία μέθοδο **`getMessage`** με την οποία μπορούμε να προσπελάσουμε αυτό το μήνυμα

Παράδειγμα χρήσης

```
public class OutOfGas extends Exception {
  public OutOfGas(String details) {
    super(details);
  }
}
```

Καλεί τον κατασκευαστή της βασικής κλάσης για να αρχικοποιήσει το πεδίο που επιστρέφεται από τη **getMessage()**

```
try {
  throw new OutOfGas("You have run out of gas.");
}
catch (OutOfGas e) {
  System.out.println(e.getMessage());
}
```

Σχετικά με το `super` στους κατασκευαστές

- Η πρώτη εντολή σε έναν κατασκευαστή μπορεί να είναι μια κλήση στον κατασκευαστή της υπερκλάσης με χρήση του **`super`** (με παραμέτρους, εάν χρειάζεται)
- Η συγκεκριμένη κλήση χρησιμοποιείται για να αρχικοποιήσει τα κληρονομημένα πεδία

- Όλοι οι κατασκευαστές (εκτός φυσικά από αυτούς της κλάσης **`Object`**) αρχίζουν με μια κλήση σε έναν άλλο κατασκευαστή – εάν δεν περιλαμβάνουν μια τέτοια κλήση, η Java προσθέτει τη **`super()`** κλήση αυτόματα

Περισσότερα για τους κατασκευαστές

- Επίσης, όλες οι κλάσεις έχουν τουλάχιστον έναν κατασκευαστή – εάν δεν περιλαμβάνουν έναν, η Java έμμεσα παρέχει έναν κατασκευαστή χωρίς ορίσματα
- Οι δύο παρακάτω ορισμοί κλάσεων είναι ισοδύναμοι:

```
public class OutOfGas extends Exception {
}
```

```
public class OutOfGas extends Exception {
  public OutOfGas() {
    super();
  }
}
```

```
public class OutOfGas extends Exception {
  private int miles;
  public OutOfGas(String details, int m) {
    super(details);
    miles = m;
  }
  public int getMiles() {
    return miles;
  }
}
```

```
try {
  throw new OutOfGas("You have run out of gas.", 42);
}
catch (OutOfGas e) {
  System.out.println(e.getMessage());
  System.out.println("Odometer: " + e.getMiles());
}
```

Ελεγχόμενες εξαιρέσεις

Ελεγχόμενες εξαιρέσεις

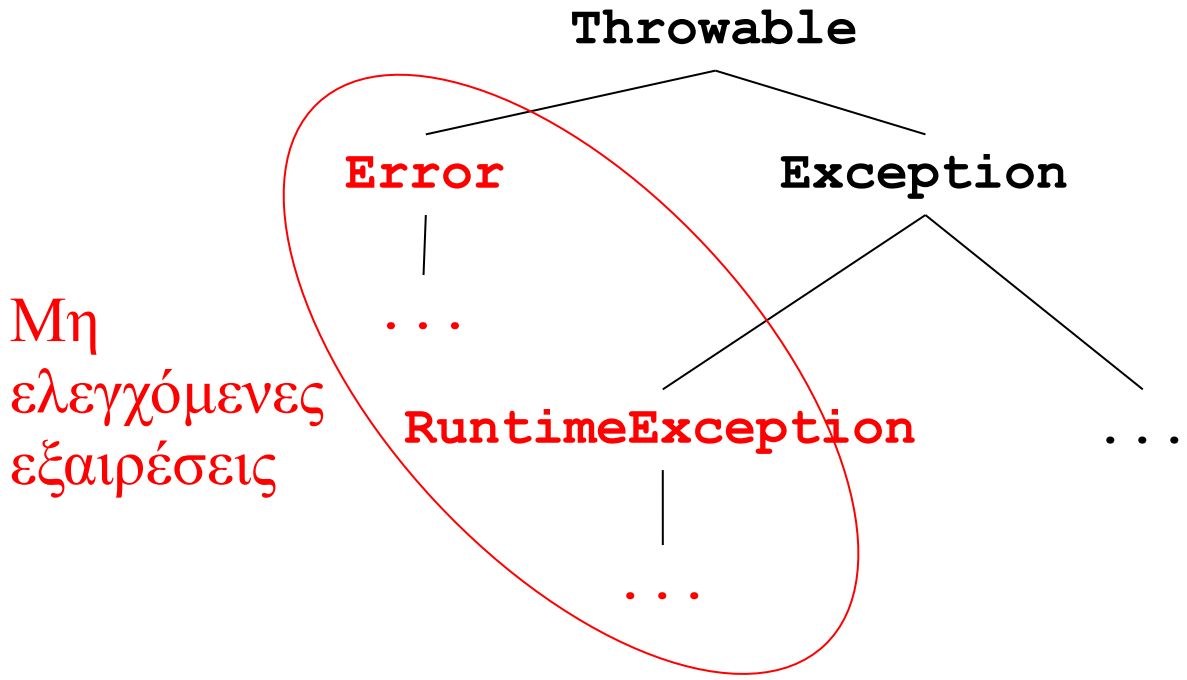
```
void z() {
  throw new OutOfGas("You have run out of gas.", 42);
}
```

