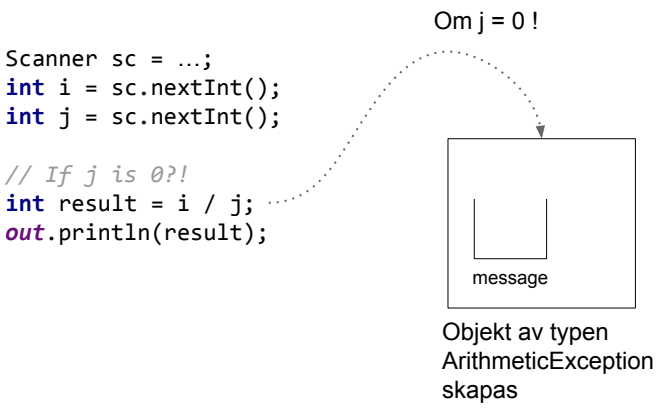
Undantag och Filhantering

DIT012/Joachim von Hacht

Undantag



2

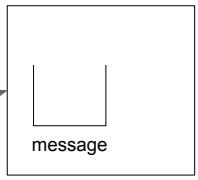
Programmet har under körning hamnat i en omöjlig situation (division med 0)

- Ett undantag (exception) uppstår
- I samband med undantaget:
  - Skapas automatisk ett objekt som bl.a. innehåller information om undantaget
  - Avbryts programmet …. om inte undantaget fångas, mer strax…
    - En felutskrift skrivs ut. Se Grunderna.

Fånga undantag

```
int i = ...;
int j = ...;
int result;

try {
    result = i / j; // If 0..
} catch (AritmeticException e ){
    // .. do this
    out.println("You divided by zero!");
}
out.println(result);
```



3

Att fånga ett undantag innebär att programmet inte avbryts

- Man fångar ett undantag genom att lägga ett anrop som kan kasta ett undantag i try-delen av en **try-catch-sats**
- Om inget undantag uppstår körs bara satserna i try-blocket, catch-blocket hoppas över.
- Om ett undantag uppstår, någonstans i try-delen, kommer satserna i blocket efter catch att utföras
    - Kvarvarande satser (efter undantaget) i try-blocket körs inte
    - Tanken är att man, i catch-delen, skall kunna åtgärda felet
    - Efter detta räknas undantaget som fångat och programmet fortsätter med första sats efter catch-blocket
- I catch-grenen initieras automatisk en parameter med en referens till undantagsobjektet som skapades
    - Typen på undantagsobjektet måste stämma med parametertypen
        - Annars sker ingen “fångning”

Kasta Undantag

```
public void add( String s ){
  if( s == null){
    // Create exception object and throw
    throw new IllegalArgumentException("nulls not allowed");
  }
  arr[i] = s;
  i++;
}


try {
    add( str );    // Call method
} catch (IllegalArgumentException e ){
    out.println( e.getMessage() ); // nulls not allowed
}
```

4

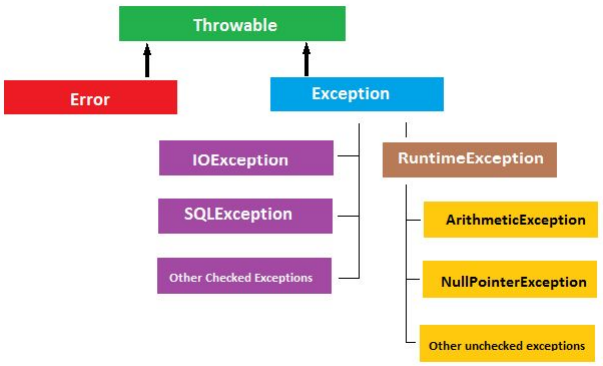
En metod kan “kasta” undantag.

- Antag t.ex. att vi har en metod som sparar referenser i en array och vi inte tillåter null-referenser i arrayen …
- … om någon annan programmerare gör fel och skickar in en referens, … vad göra???
- Jo, vi kan låta metoden kasta ett undantag om parametern är null … på så sätt upptäcks felet direkt (i stället för att null-värdet sparas och felet dyker upp långt senare).

Ett eget undantag kastas m.h.a. throw + att man skapar ett objekt av någon undantagsklass, mer strax.

