Klassfiler, Undantag, filhantering

Vecka 5, Bildserie 2

Innehåll

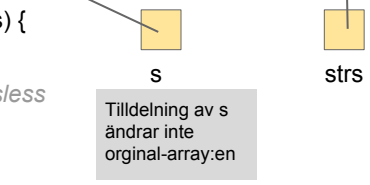
- Undantag
- Sökvägar
- Klassen File
- Filhantering
- Textfiler
- Binärfiler

Att Läsa i Boken

- 7.13
- 12.1-12.6
- 12.10
- 12.11

Kort for-loop

```
int[] is = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i : is) {      // No index just each element, left to right.
   out.print(i);
}

String strs[] = {"a", "b", "c", "d", "e", "f"};

for (String s : strs) {
   out.print(s);
   s = "X"; // Sensless
}
```



Tilldelning av s ändrar inte orginal-array:en

Om man bra vill traversera en array (eller annan samling, mer senare) finns en enklare for-loop

- Tanken är att men bara skall avläsa värden (och eventuellt producera ett resultat utifrån dessa)
  - Loopen tar ett element i taget från början till slut (index [0-(length-1)]
  - Använd om ni vill … kan förkomma i kodexempel
- Att tilldela loopvariabeln (tex. i och s i koden) är meningslöst.
  - Ändrar inte original array:en
- Behöver man ett index måste man använda den vanliga for- eller while-loopen.

Kommandoradsargument

```
public class MyProgram {
    public static void main(String[] args) {
        out.println(args[0]);    // path/to/config
        out.println(args[1]);    // 23
    }
}


// Compile using command line
$ javac MyProgram.java

// Run using command line and args
$ java MyProgram /path/to/config 23
```

Ofta vill man skicka med information till ett program då det startas (t.ex. konfigurationsinformation).

I Java (och andra språk) gör man detta med kommandoradsargument

- Då man startar programmet från kommandoraden skriver man ett antal strängar efter namnet på programmet.
- Dessa strängar skickas då (m.h.a. operativsystemet) till programmet (och till main metoden)
- Man kommer åt strängarna m.h.a. argumentet args i main-metoden.
    - I samma ordning som man skrev dem.

Verkar vara svårt att använda detta i IntelliJ

- Dock kan man öppna en terminal i IntelliJ och köra programmet från denna.

Mallen sista ...

```
public class MyProgram {
     public static void main(String[] args) {
        new MyProgram().program();
    }

   void program() {
        ...
   }

}
```

Sista analys av mallen

- public static void main är en klassmetod, behövs inget objekt för att anropa
    - D.v.s. Java kan anropa metoden så fort klassen är laddad i minnet.
    - Parametrarna args kommer från kommandoraden
- Vi skapar ett objekt av klassen
- Vi anropar en metod direkt på objektet
- När metoden är klar (och det inte finns några referenser till objektet) skräpsamlas objektet.

Undantag

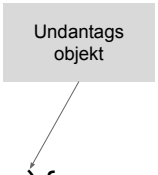
```
int i = sc.nextInt();
int j = sc.nextInt();

// If j is 0?!
int result = i / j;
out.println(result);
```

Programmet har under körning hamnat i en omöjlig situation

- Ett undantag (exception) uppstår
- Görs inte något åt detta avbryts programmet … (kraschar)
- .. inte acceptabelt!
- Vi måste kunna hantera undantag så att programmet kan fortsätta eller avslutas på ett kontrollerat sätt.