- Ο μεταγλωττιστής της Java βγάζει μήνυμα λάθους για την παραπάνω μέθοδο:

  "**The exception OutOfGas is not handled**"

- Η Java δεν παραπονέθηκε μέχρι στιγμής για κάτι ανάλογο σε προηγούμενα παραδείγματα – γιατί τώρα;
- Αυτό συμβαίνει διότι η Java διαχωρίζει τις εξαιρέσεις σε δύο είδη: τις **ελεγχόμενες** και τις **μη ελεγχόμενες**

Ελεγχόμενες εξαιρέσεις



Οι κλάσεις των μη ελεγχόμενων εξαιρέσεων είναι η **`RuntimeException`** και οι απόγονοί της και η κλάση **`Error`** και οι απόγονοι της. Όλες οι άλλες κλάσεις εξαιρέσεων είναι κλάσεις ελεγχόμενων εξαιρέσεων.

Τι είναι αυτό που ελέγχεται;

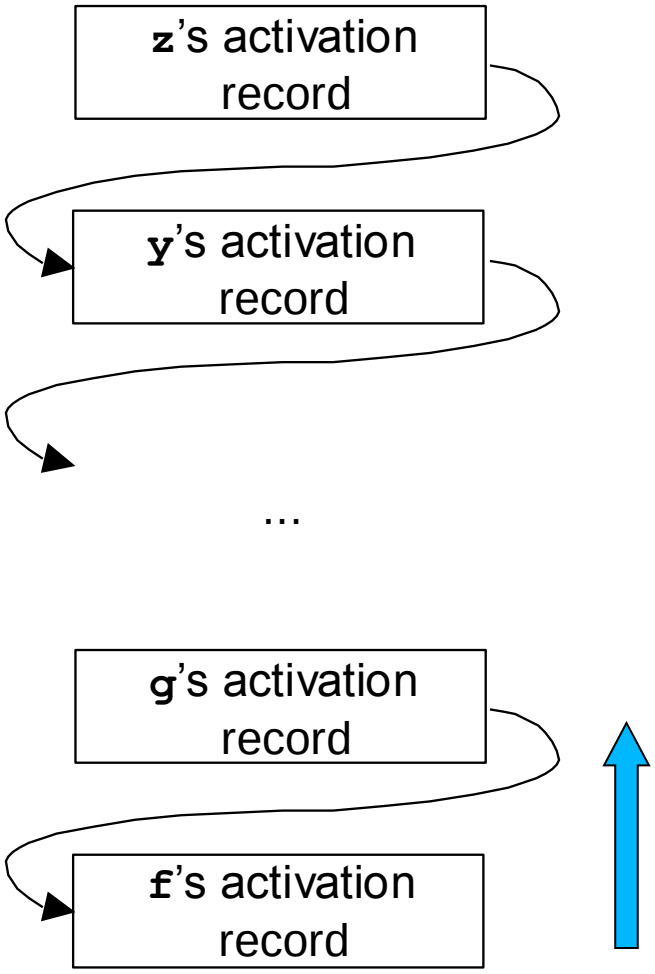
- Μια μέθοδος που μπορεί να δεχθεί μια ελεγχόμενη εξαίρεση δεν επιτρέπεται να την αγνοήσει
- Αυτό που πρέπει να κάνει είναι είτε να την πιάσει
  - Με άλλα λόγια, ο κώδικας που παράγει την εξαίρεση μπορεί να είναι μέσα σε μια εντολή **`try`** η οποία πρέπει έχει ένα **`catch`** το οποίο να πιάνει την ελεγχόμενη εξαίρεση
- Ή να δηλώσει ότι **δεν** την πιάνει
  - Χρησιμοποιώντας μια πρόταση **`throws`**