- Meddelandet, argumentet till konstruktorn, kan avläsas med metoden e.getMessage() i catch-grenen
- Finns även e.printStackTrace(), skriver ut hela anropskedjan (stacken)

Undantagsklasser



5

Undantagsobjekten är instanser av olika undantagsklasser

- … klasserna är ordnade i en hierarki

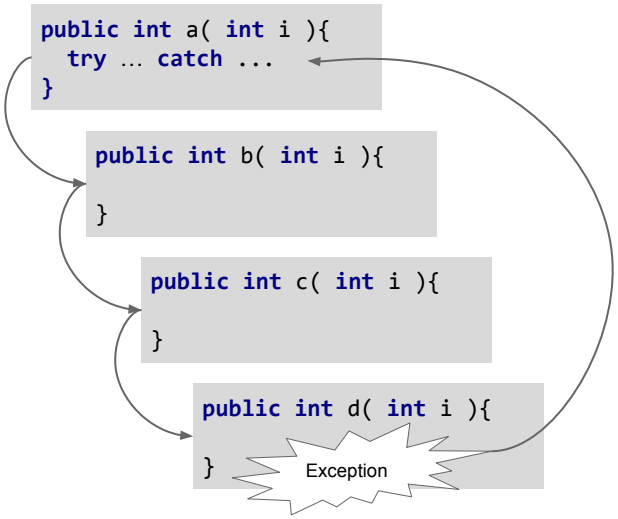
Undantagsklasserna direkt under Exception (de lila) representerar undantag som måste fångas (checked exceptions)

- D.v.s. situationer som kan kasta ett undantagsobjekt av dessa typer måste stå i en try..catch-sats
  - Kompilatorn kontrollerar, vi får ett kompileringsfel om vi inte fångar!
- Många färdiga Java-metoder kan kasta dessa typer av undantag, mer strax

Undantagsklasserna under Runtime är okontrollerade undantag (unchecked exception)

- Vi är inte tvungna att fånga dem.
- Normalt fångar man inte unchecked exceptions …
  - … man vill att undantaget skall krascha programmet (under utvecklingen) eftersom det finns ett programmeringsfel (vi har gjort fel)

Programflöde vid Undantag



6

Ett undantag kan uppstå långt ner i en anropskejda

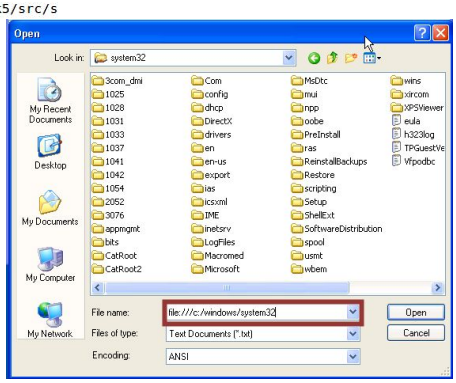
- I bilden: Metod a anropar b, som anropar c, som anropar d … ett undantag uppstår.
- Undantagshantering kommer att vandra genom anropen (anropsstacken) till dess den hittar en try..catch som kan fånga undantaget.
  - Vi får ett **icke-lokalt hopp** …
  - … programmet hoppar och fortsätter i en annan metod än den anropande.. (kanske väldigt långt bort, i någon helt annan klass…)
  - Finns ingen try-catch avbryts som sagt programmet.

Filhantering

```
hajo/samples$ pwd
/home/hajo/courses/ooit/ass/ass.src/week5/src/s
amples
hajo/samples$ tree
.
├── Exceptions.java
├── filehandling
│   ├── BinaryFiles.java
│   ├── ObjectIO.java
│   ├── TextFiles.java
│   └── UseFile.java
├── OrderNumberStatic.java
├── ShortForLoop.java
├── StaticVaribles&Methods.java
├── StringMethods.java
├── Strings.java
└── unused
    ├── CommandLinArgs.class
    ├── CommandLinArgs.java
    ├── HigherOrderFunctions.java
    ├── RemoveFromCollection.java
    ├── StringBuilderThis.java
    ├── UseAList.java
    ├── UseAStack.java
    └── UseCharacter.java

2 directories, 18 files
hajo/samples$
```

Kommandot tree i Linux



Windows filhanterare

En fil är en sammanhållen informationsmängd.

