HoGent

BEDRIJF
EN
ORGANISATIE

H15: Files en streams

Leerinhoud files en streams

- Tekstbestand schrijven
- Tekstbestand lezen
- Objecten in een bestand wegschrijven
- Objecten uit een bestand lezen

Studiefiche:

- Kan een gepaste file/stream correct implementeren in Java

15.1 Inleiding

- Lange termijn opslag van grote hoeveelheden gegevens
- Persistente gegevens (blijven bestaan na beëindiging van een programma)
- Opgeslagen op secundaire geheugenmedia:
  - Magnetische schijven of tapes
  - Optische schijven
- Bestaan uit records (gerelateerde velden)
- Organisatie van de records geeft
  - Sequentiële files
  - Random access files

Sequential versus random access file

- Sequentiële file

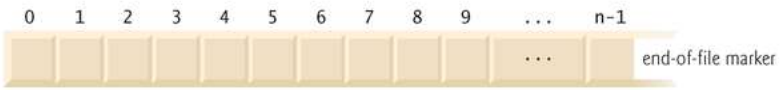
  Je doorloopt de gegevens in een sequentiële file van voor naar achter. Het duurt dus langer om een gegeven dat zich achteraan in het bestand bevindt, te bereiken.

- Random access file

  Je krijgt directe toegang tot de gegevens (daarom ook wel direct access genoemd). De tijd om het gegeven te bereiken is dus even groot, om het even op welke positie het gegeven zich bevindt.

15.2 Files en streams

- Java ziet elke file als een stream van bytes.



- Elk besturingssysteem bepaalt het einde van een file door:
  - end-of-file merkteken
  - het totaal aantal bytes van de file bij te houden
- Een java programma krijgt een signaal van het besturingssysteem wanneer het programma het einde van een stream bereikt door middel van
  - een EOFExceptie of
  - een specifieke returnwaarde van een methode

Streams ondersteunen meerdere vormen van gegevens: van eenvoudige bytes tot geavanceerde objecten.

1. Streams die bytes in- en uitvoeren zijn **byte-based streams**, de gegevens worden voorgesteld in een binair format
   vb. 2 bytes voor een char, 4 bytes voor een int, 8 bytes voor een double, …

   => Files die gecreëerd worden door gebruik te maken van byte-based streams zijn **binary files.**

2. Streams die karakters in- en uitvoeren zijn **character-based streams**, de gegevens worden voorgesteld als een sequentie van karakters.
   vb. Het getal 1 miljoen bevat 7 cijfers (een 1 gevolgd door 7 nullen), dus dit zijn 7 karakters, die elk 2 bytes innemen: in totaal zijn dus 14 bytes nodig om dit getal op te slaan. Als je daarentegen het getal 8 wilt opslaan, dan is dat slechts 1 karakter, dat dus slechts 2 bytes inneemt.

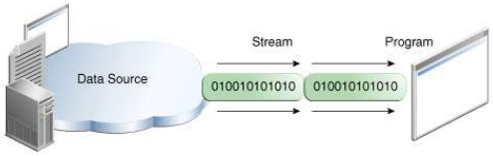
   => Files die gecreëerd worden door gebruik te maken van character-based streams zijn **text files.**

Streams

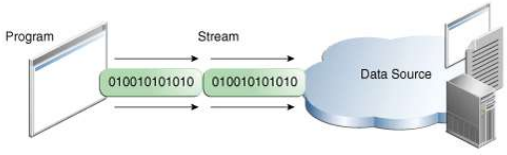
- Een stream is een manier om allerlei vormen van gegevenstransport in een computer of tussen computers voor te stellen: gegevens die via het toetsenbord binnenkomen, of via een netwerkkabel, of gegevens die via twee programma's worden uitgewisseld.