Η πρόταση `throws`

```
void z() throws OutOfGas {
  throw new OutOfGas("You have run out of gas.", 42);
}
```

- Μια πρόταση **`throws`** δηλώνει μια ή περισσότερες ελεγχόμενες κλάσεις που μπορεί να ρίξει η μέθοδος
- Αυτό σημαίνει ότι οι μέθοδοι που καλούν τη **`z`** είτε πρέπει να πιάσουν την εξαίρεση **`OutOfGas`** ή πρέπει επίσης να τη δηλώσουν στη δική τους πρόταση **`throws`**

- Εάν η μέθοδος **`z`** δηλώνει ότι **`throws OutOfGas`**…
- …τότε η μέθοδος **`y`** πρέπει να είτε να την πιάσει, ή να δηλώσει μέσω μιας **`throws`** πρότασης ότι επίσης την ρίχνει…
- …κοκ…
- σε όλες τις κλήσεις μέχρι την **`f`**



Για ποιο λόγο θέλουμε ελεγχόμενες εξαιρέσεις;

- Η πρόταση **`throws`** προσφέρει τεκμηρίωση της μεθόδου: λέει στον αναγνώστη ότι η συγκεκριμένη εξαίρεση μπορεί να είναι το αποτέλεσμα μιας κλήσης της μεθόδου
- Αλλά είναι μια **επικυρωμένη (verified)** τεκμηρίωση: εάν το αποτέλεσμα μιας κλήσης κάποιας μεθόδου ενδέχεται να είναι μια ελεγχόμενη εξαίρεση, ο compiler θα επιμείνει ότι η εξαίρεση αυτή πρέπει να δηλωθεί
- Άρα οι δηλώσεις των εξαιρέσεων μπορούν να κάνουν πιο εύκολη τόσο την κατανόηση όσο και τη συντήρηση των προγραμμάτων

Παράκαμψη των ελεγχόμενων εξαιρέσεων

- Αν δε θέλουμε ελεγχόμενες εξαιρέσεις, μπορούμε να χρησιμοποιήσουμε εξαιρέσεις οι οποίες είναι αντικείμενα κλάσεων που είναι επεκτάσεις της κλάσης **`Error`** ή της **`Throwable`**
- Οι εξαιρέσεις αυτές θα είναι μη ελεγχόμενες
- Όμως, θα πρέπει να λάβουμε υπόψη τα πλεονεκτήματα και τα μειονεκτήματα μιας τέτοιας κίνησης

Χειρισμός σφαλμάτων

Χειρισμός σφαλμάτων

- Παράδειγμα σφάλματος: απόπειρα εξαγωγής στοιχείου από μια κενή λίστα
- Τεχνικές:
  - Χρήση προϋποθέσεων (preconditions)
  - Χρήση καθολικών ορισμών (total definitions)
  - Θανατηφόρα λάθη (fatal errors)
  - Ένδειξη του σφάλματος (error flagging)
  - Χρήση εξαιρέσεων

