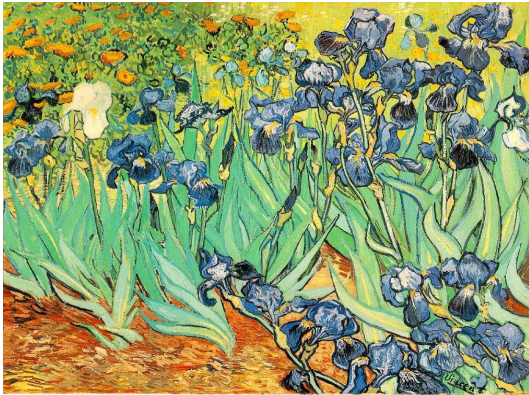


Εξαιρέσεις (στη Java)



Vincent van Gogh, *Iris*, 1889

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

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)
```

Εξαιρέσεις

2

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

Εξαιρέση της Java	Κώδικας που την εγείρει
<code>NullPointerException</code>	<code>String s = null; s.length();</code>
<code>ArithmeticException</code>	<code>int a = 42; int b = 0; int q = a/b;</code>
<code>ArrayIndexOutOfBoundsException</code>	<code>int[] a = new int[10]; a[42];</code>
<code>ClassCastException</code>	<code>Object x = new Integer(42); String s = (String) x;</code>
<code>StringIndexOutOfBoundsException</code>	<code>String s = "Hello"; s.charAt(8);</code>

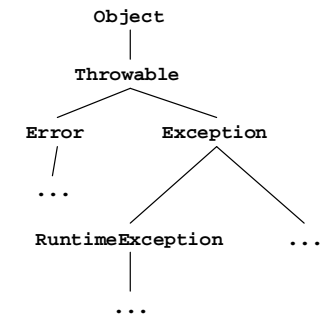
Μια εξαίρεση είναι ένα αντικείμενο

- 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

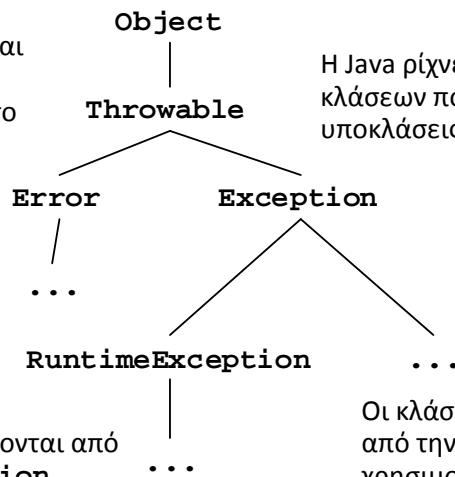
Ριπτόμενες κλάσεις

- 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

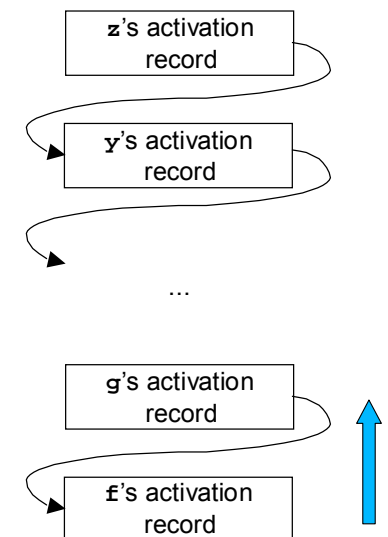
- Η εντολή `try` έχει την ευκαιρία να πιάσει εξαιρέσεις που πιθανώς να εγερθούν από την εκτέλεση του σκέλους `try`
- Αυτό περιλαμβάνει όλες τις εξαιρέσεις που ρίχνονται από μεθόδους που καλούνται (αμέσως ή εμμέσως) από το σώμα του `try`

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 ταιριάζει 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());
}
```

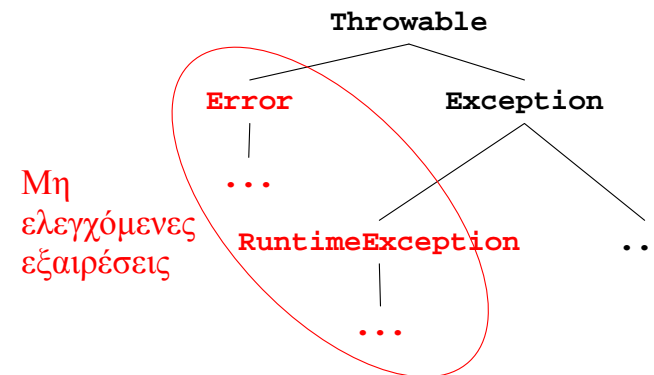
Ελεγχόμενες εξαιρέσεις

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



Μη
ελεγχόμενες
εξαιρέσεις

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

Τι είναι αυτό που ελέγχεται;

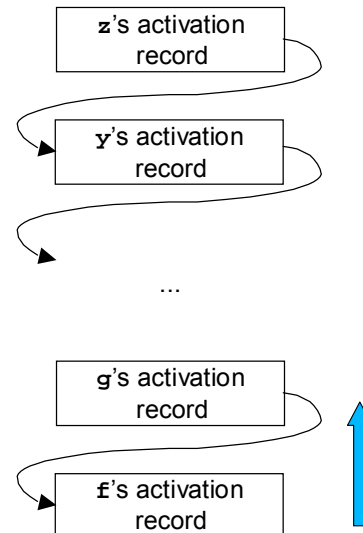
- Μια μέθοδος που μπορεί να δεχθεί μια ελεγχόμενη εξαίρεση δεν επιτρέπεται να την αγνοήσει
- Αυτό που πρέπει να κάνει είναι είτε να την πιάσει
 - Με άλλα λόγια, ο κώδικας που παράγει την εξαίρεση μπορεί να είναι μέσα σε μια εντολή `try` η οποία πρέπει έχει ένα `catch` το οποίο να πιάνει την ελεγχόμενη 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();
}
```

Εξαιρέσεις

45

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 άκαμπος: 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

Εξαιρέσεις

46

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();
}
```

Εξαιρέσεις

47

Εξαιρέσεις

48

```

/**
 * 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;
}

```

Εξαιρέσεις

49

```

/**
 * 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...

Εξαιρέσεις

50

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

Εξαιρέσεις

51

Εξαιρέσεις

52

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

Εξαιρέσεις

53

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

Εξαιρέσεις

54

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();
}
```

Εξαιρέσεις

55

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()**;

Εξαιρέσεις

56

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