- Java associeert streams met devices:
  - `System.in:` standaard input stream object, laat toe om bytes via het toetsenbord in te lezen
  - `System.out:` standaard output stream object, laat toe om bytes weer te geven op het scherm
  - `System.err:` standaard error stream object, laat toe om foutboodschappen weer te geven op het scherm

- Een programma gebruikt een input stream om gegevens te lezen van een source, één item per keer:



- Een programma gebruikt een output stream om gegevens te schrijven naar een bestemming, één item per keer:



15.3 De klasse Files

Sinds Java SE 6 kunnen we naast de package java.io ook gebruik maken van java.nio, de nieuwe I/O API:

De klasse **Files** vervangt de klasse **File.**

Ze is efficiënter en bevat onder andere static methodes om:

- Bestanden en directories aan te maken en te verwijderen.
- Eigenschappen van bestanden en directories op te vragen en aan te passen.
- **Streams aan te maken om bestanden te lezen en te schrijven** – zie verder.

Bewerkingen

- Path f = Files.createFile(“MyCode.txt”);
- Files.delete(f);

Compatibiliteit met de klasse File

Wanneer een programma toch nog gebruik moet maken van File-objecten, kunnen we een Path omzetten naar een File met de methode toFile() van de interface Path.

Ook de omgekeerde bewerking is mogelijk met de methode toPath() van de klasse File.

Path Strings in verschillende besturingssystemen

Als je het Path opgeeft naar een bepaalde file of directory, dan gebruik je een bepaalde separator. In Windows is dit een backslash (\), maar in Linux of op MacOS is het een gewone slash (/).

Om geen problemen te veroorzaken, kan je best het scheidingsteken van het lokale systeem opvragen via **File.separator**

15.4 Sequential access text files

- Sequential access files slaan records op in volgorde van de record sleutel.
- Text files zijn leesbare files.

15.4.1 Creatie sequential access text file

- Java legt geen structuur op aan een file:
  - Java kent geen records
  - Bouw zelf de structuur van een file, rekening houdend met de requirements van de applicatie
- Maakt gebruik van Formatter
  - De klasse **Formatter** helpt om gegevens onder de vorm van geformatteerde data te converteren naar de interne datatypen
  - Formatter maakt gebruikt van streams → OutputStream
    - Voorbeeld: System.out
    - Voorbeeld: **Files.newOutputStream(Paths.get(bestandsnaam))**

      of Files.newOutputStream(Paths.get(bestandsnaam), StandardOpenOption.APPEND) // achteraan toevoegen
- Als het bestand nog niet bestond, wordt het gemaakt

Account record

```
public class AccountRecord
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

   public AccountRecord()
   {
      this( 0, "", "", 0.0 );       // call four-argument constructor
   }

   // … setters/getters/…
}
```


Schrijven naar een text file

```
public class CreateTextFile
 {
   private Formatter output;              // object used to output text to file
   public void openFile()                 // enable user to open file
   {
      try
      {
            output = new Formatter( Files.newOutputStream(Paths.get("clients.txt")));
      }
      catch (InvalidPathException ie)
      {     System.err.println( "Error finding file." );
            System.exit( 1 );
      }
      catch ( IOException filesNotFoundException )
      {     System.err.println( "Error creating file." );
            System.exit( 1 );
      }
   } // end method openFile
```

```
public void addRecord(AccountRecord record)     // add a record to file
{
    try
    {     // write new record
          output.format( "%d %s %s %.2f%n", record.getAccount(),
                record.getFirstName(), record.getLastName(), record.getBalance() );
     }
    catch ( FormatterClosedException formatterClosedException )
    {
          System.err.println( "Error writing to file." );
    }

} // end method addRecords

public void closeFile()              // close file
{
   if ( output != null )
      output.close();
} // end method closeFile
} // end class CreateTextFile
```



```
public class CreateTextFileApplication
{
  public static void main( String[] args )
  {
      CreateTextFile application = new CreateTextFile();

      application.openFile();
      AccountRecord record = new AccountRecord (100,”Bob”,”Jones”,24.98);
      application.addRecord(record);
      application.closeFile();

  } // end main
} // end class CreateTextFileApplication
```