Χρήση προϋποθέσεων

- Τεκμηριώνουμε (με τη μορφή σχολίων) όλες τις αναγκαίες προϋποθέσεις για την αποφυγή λαθών

```
/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is not empty.
 * @return the popped int
 */
public int pop() {
  Node n = top;
  top = n.getLink();
  return n.getData();
}
```

- Η καλούσα μέθοδος πρέπει είτε να εξασφαλίσει ότι οι προϋποθέσεις ισχύουν, ή να τις ελέγξει εάν δεν είναι βέβαιη ότι ισχύουν

```
if (s.hasMore()) x = s.pop();
else …
```

Μειονεκτήματα της χρήσης προϋποθέσεων

- Εάν κάποια κλήση ξεχάσει τον έλεγχο, το πρόγραμμα θα εγείρει κάποιο σφάλμα: **`NullPointerException`**
  - Εάν δε χειριστούμε το σφάλμα, το πρόγραμμα θα τερματίσει με ένα μήνυμα λάθους το οποίο δε θα είναι πολύ διευκρινιστικό
  - Εάν πιάσουμε το σφάλμα, για το χειρισμό του το πρόγραμμα ουσιαστικά θα πρέπει να βασιστεί σε κάποια μη τεκμηριωμένη πληροφορία για την υλοποίηση της στοίβας. (Εάν η υλοποίηση της στοίβας αλλάξει, π.χ. γίνει με χρήση πίνακα αντί για συνδεδεμένη λίστα, το σφάλμα εκτέλεσης θα είναι διαφορετικό.)

Καθολικός ορισμός

- Μπορούμε να αλλάξουμε τον ορισμό της **`pop`** έτσι ώστε να δουλεύει σε κάθε περίπτωση
- Δηλαδή να ορίσουμε κάποια “λογική” συμπεριφορά για το τι σημαίνει **`pop`** σε μια κενή στοίβα
- Κάτι αντίστοιχο συμβαίνει και σε άλλες περιπτώσεις, π.χ.
  - Στις συναρτήσεις για ανάγνωση χαρακτήρων από ένα αρχείο στη C που επιστρέφουν τον χαρακτήρα EOF εάν η ανάγνωση φτάσει στο τέλος του αρχείου
  - Στους αριθμούς κινητής υποδιαστολής κατά IEEE που επιστρέφουν NaN (για αριθμούς που δεν αναπαρίστανται) και συν/πλην άπειρο για πολύ μεγάλα/μικρά αποτελέσματα

```
/**
 * Pop the top int from this stack and return it.
 * If the stack is empty we return 0 and leave the
 * stack empty.
 * @return the popped int, or 0 if the stack is empty
 */
public int pop() {
  Node n = top;
  if (n == null) return 0;
  top = n.getLink();
  return n.getData();
}
```

Μειονεκτήματα των καθολικών ορισμών

- Μπορεί να κρύψουν σημαντικά προβλήματα του σχεδιασμού λογισμικού
- Για παράδειγμα, εάν ένα πρόγραμμα που χρησιμοποιεί μια στοίβα έχει πολύ περισσότερες κλήσεις **`pop`** από **`push`**, αυτό μάλλον δείχνει κάποιο προγραμματιστικό λάθος στη διεργασία το οποίο μάλλον πρέπει να διορθωθεί αντί να αποκρυφτεί

Θανατηφόρα λάθη

- Ελέγχουμε κάποιες προϋποθέσεις και εάν δεν ισχύουν σταματάμε την εκτέλεση του προγράμματος

```
/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty.  If called when the stack is empty,
 * we print an error message and exit the program.
 * @return the popped int
 */
public int pop() {
  Node n = top;
  if (n == null) {
    System.out.println("Popping an empty stack!");
    System.exit(-1);
  }
  top = n.getLink();
  return n.getData();
}
```