- Lagras på någon typ av sekundärminne.
- En fil är beständig, d.v.s. den existerar efter det att programmet som skapade filen är avslutat.
- Filer har storlekar (kB, MB, GB), en tidsstämpel, en ägare, rättigheter, m.m.
- Vissa filer kan fungera som behållare för andra filer, kallas bibliotek (eller kataloger eller mappar, directory)
- Det finns många filformat

Filhantering innebär typiskt att: Skapa ny fil, öppna, läsa, skriva, lägga till, ändra, stänga, ta bort, kopiera, flytta fil(er).

- Sköts ofta av användare (människor) men behöver också kunna skötas av program.

Filsystem och Sökvägar

Absoluta

```
// Windows  C: is root
C:\Documents\sally\statusReport
C:\Documents\sally\statusreport  // Case insensitive, same dir

// Unix, Linux, Mac. leading / is root
/home/sally/statusReport
/home/sally/statusreport     // Case sensitive! This is another dir
```

Relativa

```
// Windows  "." = curent dir, ".." = dir one step up
.\sally\statusReport
..\sally\statusReport

// Unix, Linux, Mac
./sally/statusReport
../sally/statusReport
../../sally/statusReport
```

8

Filerna i ett system är normalt ordnade i en (upp och nedvänd) trädstruktur, ett filsystem

- En sökväg (path) betecknar en unik position i filsystemet (trädstrukturen)
- Sökvägen kan vara absolut d.v.s. ange hela vägen från trädets rot till positionen …
- … eller relativ d.v.s. bara ange vägen utifrån en given plats i trädet.
    - Om vi använder absoluta sökvägar i programmet måste programmet alltid (på alla datorer) ligga på samma ställe i filsystemet.
- Sökvägar byggs upp av en rotbeteckning, biblioteksnamn samt ett avgränsningstecken "/" (för Unix/Linux/Mac) alternativt "\" för Windows.
- Alla filsystem utom Windows är case sensitive.
- Om ett program skall fungera på flera olika operativsystem måste vi se till att sökvägarna/filnamnen fungerar för alla
    - Finns sätt att göra detta i Java t.ex. m.h.a. System.

Sökvägar i IntelliJ utgår från projektkatalogen

- Innebär att om filen finns i någon mapp i src får vi inleda med sökvägen med "src/…/…".

Innehållet i en textfil tolkas som tecken och är strukturerat i rader (en Java-fil är en textfil)

- Radsluten beror på operativsystem (vilka tecken (koder) som används för radslut)
  - Rött i bilden.

När filen sparas/läses sker en omvandling från/till bytes till/från tecken (t.ex. Unicode)

- 199 sparas som ‘1’ ‘9’ ‘9’ vilket kodas som 0x31, 0x39, 0x39 (hexadecimalt)

Binärfiler

feca beba 0000 3400 5000 000a 0015 072a 2b00 0009 002c 0a2d
0200 2e00 0009 0006 072f 3000 000a 0006
0a2a 0600 3100 0009
002c 0832 3300 000a 0034 0a35 0200 3600 0008 0737 3800 000a
000e 082a 3900 000a 000e 0a3a 0e00 3b00 000a 000e 0a3c 3400
3d00 0007 013e 0200 6373 0001 4c13 616a
6176 752f 6974 2f6c 6353 6e61 656e 3b72
0001 3c06 6e69 7469 013e 0300 2928 0156 0400
6f43 6564 0001 4c0f 6e69 4e65 6d75
6562 5472 6261 656c 0001 4c12 636f 6c61 6156 6972 6261 656c
6154 6c62 0165 0400 6874 7369 0001 4c13 7865 7265 6963 6573
2f73 7845 5f31 4d42 3b49 0001 6d04 6961 016e 1600 5b28 6a4c
7661 2f61 616c 676e 532f 7274 6e69 3b67 5629 0001 6104 6772
0173 1300 4c5b 616a 6176 6c2f 6e61 2f67
…
...