Sample data			
100	Bob	Jones	24.98

Opmerkingen

- IOException → bij Formatter
  - In geval je geen schrijfrechten hebt voor de file
  - De file wordt niet gevonden of kan niet gecreëerd worden
- InvalidPathException
  - Het opgegeven pad klopt niet
- FormatterClosedException → bij Formatter
  - In geval Formatter reeds afgesloten (close) terwijl je schrijft

15.4.2 Lezen van een sequential access text file

- Maakt gebruik van Scanner
  - De klasse **Scanner** helpt om gegevens te converteren.
  - Scanner maakt gebruikt van streams → InputStream
    - Voorbeeld: System.in
    - Voorbeeld: **Files.newInputStream(Paths.get(bestandsnaam))**
- Scanner splitst de inputstream op in tokens gescheiden door whitespace karakters.
- Enkele methoden van Scanner:
  - **hasNext**(), **hasNextInt()** geven een boolean terug
  - **next**(), **nextLine**() geven een String terug
  - **nextInt**() geeft een int terug
  - …

Een sequential access text file lezen

Voorbeeld: De rekeninggegevens van enkele klanten zijn weggeschreven in een tekstbestand <clients.txt>. Ze worden opgevraagd en op het scherm getoond.

- Invoer:

Sample data

```
100    Bob     Jones    24.98
200    Steve   Doe      -345.67
300    Pam     White    0.00
400    Sam     Stone    -42.16
500    Sue     Rich     224.62
```


- Uitvoer:

```
Account  First Name  Last Name       Balance
100      Bob         Jones             24.98
200      Steve       Doe             -345.67
300      Pam         White             0.00
400      Sam         Stone           -42.16
500      Sue         Rich            224.62
```

```
public class ReadTextFileApplication
{
  public static void main( String args[] )
  {
    ReadTextFile application = new ReadTextFile();

    application.openFile();
    application.readRecords();
    application.closeFile();
  }
}
```



```
public class ReadTextFile
{  private Scanner input;

  public void openFile()
  {  try
     {
          input = new Scanner
                      (Files.newInputStream(Paths.get("clients.txt“) ) );
     }
     catch (InvalidPathException ie)
     {    System.err.println( "Error finding file." );
          System.exit( 1 );
     }
     catch ( IOException filesNotFoundException )
     {    System.err.println( "Error reading file." );
          System.exit( 1 );
     }
  }
```

```
public void readRecords()
{ // object record voor uitvoer op het scherm
  AccountRecord record = new AccountRecord();
  System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
    "First Name", "Last Name", "Balance" );

  try       // lees records v/d file via het Scanner object input
  {  while ( input.hasNext() )
    {  record.setAccount( input.nextInt() );          // read account number
       record.setFirstName( input.next() );           // read first name
       record.setLastName( input.next() );            // read last name
       record.setBalance( input.nextDouble() );       // read balance
       // toon de record inhoud

       System.out.printf( "%-10d%-12s%-12s%10.2f\n",
         record.getAccount(), record.getFirstName(),
           record.getLastName(), record.getBalance() );
    }
  }
```

HoGent 23

```
        catch ( InputMismatchException elementException )
        {     System.err.println( "File improperly formed." );
              input.close();
              System.exit( 1 );
        }
        catch ( NoSuchElementException elementException )
        {     System.err.println( "Element missing");
              input.close();
              System.exit( 1 );
        }
        catch ( IllegalStateException stateException )
        {
              System.err.println( "Error reading from file." );
              System.exit( 1 );
        }
     } // end method readRecords

     public void closeFile()
     {
        if ( input != null )
           input.close();          // close file
     }
  } // end klasse ReadTextFile
```

HoGent 24

Opmerkingen

- IOException -> bij Scanner
  - Indien het opgegeven bestand niet wordt gevonden.
- NoSuchElementException
  - Er ontbreken elementen (thrown by nextElement)
- InputMismatchException
  (subklasse van NoSuchElementException)
  - Indien de organisatie/type gegevens niet overeenstemmen.
- IllegalStateException
  - In geval van lezen terwijl Scanner reeds gesloten is.