Μειονεκτήματα

- Το καλό με το συγκεκριμένο χειρισμό είναι ότι τουλάχιστον δεν κρύβουμε το πρόβλημα…
- Αλλά ο χειρισμός δεν είναι συμβατός με το στυλ του αντικειμενοστρεφούς προγραμματισμού: ένα αντικείμενο κάνει πράγματα μόνο στον εαυτό του, όχι σε ολόκληρο το πρόγραμμα
- Επιπλέον είναι κάπως άκαμπτος: διαφορετικές κλήσεις μπορεί να θέλουν να χειριστούν το σφάλμα διαφορετικά
  - Με τερματισμό
  - Με κάποια ενέργεια καθαρισμού των συνεπειών και τερματισμό
  - Με επιδιόρθωση και συνέχιση της εκτέλεσης
  - Με αγνόηση του σφάλματος

Ένδειξη του σφάλματος (error flagging)

- Η μέθοδος που ανιχνεύει κάποιο σφάλμα πρέπει να επιστρέψει μια ένδειξη για αυτό:
  - Επιστροφή μιας ειδικής τιμής (όπως κάνει π.χ. η **`malloc`** στη C)
  - Ανάθεση κάποιας τιμής σε μια καθολική μεταβλητή (όπως π.χ. η **`errno`** στη C)
  - Ανάθεση κάποιας μεταβλητής που ελέγχεται με κλήση μιας κατάλληλης μεθόδου (όπως π.χ. η **`ferror(f)`** στη C)
- Η καλούσα μέθοδος πρέπει να ελέγξει για την ύπαρξη σφάλματος

```
/**
 * Pop the top int from this stack and return it.
 * This should be called only if the stack is
 * not empty.  If called when the stack is empty,
 * we set the error flag and return an undefined
 * value.
 * @return the popped int if stack not empty
 */
public int pop() {
  Node n = top;
  if (n == null) {
    error = true;
    return 0;
  }
  top = n.getLink();
  return n.getData();
}
```

```
/**
 * Return the error flag for this stack.  The error
 * flag is set true if an empty stack is ever popped.
 * It can be reset to false by calling resetError().
 * @return the error flag
 */
public boolean getError() {
  return error;
}

/**
 * Reset the error flag.  We set it to false.
 */
public void resetError() {
  error = false;
}
```

```
/**
 * Pop the two top integers from the stack, divide
 * them, and push their integer quotient.  There
 * should be at least two integers on the stack
 * when we are called.  If not, we leave the stack
 * empty and set the error flag.
 */
public void divide() {
  int i = pop();
  int j = pop();
  if (getError()) return;
  push(i/j);
}
```

Όλες οι τεχνικές ένδειξης σφαλμάτων απαιτούν κάποιον ανάλογο έλεγχο για την ύπαρξη ή όχι σφάλματος.

Παρατηρήστε ότι οι μέθοδοι που καλούν την **`divide`** πρέπει επίσης να ελέγξουν για σφάλμα, όπως και οι μέθοδοι που καλούν τις μεθόδους με κλήσεις της **`divide`**, κοκ...

Χρήση εξαιρέσεων

- Με χρήση εξαιρέσεων, η μέθοδος που ανιχνεύει πρώτη το σφάλμα εγείρει κάποια εξαίρεση
- Η εξαίρεση μπορεί να είναι ελεγχόμενη ή μη ελεγχόμενη
- Οι εξαιρέσεις είναι μέρος της τεκμηριωμένης συμπεριφοράς της μεθόδου

```
/**
 * Pop the top int from this stack and return it.
 * @return the popped int
 * @exception EmptyStack if stack is empty
 */
public int pop() throws EmptyStack {
  Node n = top;
  if (n == null) throw new EmptyStack();
  top = n.getLink();
  return n.getData();
}
```

```
/**
 * Pop the two top integers from the stack,
 * divide them, and push their integer quotient.
 * @exception EmptyStack if stack runs out
 */
public void divide() throws EmptyStack {
  int i = pop();
  int j = pop();
  push(i/j);
}
```