Binärfil utskriven på hexadecimal form (raderna beror från utskriften)

Binärfiler är ren digital data

- Ej ordnade i rader, ej läsbara för människor
- Ingen omvandling av data sker, ren binär data skrivs och läses

Läsa en Textfil

```
// Method to read a file. NOTE throws
List<String> readFile(Path path) throws IOException {
   List<String> lines = new ArrayList<>();
    // Use "try with resources" style to ensure stream is closed
   try (BufferedReader reader = Files.newBufferedReader(path)) {
       String line;
       while ((line = reader.readLine()) != null) {
           lines.add(line);
       }
       return lines;
   }
}


// Call must use try-catch, method throws checked exception
try {
   Path path = Paths.get("src/samples/filehandling/junk.txt");
   List<String> copy = readFile(path);
} catch (IOException e) {
   e.printStackTrace();
}
```

11

Finns flera sätt (nya och gamla) att läsa och skriva textfiler i Java

- Förvirrande för nybörjare (och andra)
- Dessutom ganska tekniskt komplicerat
- Detta är inget man lär sig utantill

Analys av kod:

- Vi ska läsa en textfil
- För detta har vi skapat en metod readFile. I metoden kan det kastas checked exceptions (beror på de färdiga Java-metoder vi anropar).
  - Någon måste hantera dessa.
  - Vi väljer att inte göra det i metoden utan skickar ansvaret till den som anropar metoden genom att ange throws i metodhuvudet. Detta gäller bara checked exception!
- Innehållet i filen spar vi i en List<String>, varje rad lagras som en sträng i listan
- Vi använder dessutom en speciell form av try, kallas try-with-resources.
  - Problemet är att vi måste se till att inströmmen (som används i bakgrunden) stängs ifall ett undantag uppstår, annars kan programmet "äta" upp en massa resurser (minne).
  - Om vi använder try-with-resources garanterar Java att strömmen stängs.

- I try satsen kör vi en loop som läser rad för rad
    - Om det inte finns någon mer rad kommer line-variabeln att innehålla null
        - .. om så bryter vi loopen
    - Annars lägger vi till raden i List.
- Vid anropet till readFile ger vi ett Path-objekt som argument till metoden

Skriva en Textfil

```
void writeFile(Path path, List<String> lines) throws IOException {
   try (BufferedWriter writer = Files.newBufferedWriter(path)) {
       for (String line : lines) {
           writer.write(line);
           writer.newLine();
       }
   }
}


List<String> text = Arrays.asList("a", "b", "c", "d", "e");
try {
   Path path = Paths.get("src/samples/filehandling/junk.txt");
   writeFile(path, text);
  } catch (IOException e) {
   e.printStackTrace();
}
```


Påminner mycket om att läsa en fil.

Läsa ett Objekt (binärfil)

```
class MyClass implements Serializable {
   private String name;
   public MyClass(String name) { this.name = name;}
   public String getName() { return name;}
   public void setName(String name) {this.name = name;}
}


MyClass readObject(Path path) throws IOException, ClassNotFoundException {
  try (ObjectInputStream is = new ObjectInputStream(Files.newInputStream(path));){
       return (MyClass) is.readObject(); // Read full object from file
  }
}

// Call
try {
   Path path = Paths.get("src/samples/filehandling/pelle.ser" );
   MyClass o = readObject(path);
} catch (IOException | ClassNotFoundException  e) {
   e.printStackTrace();
}
```


Man kan skriva och läsa objekt från till (binär)filer.

- Klassen måste ange implements Serializable (d.v.s. det går att omvandla objekten till byteströmmar)
- För att kunna läsa objekt måste det finnas en klass (att omvandla bytes:en till)
    - Annars ClassNotFoundException
- Det vi läser måste explicit typomvandlas

Skriva ett Objekt (binärfil)

```
void writeObject(Path path, MyClass object) throws IOException {
   try (ObjectOutputStream os =
                  new ObjectOutputStream(Files.newOutputStream(path))) {
       os.writeObject(object);  // Write full object to file
   }
}


// Create object
MyClass o = new MyClass("pelle");

// Call
try {
   Path path = Paths.get("src/samples/filehandling/pelle.ser" );
   writeObject(path, o);
} catch (IOException | ClassNotFoundException  e) {
   e.printStackTrace();
}
```


Ungefär som att läsa

Klassen File

```
File f = new File("src/samples/tmp");
out.println(f.mkdir());   // Create directory
out.println(f.getName());
out.println(f.exists());
out.println(f.isDirectory());
out.println(f.getAbsolutePath());
try {
   File junk = new File("src/samples/tmp/junk.txt");
   out.println(junk.createNewFile());
} catch (IOException e) {
   e.printStackTrace();
}
File[] files = f.listFiles(); //Content of directory
out.println(files.length);
out.println(files[0]);
```


Filhantering i ett Java-program kan skötas m. h. a. den färdiga klassen File.

- Konstruktorn och många av metoderna kastar kontrollerade undantag, måste hanteras.