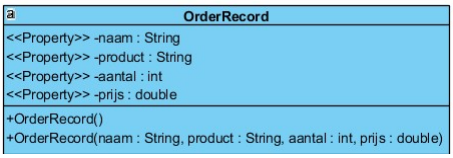
Openen, verwerken en sluiten van een file in 1 methode via try-with-resources:

```
try (Formatter out = new Formatter(.....))
{
        /* Werken met de stream */
}
catch(IOException ex)
{
        /* De gepaste exceptions opvangen en afhandelen */
}
```

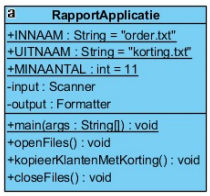
➔ **In deze notatie openen we de stream tussen de haakjes van de try-structuur. De stream die we hier openen, zal automatisch worden afgesloten na afloop van de try-structuur, ook wanneer er exceptions optreden.**

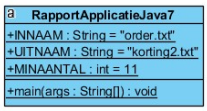
Oefening 1

Maak een rapport <korting.txt> (tekstbestand) van alle klanten die meer dan 10 eenheden bestellen.
Geef daarbij 5% korting op het te betalen bedrag.
De gegevens staan in een tekstbestand <orders.txt>.



Bij RapportApplicatieJava7 werken we met de try-with-resources.





Oefening 1

- Voorbeeld van invoerbestand:
  <orders.txt>(naam klant, product, aantal, eenheidsprijs)

Janssens	appels	50	40
Peeters	peren	5	7.5
Gunter	kiwi	100	100
Klaas	citroen	10	20
Pjotr	speculoos	30	20

- Resultaat van uitvoerbestand:
  <korting.txt> (naam klant, product, aantal, nieuwe_eenheidsprijs)

Janssens	appels	50	38.00
Gunter	kiwi	100	95.00
Pjotr	speculoos	30	19.00

15.4.4 Updating sequential access text file

- Data in vele sequential files kunnen niet gewijzigd worden zonder het risico dat de andere data vernietigd worden.
- Bijvoorbeeld: het artikel “peer” wordt gewijzigd in “kweepeer”, het oude kan niet zomaar overschreven worden want het nieuwe heeft meer ruimte nodig.
- Velden in een text file kunnen in grootte variëren.
- Ipv records in een sequential access file te wijzigen, wordt vaak de volledige file herschreven.
- De volledige file herschrijven voor 1 record is (te)veel werk, dus maar nuttig indien meerdere wijzigingen moeten doorgevoerd worden.

15.5 Object serialisatie (niet in boek!)

- Om een object te lezen uit of te schrijven in een file, voorziet Java **object serialization.**
- Een **serialized object** wordt voorgesteld door een sequentie van bytes die naast de **data** ook het **type** van de informatie van het object bevat.
- Objecten van klassen die de interface **Serializable** implementeren, kunnen geserialiseerd worden via **ObjectOutputStream**.

Account record

```
public class AccountRecord implements Serializable
{
  private int account;
  private String firstName;
  private String lastName;
  private double balance;

  public AccountRecord()
  {
    this( 0, "", "", 0.0 );       // call four-argument constructor
  }

  // … setters/getters/…
}
```


15.5.1 Aanmaken van een sequential file van objecten

ObjectOutputStream schrijft enkel serialized objecten weg.

```
try (ObjectOutputStream output =
            new ObjectOutputStream(
                  Files.newOutputStream(
                        Paths.get("c:\\oef\\test.ser"))))

  { //Schrijf de objectdata naar de file:
      output.writeObject(object of lijst van objecten);
  }
catch (IOException ex)
  { /* De gepaste exceptions opvangen en afhandelen */ }
```

- Serializable is een tagging interface => bevat geen abstracte methodes; klassen die deze interface implementeren behoren tot een bepaalde set van klassen. Als een klasse de interface Serializable implementeert, is het lid van de Serializable klassen.
- Alles wat **transient** wordt gedeclareerd, zal genegeerd worden tijdens het serialisatieproces.
- Alle primitieve-type variabelen zijn serializable.