Η καλούσα μέθοδος δεν ελέγχει για σφάλμα—απλώς προωθεί την εξαίρεση

Πλεονεκτήματα

- Έχουμε διευκρινιστικά μηνύματα λάθους ακόμα και εάν δεν πιάσουμε την εξαίρεση
- Οι εξαιρέσεις είναι μέρος της τεκμηριωμένης διαπροσωπείας των μεθόδων
- Σφάλματα εκτέλεσης πιάνονται άμεσα και δεν αποκρύπτονται
- Η καλούσα μέθοδος δε χρειάζεται να ελέγξει για σφάλμα
- Ανάλογα με την περίπτωση, έχουμε τη δυνατότητα είτε να αγνοήσουμε είτε να χειριστούμε κατάλληλα το σφάλμα

Ολόκληρη η σύνταξη του `try`

```
<try-statement>  ::=  <try-part>  <catch-parts>
                   |  <try-part>  <catch-parts>  <finally-part>
                   |  <try-part>  <finally-part>
<try-part>  ::=  try  <compound-statement>
<catch-parts>  ::=  <catch-part>  <catch-parts>  |  <catch-part>
<catch-part>  ::=  catch  (<type>  <variable-name>)
                        <compound-statement>
<finally-part>  ::=  finally  <compound-statement>
```

- Ένα **`try`** έχει ένα προαιρετικό σκέλος **`finally`**
- Το σώμα του **`finally`** εκτελείται πάντα στο τέλος της εντολής **`try`**, ό,τι και αν συμβεί

Χρήση του `finally`

- Το σκέλος **`finally`** συνήθως χρησιμοποιείται για κάποιες λειτουργίες καθαρισμού (που είναι απαραίτητες να γίνουν)
- Για παράδειγμα, ο παρακάτω κώδικας κλείνει το αρχείο ανεξάρτητα του αν εγερθεί κάποια εξαίρεση ή όχι

```
file.open();
try {
  workWith(file);
}
finally {
  file.close();
}
```

Άλλο ένα παράδειγμα

```
System.out.print("1");
try {
  System.out.print("2");
  if (true) throw new Exception();
  System.out.print("3");
}
catch (Exception e) {
  System.out.print("4");
}
finally {
  System.out.print("5");
}
System.out.println("6");
```

Τι τυπώνεται;

Τι θα συμβεί εάν αλλάξουμε το **`new Exception()`**

- σε **`new Error()`**;
- σε **`new Throwable()`**;

Αποχαιρετισμός στη Java

Μέρη της Java που δεν εξετάσαμε (1)

- Θεμελιώδεις εντολές της γλώσσας
  - **`do`**, **`for`**, **`break`**, **`continue`**, **`switch`**
- Εκλεπτύνσεις
  - Εσωτερικές κλάσεις που ορίζουν κλάσεις με εμβέλεια: μέσα σε άλλες κλάσεις, σε μπλοκ, σε εκφράσεις
  - Την εντολή **`assert`** (Java 1.4)
- Πακέτα (packages)
  - Οι κλάσεις ομαδοποιούνται σε πακέτα
  - Σε πολλές υλοποιήσεις της Java, όλα τα αρχεία πηγαίου κώδικα σε κάποιο directory αντιστοιχούν σε ένα πακέτο
  - Η προκαθορισμένη προσπέλαση (χωρίς **`public`**, **`private`** ή **`protected`**) έχει εύρος πακέτου

Μέρη της Java που δεν εξετάσαμε (2)

- Δομές ταυτοχρονισμού
  - Γλωσσικές δομές συγχρονισμού (synchronization constructs) για πολλαπλά νήματα εκτέλεσης
  - Μέρη του API για τη δημιουργία νημάτων
- Το πολύ μεγάλο σε έκταση API
  - containers (stacks, queues, hash tables, etc.)
  - graphical user interfaces
  - 2D and 3D graphics
  - math
  - ταίριασμα προτύπων με κανονικές εκφράσεις
  - file I/O, network I/O και XML
  - encryption και ασφάλεια
  - remote method invocation (RMI)
  - interfacing to databases and other tools