Fånga undantag

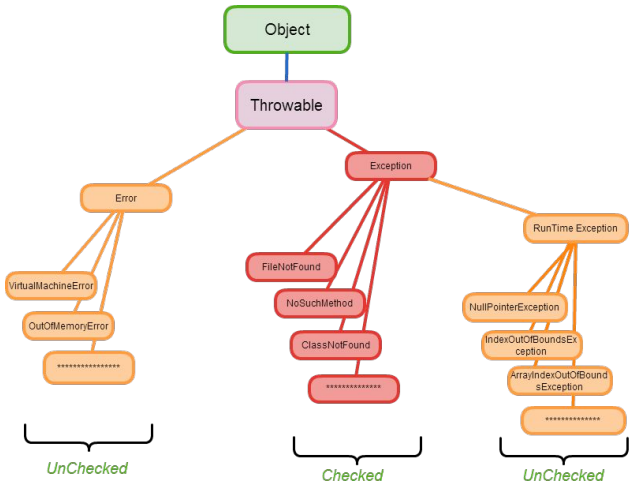
```
int i = sc.nextInt();
int j = sc.nextInt();
int result;

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



Program kan "fånga" undantag

- Att fånga ett undantag innebär att programmet inte avbryts
- Man fångar ett undantag genom att lägga ett anrop som kan generera 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 hanterat och programmet fortsätter med första sats efter catch-blocket
    - catch får automatiskt ett undantagsobjekt (e i bilden) som parameter (används inte i bilden)
        - Typen på undantaget måste stämma med parametertypen för undantagsobjektet vid catch
        - Annars sker ingen "fångning", mer strax ...
- Man kan lägga till ett finally block efter catch-blocket
    - finally blocket är garanterat att alltid köras!

Undantagklasser



Undantagsobjektet innehåller information om undantaget

- Finns många olika klasser för olika sorters undantagsobjekt...
- … de är ordnade i en super/subklass hierarki,
- Kan även skapa egna undantagsklasser (vilket vi inte gör)

Vissa undantag är kontrollerade (“checked”)

- Innebär att de måste fångas
- Om ett objekt av någon typ under Exception kan kastas innebär det att undantaget är checked
- Om så, kontrolleras redan vid kompileringen att undantaget hanteras … om ej kompileringsfel!.
- De som inte är checked är unchecked (icke kontrollerade).

Kasta Eget Undantag

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

Ett program kan generera 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).

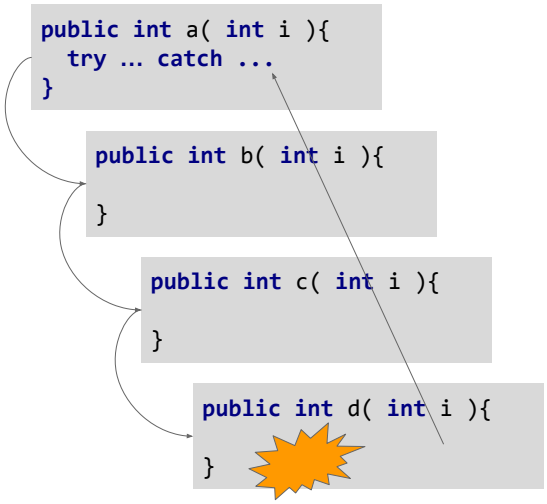
I sådana situationer använder man unchecked exceptions, man vill ju att undantaget skall krascha programmet eftersom det finns ett fel (någon har gjort fel).

- Normalt fångar man alltså inte unchecked exceptions.

Ett eget undantag genereras m.h.a. throw + att man skapar ett objekt av någon undantagsklass

- Det objekt man skapar dyker upp som parameter vid catch
- Meddelandet man skickar till konstruktorn kan avläsas med metoden e.getMessage().

Programflöde vid Undantag

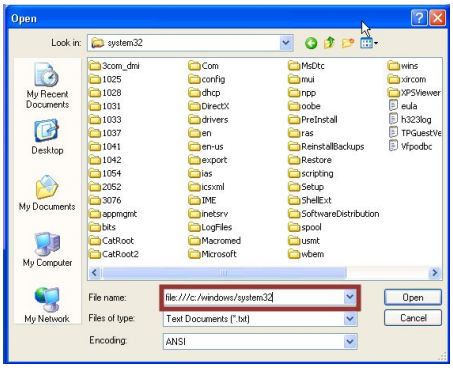


Ett undantag kan uppstå långt ner i en anropskejda

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

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$
```



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 tidstämpel, en ägare, rättigheter, m.m.
- Vissa filer kan fungera som behållare för andra filer, kallas bibliotek (eller kataloger eller mappar, directory)

Filhantering innebär typiskt att: Skapa ny fil, öppna, läsa, skriva, 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
```

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.
- Sökvägar byggs upp av en rotbeteckning, biblioteksnamn samt avgränsningstecknet "/" (för Unix/Linux/Mac) alternativt "\" för Windows.
- Alla filsystem utom Windows är case sensitive.

Sökväger i IntelliJ utgår från projektkatalogen

- Innebär att om filen finns i någon mapp i src får vi inleda med sökvägen med "src/…/…".
- Om vi använder absoluta sökväger i programmet måste programmet alltid (på alla datorer) ligga på samma ställe i filsystemet, inte så bra!

Klassen File

```
File f = new File("src/samples/tmp");
out.println(f.mkdir());
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();
out.println(files.length); // 1
out.println(files[0]);     // src/samples/tmp/junk.txt
```

För att sköta filhantering i ett program använder man bl.a. den färdiga klassen File.

- e.printStackTrace() dumpar hela felmeddelan (exakt plats för fel och i vilken fil)

Textfiler

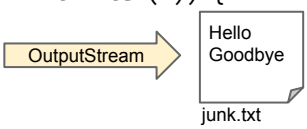
```
// Unix/Linux
Det är skimmer i molnen och glitter i sjön,LF
det är ljus över stränder och näs,LF
och omkring står den härliga skogen grönLF
bakom ängarnas gungande gräs.LF