=> In een klasse die Serializable implementeert, moet iedere variabele Serializable zijn.

```
public class TreeRecord implements Serializable
{
    private transient int aantalKnopen;
    private Tree t;
    public TreeRecord(int aantal, Tree t)
    {
        aantalKnopen = aantal;
        this.t = t;
    }
    public int getAantalKnopen() {return aantalKnopen;}
    public Tree getTree() {return t;}
}

public class Tree implements Serializable
{ ...}
```

15.5 Object deserialisatie

- Nadat een serialized object in een file geschreven is, kan het gelezen worden uit de file en **deserialized** om het object opnieuw in het geheugen te creëren.
- Objecten van klassen die de interface **Serializable** implementeren, kunnen gedeserialiseerd worden via **ObjectInputStream**.

15.5.2 Lezen van een sequential file van één object - deserialisatie

ObjectInputStream leest enkel serialized objecten

```
try (ObjectInputStream input =
    new ObjectInputStream(
            Files.newInputStream(
                Paths.get(bestand))))
    {
        AccountRecord record =
                ( AccountRecord) input.readObject();
        List < AccountRecord> records =
            (List<AccountRecord>) input.readObject();
    }
  catch (IOException ex)
   { /* De gepaste exceptions opvangen en afhandelen */ }
```

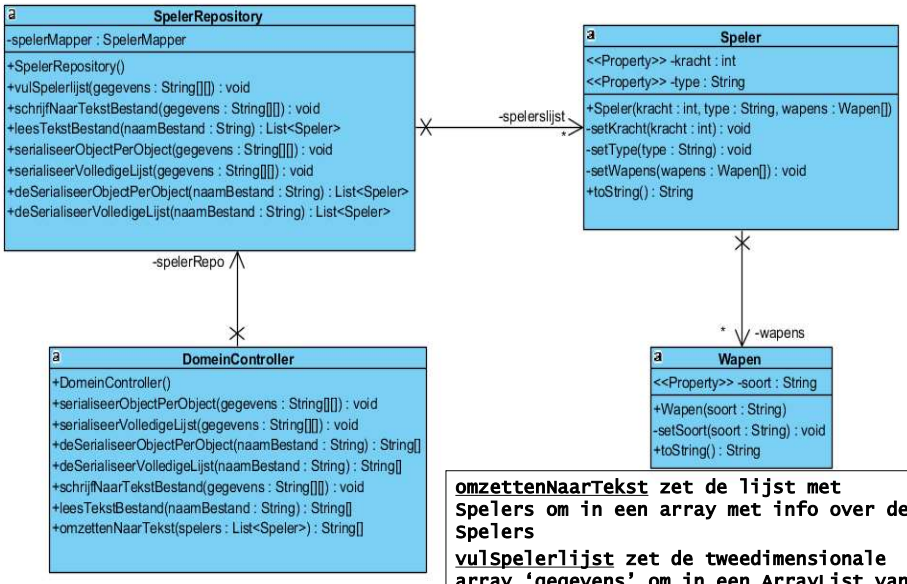
readObject() geeft een Object terug of een lijst van objecten

15.5.3 Lezen van een sequential file van objecten - deserialisatie

```
List<AccountRecord> data = new ArrayList<>();
File filenaam = new File("c:\\oef\\test.ser");

try (ObjectInputStream input =
      new ObjectInputStream(Files.newInputStream(Paths.get(filenaam))))
{
      while (true) {
            data.add((AccountRecord) input.readObject());
      } //end-while
} //end-try
catch (EOFException e) //Einde file aangeduid met EOFException
{ // OK eof bereikt   }
catch (ClassNotFoundException e)
{     System.err.print("ongeldige objectstream"); System.exit(1);  }
catch (IOException e)
{     System.err.print(e.getMessage()); System.exit(1); }
```


Oefening 2 – 3 – 4 – 5: domein



omzettenNaarTekst zet de lijst met Spelers om in een array met info over de Spelers

vulSpelerlijst zet de tweedimensionale array 'gegevens' om in een ArrayList van Speler (= attribuut spelerslijst van de repository)

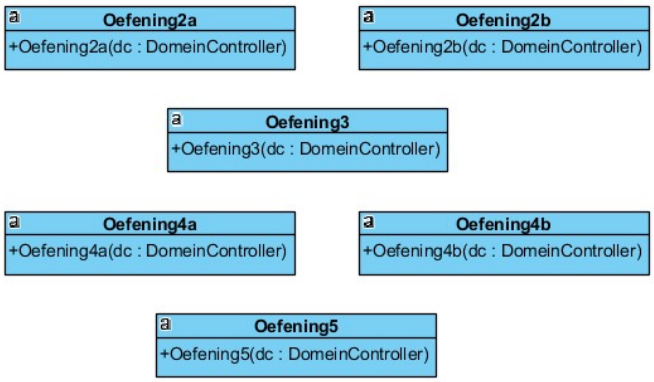
Oefening 2 – 3 – 4 – 5: persistentie

`voegSpelerToe` voegt aan de Spelerslijst (= 4de parameter)
een speler toe ahv de eerste 3 parameters

`zonderspaties` vervangt alle spaties door een underscore

`zonderUnderscores` vervangt alle underscores door een spatie

Oefening 2 – 3 – 4 – 5: ui



Bewaren

- Oefening 2: gebruik **serialisatie**
  - Oefening 2a: Creëer een bestand **oef2a.ser** en schrijf er de geserialiseerde objecten één per één naar toe ahv **serialiseerObjectPerObject**
  - Oefening 2b: Creëer een bestand **oef2b.ser** en serialiseer de lijst met de spelers in één instructie ahv **serialiseerVolledigeLijst**
- Oefening 3: schrijf een ˝**plain**˝ tekstbestand **oef3.txt**
  - Creëer een bestand en schrijf lijnen tekst, één per Speler-object, met als scheidingsteken tab ahv **schrijfNaarTekstBestand**

```
50      elf         zwaard stof
200     trol        aks blote_handen (spatie -> _)
120     tovenaar    vloek onzichtbaar
```


Terughalen

- Oefening 4: gebruik **deserialisatie**
  - Oefening 4a: Lees het aangemaakte bestand **oef2a.ser** van oef 2a terug uit en bouw de lijst met spelers terug op ahv **deSerialiseerObjectPerObject**
  - Oefening 4b: Lees het aangemaakte bestand **oef2b.ser** van oef 2b terug uit en bouw de lijst met spelers terug op ahv **deSerialiseerVolledigeLijst**
- Oefening 5: lees een ˝**plain**˝ tekstbestand
  - Lees het aangemaakte bestand **oef3.txt** van oef 3 terug uit en bouw de lijst met spelers terug op ahv **leesTekstBestand** (_ -> spatie)

15.6 Openen en bewaren van bestanden met JavaFX FileChooser

- FileChooser klasse=> javafx.stage package
- FileChooser laat toe om te navigeren in jouw bestanden en een bestand te kiezen.
- Kan gebruikt worden om
  - een bestand te openen, of
  - een bestand te bewaren.
- Analoog is er ook een DirectoryChooser om een map te kiezen

15.6 Openen en bewaren van bestanden met JavaFX FileChooser

- Volgende code creëert een FileChooser-object, stelt een titel in en toont het op een gegeven stage:

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Open Resource File");
fileChooser.showOpenDialog(stage);
```

- Je kan ook een directory initialiseren:

```
fileChooser.setInitialDirectory(new File(System.getProperty("user.home")) );
```

Bewaren van bestanden

- De FileChooser API geeft ook de mogelijkheid om een bestand in een bepaalde folder op te slaan.
- De showSaveDialog methode opent een bewaar-dialoogvenster.

```
FileChooser fileChooser1 = new FileChooser();
fileChooser1.setTitle("Save Image");
File file = fileChooser1.showSaveDialog(stage);
```


Bijkomend leermateriaal

- http://docs.oracle.com/javase/tutorial/essential/io/index.html