// Mac
Det är skimmer i molnen och glitter i sjön,CR
det är ljus över stränder och näs,CR
och omkring står den härliga skogen grönCR
bakom ängarnas gungande gräs.CR

// Windows
Det är skimmer i molnen och glitter i sjön,CRLF
det är ljus över stränder och näs, CRLF
och omkring står den härliga skogen grön CRLF
bakom ängarnas gungande gräs.CRLF
```

Innehållet i en textfil tolkas som tecken och är strukturerat i rader

- Radsluten beror på operativsystem (vilka tecken (koder) som används för radslut)
  - Använd System.lineSeparator() i Java för att klara olika OS
- 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)

Skriva till en Textfil

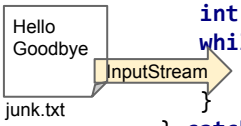
```
File f = new File("src/samples/filehandling/junk.txt");
// This will ensure the PrintWriter is closed
try (PrintWriter out = new PrintWriter(f)) {
   out.println("Hello");
   out.println("Goodbye");
} catch (IOException e) {
   e.printStackTrace();
}
```



Använd en PrintWriter

- Det kontrollerade undantaget IOException kan uppstå, måste hantera!
- .. annars kompileringsfel.
- Vi använder dessutom en speciell form av try, kallas try-with-resources.
    - Problemet är att vi måste se till att utströmmen 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 (och resurserna frigörs)

Läsa från en Textfil




```
// Assume we know how many lines in file
String[] lines = new String[10];
File file = new
File("src/samples/filehandling/junk.txt");
// This will ensure the Scanner is closed
try (Scanner sc = new Scanner(file)) {
   int i = 0;
   while (sc.hasNextLine()) {
      lines[i++] = sc.nextLine();
   }
} catch (IOException e) {
   e.printStackTrace();
}
```

Genom att koppla File-objektet till en Scanner kan programmet läsa från en fil.

- OBS! Att readLine() tar bort radslutstecknet.
- Vill man behålla radstrukturen kan man använda en array (som i bilden)

Skicka vidare Undantag

```
// Now, caller must handle exception
public void writeToFile() throws FileNotFoundException {
   File f = new File("src/samples/filehandling/junk.txt");
   try (PrintWriter out = new PrintWriter(f)){
       out.println("Hello");
       out.println("Goodbye");
    }
}

// Somewhere
try {
   writeToFile();
} catch (FileNotFoundException e) {
   e.printStackTrace();
}
```

Ofta kan man inte hantera ett checked exception precis där det uppstod

- Vill kanske visa en dialogruta där användaren kan avgöra vad som skall ske.
- Om så kan man skicka vidare undantaget till anropande metod (som i sin tur kan skicka vidare)
- Någonstans i anropskedjan måste det finnas en try-sats, annars kompileringfel (eftersom undantaget är kontrollerat)

Man kan i en metod välja att inte hantera ett kontrollerat undantag

- Om så måste man i metodhuvudet ange att metoden kastar ett undantag, anges med **throws**
- Kan ge "grötig" kod om man har långa anropskedjor där flera undantag kan uppstå.

Programmering

```
// Copy file junk.txt to file copyofjunk.txt
copy("./src/samples/filehandling",
    "junk.txt","./src/samples/filehandling", "copyofjunk,txt"
);
```

Hur skriver man metoden?

- Metoden kopierar en textfil.

Binärfiler

```
public void writeMyObject(String path, MyClass object) throws
                                                        IOException {
   try (FileOutputStream fileOut =
        new FileOutputStream(path + object.getName() + ".ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
       out.writeObject(object);
   }
}


public MyClass readMyClass(String path, String name) throws
                            IOException, ClassNotFoundException {
   MyClass object;
   try (FileInputStream fileIn = new FileInputStream(path + name);
        ObjectInputStream in = new ObjectInputStream(fileIn)) {
       object = (MyClass) in.readObject();
   }
   return object;
}
```

Om man inte arbetar med text utan bara vill spara något man har i minnet "rakt av" kan man använda binära filer.

- Man dumpar eller läser bytes direkt från/till disk.
- Filerna blir inte "läsbara", finns inga rader.
- I Java används olika binära strömmar för binär läsning och skrivning, t.ex. FileOutputStream.

Ett intressant sätt att använda binärfiler är att skriv ut/läsa in objekt till/från disk.

- För att det skall fungera måste klassen objekten tillhör implementera gränssnittet Serializable.
  - Vi skriver … implements Seializable efter klassnamnet.
- Vad ett gränssnitt är tas upp senare i kursen.

Inför Övning 5

streameditor